Modeling Readability to Improve Unit Tests

Ermira Daka, José Campos, and Gordon Fraser
University of Sheffield
Sheffield, UK

Jonathan Dorn and Westley Weimer
University of Virginia
Virginia, USA

ABSTRACT

Writing good unit tests can be tedious and error prone, but even once they are written, the job is not done: Developers need to reason about unit tests throughout software development and evolution, in order to diagnose test failures, maintain the tests, and to understand code written by other developers. Unreadable tests are more difficult to maintain and lose some of their value to developers. To overcome this problem, we propose a domain-specific model of unit test readability based on human judgements, and use this model to augment automated unit test generation. The resulting approach can automatically generate test suites with both high coverage and also improved readability. In human studies users prefer our improved tests and are able to answer maintenance questions about them 14% more quickly at the same level of accuracy.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

Keywords. Readability, unit testing, automated test generation

1. INTRODUCTION

Unit testing is a popular technique in object oriented programming, where efficient automation frameworks such as JUnit allow unit tests to be defined and executed conveniently. However, producing good tests is a tedious and error prone task, and over their lifetime, these tests often need to be read and understood by different people. Developers use their own tests to guide their implementation activities, receive tests from automated unit test generation tools to improve their test suites, and rely on the tests written by developers of other code. Any test failures require fixing either the software or the failing test, and any passing test may be consulted by developers as documentation and usage example for the code under test. Test comprehension is a manual activity that requires one to understand the behavior represented by a test — a task that may not be easy if the test was written a week ago, difficult if it was written by a different person, and challenging if the test was generated automatically.

How difficult it is to understand a unit test depends on many factors. Unit tests for object-oriented languages typically consist of sequences of calls to instantiate various objects, bring them to appropriate states, and create interactions between them. The particular choice of sequence of calls and values can have a large impact on the resulting test. For example, consider the pair of unit tests shown in Figure 1. Both tests exercise the same functionality with respect to the constructor of the class `ObjectHandlerAdaptor` in the Xineo open source project (which treats `null` and `rootHandler0` arguments identically). Despite this identical coverage of the subject class in practice, they are quite different in presentation.

```
ElementName elementName0 = new ElementName("", "");
Class<Object> class0 = Object.class;
VirtualHandler virtualHandler0 = new VirtualHandler(
    elementName0, (Class) class0);
Object object0 = new Object();
RootHandler rootHandler0 = new RootHandler((ObjectHandler
    ) virtualHandler0, object0);
ObjectHandlerAdapter objectHandlerAdapter0 = new
    ObjectHandlerAdapter((ObjectHandlerInterface)
    rootHandler0);
assertEquals("ObjectHandlerAdapter",
    objectHandlerAdapter0.getName());
```

```
ObjectHandlerAdapter objectHandlerAdapter0 = new
    ObjectHandlerAdapter((ObjectHandlerInterface) null);
assertEquals("ObjectHandlerAdapter",
    objectHandlerAdapter0.getName());
```

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

In terms of concrete features that may affect comprehension, the first test is longer, uses more different classes, defines more variables, has more parentheses, and has longer lines. The visual appearance of code in general is referred to as its *readability* — if code is not readable, intuitively it will be more difficult to perform any tasks that require understanding it. Despite significant interest from managers and developers [8], a general understanding of software readability remains elusive. For source code, Buse and Weimer [7] applied machine learning on a dataset of code snippets with human annotated ratings of readability, allowing them to predict whether code snippets are considered readable or not. Although unit tests are also just code in principle, they use a much more restricted set of language features; for example, unit tests usually do not contain conditional or looping statements. Therefore, a general code readability metric may not be well suited for unit tests.

In this paper, we address this problem by designing a domain-specific model of readability based on human judgements that applies to object oriented unit test cases. To support developers in deriving readable unit tests, we use this model in an automated approach to improve the readability of unit tests, and integrate this into an automated unit test generation tool. We present:

- An analysis of the syntactic features of unit tests and their importance based on human judgement (Section 2.).
- A regression model based on an optimized set of features to predict the readability of unit tests (Section 2.).

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786838

- A technique to automatically generate more readable tests (Section 3.).
- An empirical comparison between code and test readability models (Section 4.).
- An empirical evaluation of the test improvement technique (Section 4.).
- An empirical evaluation of whether humans prefer the tests optimized by our technique to the non-optimized versions (Section 4.).
- An empirical study into the effects of readability on test understanding (Section 4.).

Analysis of 116 syntactic features of unit tests shows that readability is not simply a matter of the overall test length; several features related to individual lines, identifiers, or entropy have a stronger influence. Our optimized model using 24 of these features results in a model with a correlation of 0.79 with the user data, which improves over the ability of general code readability models. Our technique to improve tests succeeds in increasing readability in more than half of all automatically generated unit tests, and validation with humans confirms that the optimized tests are preferred. Although we observe a reduction in the time humans spend on understanding these tests, our experiments point out that understandability goes beyond readability: For example, understanding exceptional behavior appears to be more difficult than understanding regular behavior, even if a test with exceptions may be more readable. This suggests potential for future work to improve test understandability.

2. UNIT TEST READABILITY METRIC

The source code readability metric by Buse and Weimer [7] is built on a dataset of human annotator ratings, where each code snippet received readability ratings in the range of 1 to 5. Our aim is to create a predictive model that tells us how readable a given unit test is. Whereas Buse and Weimer trained a classifier to distinguish between readable and less readable code, our aim goes beyond this: We would like to use the model to guide test generation in producing more readable tests. Therefore, we desire a regression model that predicts relative readability scores for unit tests.

Our overall approach begins with producing a dataset of tests annotated with numeric human ratings of readability. We then identify a range of syntactic features that may be predictive of readability (e.g., identifier length, token entropy, etc.). We then use supervised machine learning to construct a predictive model of test case readability from those features. To predict the readability of a new test case we calculate its feature values and apply the learned model. In this paper, we use a simple linear regression learner [39], although in principle other regression learners are also applicable (e.g., multilayer perceptron [39]). However, in linear regression the resulting model (which consists of weightings for individual features) can easily be interpreted, and the learning is quick, which facilitates the selection of suitable subsets of features.

In this section, we describe how we collected the data to learn this model, the features of unit tests we considered, and the machine learning we applied to create the final model.

2.1 Human Readability Annotation Data

Both the test cases considered and the human annotators chosen influence the quality of our readability model. While it is relatively easy to assemble a diverse group of unit tests, particular attention must be paid to participant selection and quality in this sort of human study. For scalability we used crowdsourcing to obtain participants, but found that the use of a qualification test is critical for such crowdsourced participants.

Supervised learning requires a training set. In this work we used a number of diverse and indicative open-source Java projects: Apache commons, poi, trove, jfreechart, joda, jdom, itext and guava. Each of these projects comes with an extensive test suite of developer-written unit tests. In addition, we applied the EvoSuite [20] unit test generation tool in its default configuration to produce a branch coverage test suite for each of the projects. Training tests were then selected manually with the aim to achieve a high degree of diversity (e.g., short and long tests, tests with exceptions, if-conditions, etc.)

To collect the human annotator data, we used Amazon Mechanical Turk,[1] and asked crowd workers to rate unit tests on a scale of 1 to 5 using the presentation setup of Buse and Weimer [7]. As in previous work, annotators were not given a formal definition of what to consider readable, and were instead instructed to rate code purely based on their subjective interpretation of readability. However, while the original Buse and Weimer survey involved undergraduates from the same institution, we find that crowdsourcing leads to a much broader diversity of annotator expertise which must be accounted for to learn a useful model.

We began by assessing the utility of general crowd worker responses for this task. We selected 100 test cases from our eight benchmark projects, including developer-written as well as automatically generated tests, and collected 2,388 human ratings for these test cases (i.e., each annotator rated 15–32 test cases). To evaluate the quality of these responses, we measured the inter-annotator agreement by calculating the average Pearson's correlation between each annotator and the average test scores. The inter-annotator agreement in this data set, generated from participants with no expertise requirements or filtering, is only weak, with a value of 0.25.

This low correlation could arise for many reasons, including an unsuitable choice of test cases or insufficient qualification of the human annotators. To investigate this issue, we manually selected 50 test cases that are examples of either very high quality (e.g., concise, well documented, well formatted) or very low quality (e.g., long, complex, badly formatted). The initial selection was made by the first author, and this set was refined by iterations with another author, retaining only tests with unison agreement. For these test cases we again gathered human annotator scores and measured the inter-annotator agreement. The results confirmed the need to require annotator expertise. The agreement was even lower than on the first set of tests: The inter-annotator agreement in this experiment was only 0.2, which leads us to the conclusion that the more likely explanation is insufficient qualification of the human annotators.

2.2 Final Annotation Data Set

Based on these pilot studies, we designed our final experiment to use a qualification test. This qualification test consisted of four questions of understanding based on example Java code. Only human annotators who correctly answered three out of the four questions were allowed to participate. Our observations that crowdsourced participants can be fruitfully used for such human studies [27, 37] provided that care is taken to avoid participants who are simply trying to "game the system" [23] is consistent with previous work.

Our final experiment gathered data on 450 human- and machine-written test cases, ultimately obtaining 15,669 human readability scores. We conducted the experiment in stages, initially focusing on all tests equally but subsequently gathering additional annotations on particular tests to ensure that each test feature considered (see Section 2.3) had enough annotations for machine learning purposes. Restricting attention to qualified participants increased the inter-annotator agreement substantially, to 0.5. In addition, we generated 200 pairs of tests using the EvoSuite tool, such that each pair had the same coverage quality but a different textual representation. For these pairs of tests we gathered a separate set of forced-choice judg-

[1] `http://aws.amazon.com/mturk/`, accessed 03/2015.

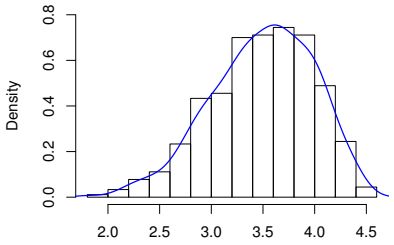


Figure 2: Score distribution for the human test annotation dataset.

ments (i.e., annotators were asked to select which of the two tests were the most readable) for use in feature selection (see Section 2.5).

Figure 2 shows the underlying frequency distribution of readability scores for the 450 tests. The histogram shows that there are few tests with very high or very low scores, and the majority of scores is in the range from 3.0-4.0. We note that our distribution of test readability is quite different from the bimodal distribution of source code readability observed by Buse and Weimer [7], motivating the need for a test-specific model of readability.

2.3 Features of Unit Tests

The readability of a unit test may depend on many factors. We aggregated features by combining structural, logical complexity, and density code factors. All features used by Buse and Weimer [7] are included in this set, as well as the entropy and Halstead features used by Posnett et al. [34]. Additionally, we included the following new features:

Assertions: This feature counts the number of standard JUnit assertions. Additionally we included a binary feature *has assertions* which has value 1 if a test case contains assertions, and 0 otherwise.

Exceptions: We propose a feature to measure exceptions since exceptional behavior is handled differently from regular behavior in tests. As we did not encounter any tests with more than one expected exception, we use a binary feature *has exceptions*.

Unused Identifiers: Unit tests are typically short and contain few variables. In our anecdotal experience with EvoSuite we found that developers do not prefer tests that define but never use variables.

Comments: We count the number of lines for single-line ("//") and multi-line ("/* ... */") comments. However, as EvoSuite and other tools generate comments mainly within the catch block of an expected exception, where it shows the error message of the exception, we refined the comments feature to two versions, one that counts regular comments, and one that counts the comments within catch blocks. Comments are removed before calculating code-specific features (e.g., features based on numbers, classes) but included for general presentational features (e.g., line length).

Token features: We identified several additional common syntactic features not captured in past readability models. In particular, we observe that unit test generation tools often tend to include defensive (sometimes redundant) casts. Furthermore, we propose a feature to count the overall tokens identified by the parser. Finally, we refined the "single character occurrence" feature used by Buse and Weimer, which counts the number of occurrences of different special characters (parenthesis, quotation marks, etc.) with a feature that counts all the special characters (*single characters*).

Datatype features: Different primitive data types have a possible impact on readability. Hence, we propose features based on the occurrence of the value null, Boolean values (true and false), array accesses, type constants (e.g., Object.class), floating point numbers, digits, strings, characters, and the length of strings. We also added a feature measuring the "English-ness" of string literals, using the language model of Afshan et al. [1].

Statement features: We propose features to count different types of statements, in particular constructor calls, field accesses, and method invocations.

Class and method diversity: In addition to diversity, as captured by entropy features at the level of tokens and bytes, we also propose features to capture diversity in terms of the classes, methods, and identifiers used. For each of these, we include a feature counting the unique number as well as the ratio of unique to total number.

Table 1 lists all the candidate features, showing which of the features we used in terms of the **total** value for the unit test, the **average** value per line, and its **max** value in any line in the test case. In total, we considered 116 candidate features (in Section 2.5 we use feature selection to build our model from the 24 most relevant).

To analyze the influence and predictive power of the individual features, Table 1 shows the Pearson's correlation between each feature and the average test scores (ranging from a weak positive correlation of 0.21 for average blank lines to a strong negative correlation of -0.5 for the maximum line length and total identifier length), and the result of a one-feature-at-a-time analysis. For the latter, we train a linear regression model using only one feature, and determine the correlation with 10-fold cross validation. We also considered a leave-one-feature-out analysis, where one measures the effect of a feature by in terms of the difference between a model learned using all features and with all but the feature under consideration; however, leave-one-out analyses are not applicable in the presence of feature overlap and due to our very large set of related features the results are not representative. Finally, we also applied the Relief-F method [36], which agrees with the one-feature-at-a-time analysis and is thus omitted for brevity.

2.4 Feature Discussion

Considering that a unit test is often simply a sequence of calls, one would expect the length of that sequence to be one of the main factors deciding on the readability. However, as shown in Table 1, the number of statements (test length) on its own surprisingly only has weak predictive power. However, other features related to the length have a larger influence on the readability. In particular, the line length plays an important role, both in terms of maximum line length as well as the total line length. The maximum line length presumably is important because a test case can have bad readability even if most lines are short and only a single line is very long. The "total" line length essentially amounts to the total number of characters in the test and thus is a better representation of length than the number of statements.

We furthermore observe that the identifiers in a test have a large influence on its readability. This refers to features related to the number of identifiers, their length, and their diversity, and is a challenge for test generation tools, which typically use simple heuristics to derive names for variables. The diversity in general results in important features, for example captured by byte and token entropy [34].

Only a few features are positively correlated with test readability: comments have a weak positive correlation, as does the ratio of blank lines (avg. blank lines). Surprisingly, exceptions also have very weak positive correlation. We expected assertions to show a strong influence on readability, but there is no correlation between assertions and the test score, and the predictive power is weak.

The small influence of loops and conditional statements to some extent may be attributed to our choice of test cases for annotation: Many of the tests are generated by EvoSuite, which generates only tests that are sequences of calls. The manually written tests with loops and conditional statements included in the dataset tend to be short and well-formatted, contributing to the small but positive influence of these features.

Table 1: Predictive power of features based on correlation and one feature at a time analysis, and optimized regression model.

	Correlation			One feature a time				Correlation			One feature a time		
Feature name	total	max	avg	total	max	avg	Feature name	total	max	avg	total	max	avg
identifier length	**-0.50**	**-0.42**	**-0.46**	0.50	0.41	0.45	commas	-0.15	-0.20	-0.15	0.13	0.14	0.13
line length	**-0.45**	**-0.50**	-0.43	0.4	0.49	0.41	halstead difficulty	-0.15	-	-	-	-	-
constructor calls	**-0.45**	-	-0.24	0.44	-	0.20	**has exceptions**	**0.15**	-	-	0.11	-	-
byte entropy	-0.39	-	-	0.31	-	-	**identifier ratio**	**0.15**	-	-	0.11	-	-
unique identifiers	**-0.37**	-0.25	-0.14	0.36	0.22	0.11	method invocations	-0.14	-0.06	-0.03	0.09	-0.00	-0.07
identifiers	**-0.36**	-0.23	-0.29	0.36	0.20	0.27	**string length**	-0.14	-0.24	**-0.20**	0.09	0.23	0.19
assignments	-0.33	-	-0.16	0.32	-	0.13	arrays	-0.14	-0.05	-0.06	0.11	-0.05	-0.01
casts	-0.33	-0.33	-0.28	0.32	0.32	0.26	indentation	-0.13	0.08	-0.01	0.10	0.02	-0.25
parentheses	-0.31	-	-0.28	0.30	-	0.26	field accesses	-0.13	-0.14	-0.17	0.11	0.11	0.15
keywords	-0.30	-0.28	-0.17	0.28	0.27	0.14	halstead effort	-0.13	-	-	0.13	-	-
halstead volume	-0.27	-	-	0.18			**assertions**	**-0.12**	-	-0.04	0.09	-	-0.10
distinct methods	**-0.27**	-0.05	0.00	0.26	-0.02	-0.18	additional assertions	-0.12	-	-	0.09	-	-
single characters	-0.26	-0.32	-0.24	0.17	0.22	0.16	**nulls**	-0.12	**-0.20**	**-0.15**	0.06	0.19	0.12
periods	-0.25	-0.22	-0.17	0.24	0.20	0.13	class ratio	-0.10	-	-	0.04	-	-
comparison operations	-0.25	-	-0.23	0.24	-	0.23	blank lines	0.08	-	0.21	0.03	-	0.19
tokens	-0.24	-0.28	-0.21	0.15	0.17	0.14	unused identifiers	-0.07	-	-	0.02	-	-
digits	-0.24	-0.24	-0.19	0.21	0.22	0.161	string score	0.06	-	-	0.01	-	-
token entropy	**-0.24**	-	-	0.17	-	-	strings	-0.05	-0.19	-0.13	0.00	0.17	0.10
floats	**-0.22**	-0.21	-0.17	0.20	0.19	0.14	excep. comments	0.05	-	0.15	-0.00		0.12
comments	0.20	-	0.04	-	0.18	-0.01	**arithmetic operations**	-0.04	-	**0.03**	-0.07	-	-0.05
test length	-0.19	-	-	0.13	-	-	**branches**	0.04	-	**0.06**	0.02	-	0.02
numbers	**-0.19**	-0.18	-0.07	0.18	0.17	0.04	types	-0.02	-0.08	-0.04	-0.11	0.05	-0.02
spaces	-0.19	-	-0.17	0.13	-	0.13	**has assertions**	**0.01**	-	-	-0.14	-	-
loops	0.19	-	**0.18**	0.17	-	0.17	**method ratio**	**0.01**	-	-	-0.21	-	-
booleans	-0.16	-0.17	-0.01	0.14	0.13	-0.23	**characters**	0.00	**-0.03**	0.04	-0.15	-0.08	-0.01

$$\begin{aligned}\text{test\_score} = {} & -0.0001 \times \text{total\_line\_length} - 0.0021 \times \text{max\_line\_length} + 0.0076 \times \text{total\_identifiers} - 0.0004 \times \text{total\_identifier\_length} - 0.0067 \times \text{max\_identifier\_length} - 0.005 \times \text{avg\_identifier\_length} \\ & + 0.0225 \times \text{avg\_arithmetic\_operations} + 0.9886 \times \text{avg\_branches} + 0.1572 \times \text{avg\_loops} + 0.0119 \times \text{total\_assertions} - 0.0147 \times \text{total\_has\_assertions} + 0.1242 \times \text{avg\_characters} \\ & - 0.043 \times \text{total\_class\_instances} - 0.0127 \times \text{total\_distinct\_methods} + 0.0026 \times \text{avg\_string\_length} + 0.1206 \times \text{total\_has\_exceptions} - 0.019 \times \text{total\_unique\_identifiers} - 0.0712 \times \text{max\_nulls} \\ & - 0.0078 \times \text{total\_numbers} + 0.1444 \times \text{avg\_nulls} + 0.334 \times \text{total\_identifier\_ratio} + 0.0406 \times \text{total\_method\_ratio} - 0.0174 \times \text{total\_floats} - 0.3917 \times \text{total\_byte\_entropy} + 5.7501\end{aligned}$$

2.5 Feature Selection

Feature selection is a widely used technique that reduces the dimension of a dataset with respect to a given value [12]. Selecting a subset of potential inputs from the total feature set can help on compacting relevant information, and removing the noise in prediction [31]. To improve the learning process and the generality of the resulting model, it is thus desirable to reduce the number of features using feature selection techniques.

Feature selection techniques are classified [15, 18] in two main categories, called filter and wrapper models. A filter model typically consists of removing features that are shown to have low predictive power. For example, as a baseline we considered the use of correlation and the Relief-F filter model method [36], which select 43 and 21 features leading to a correlation of 0.65 and 0.7 respectively. We desire a higher quality feature set, however, and thus focus on wrapper model feature selection.

A wrapper model selects subsets of variables considering the learning method as a black box, and scoring inputs based on their predictive power. We considered forward and backward feature selection; in forward selection one starts from an empty set of features, and iteratively adds features based on the resulting predictive power. In backward selection the starting point is the full set of features, and one iteratively removes individual features.

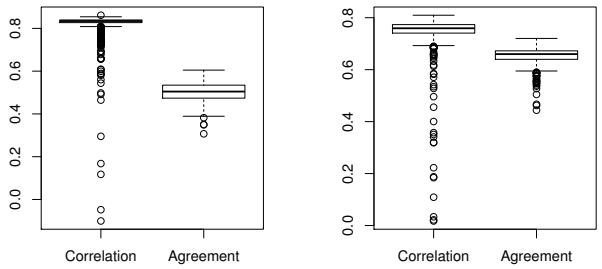
We used a steepest ascent hill climbing algorithm to perform this feature selection. That is, for forward selection we start with a randomly chosen feature, create a regression model using only that feature, and calculate the correlation using 10-fold cross-validation. Then, for each other feature, we determine the correlation of a model trained using this and the first feature. The pair with the highest correlation is the new starting point, and we explore all possible variants to add another feature. This is done iteratively, until there exists no feature that can be added while increasing the correlation value. In our experiments, forward feature selection achieved substantially better results than backward feature selection and is used for our model.

To avoid overfitting the model to the training set, in addition to standard cross-validation we used the set of 200 pairs of test cases with human annotation data described in Section 2.2. 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), using Pearson's correlation.

We applied the forward feature selection 1,000 times (see Figure 3a), and the best configuration consisting of 16 features achieved a correlation of 0.86. However, when measuring user agreement, the correlation of the same configuration is only 0.49, which suggests a certain degree of overfitting to the training data. To counter this, we re-ran feature selection, but rather than using the correlation to guide the hill-climbing, we used the sum of correlation and user agreement. As Figure 3b shows, this reduces the achieved correlation slightly, but increases the agreement substantially.

The most frequently selected feature is "total identifier length", which occurred in 90% of all runs; the second most frequently selected feature is "max line length" (82% of runs), showing the importance of this feature. Interestingly, "total exceptions" was selected in 56% of the runs, suggesting that although the predictive power of this feature on its own is not so strong, it appears independent from other features. Byte entropy (52%) is selected more frequently than token entropy (19%), which suggests that token entropy is correlated with other features.

In the end, the overall best configuration consists of 24 features, shown in bold in Table 1, and the resulting regression model is shown below the table. This combination of features achieves a



(a) Selection using correlation

(b) Selection using correlation and agreement

Figure 3: Results of the feature selection measured in terms of Pearson's correlation with 10-fold cross validation, and agreement with user preferences for test pairs.

Algorithm 1 Test Case Optimization

Require: Test case t, coverage objective c
Ensure: Optimized test case t'

```
1:  procedure OPTIMIZE(t, c)
2:      T ← GENALTERNATIVES(t, c, length(t))
3:      t' ← select highest ranked t' in T
4:      return t'
5:  procedure GENALTERNATIVES(t, c, start)
6:      T ← {t}
7:      for p ← start down to 1 do
8:          s ← statement at position p in t
9:          for all s' ∈ get all replacements for s do
10:             t' ← replace s with s' in t
11:             if t' satisfies c then
12:                 T ← T ∪ GENALTERNATIVES(t', c, p − 1)
13:     return T
```

correlation of 0.79 with a root relative squared error rate of 61.58%, and has a high user agreement (0.73).

3. GENERATING READABLE TESTS

Given a predictive model of test readability, we would now like to apply this model to improve automated unit test generation. As search-based testing is a common technique to generate unit tests, in principle it would be possible to simply include the readability prediction as a second objective in a multi-objective optimization, and thus optimize tests towards both, coverage and readability, at the same time. However, there is a dichotomy between the need for search techniques to include redundancy in the tests to explore the state space (i.e., statements that do not contribute to the code coverage), and the detrimental effects of this redundancy on the readability. Therefore, we use a post-processing technique to optimize unit tests, which has the additional benefit that it is independent of the underlying test generation technique, allowing our approach to apply to any such black box unit test generator.

Algorithm 1 describes this post-processing algorithm: We assume a test case t is a sequence of statements $t = \langle s_1, s_2, \ldots, s_l \rangle$ of length l, where each statement is either a method call, a constructor call, some other form of assignment (e.g., primitive values, public fields, arrays, etc.) or an assertion. The algorithm is given a test case t generated for coverage obligation c. It is assumed that t is minimized with respect to c; that is, removing any of the statements in t means that c is no longer satisfied.

Given these inputs, we generate the set of alternative versions of t that still satisfy c as follows: We iterate over the statements in the test from the last statement to the first statement (Line 7). For each statement we determine the possible set of replacement statements (Line 9), consisting of all possible method or constructor calls that generate the same return type. This restriction ensures that the variable defined at the statement (if any) still exists after the replacement. We only consider replacements for statements calling constructors or methods, but in the future, other types of transformations could also be integrated. Any additional parameters of the replacement call are assigned randomly chosen existing variables of the desired types, the value null, or if no variable of the required type exists, then an instance can also be generated by recursively inserting a random generator for that type and satisfying its dependencies (e.g., based on the statement inseration in EVOSUITE [20]).

For each candidate replacement t' we determine if it still satisfies coverage objective c by executing t' and observing its coverage. If it does, then we recursively apply the replacement algorithm to t' starting at the position preceding the modified statement, and keep all valid replacements. In the end, T contains the set of valid replacements for t that still satisfy c. The tests in T are then sorted by readability, and the most readable test in T is selected.

For example, consider the first test case in Figure 1, which was generated to cover the constructor of `ObjectHandlerAdapter`: Alternative generation would start with the last statement, which is an assertion, and thus is not modified. The next statement considered is the constructor call. The class `ObjectHandlerAdapter` has four different constructors, but as the one called in the original test is the coverage objective of the test, replacements with the three other constructors no longer satisfy this objective and are discarded. Because the algorithm is randomized it also attempts to replace the constructor call with a new parameter assignment. Assume it satisfies the parameter with a null reference: The coverage obligation is still satisfied, and minimization can now remove the first five statements of this alternative, as they are no longer used in the constructor call, and thus not needed in order to satisfy the coverage goal. The new test has no more statements to modify, so no further alternatives can be generated from this test, and the algorithm continues generating alternatives with the next statement of the original statement, which is the constructor call of `RootHandler`. In the end, the alternative that calls the constructor with a null value has the highest readability value (3.67 vs. 3.30 for the original test), and is chosen as replacement.

4. EMPIRICAL EVALUATION

This section contains an empirical evaluation of default test cases (i.e, test cases generated with default parameters) and test cases optimized for readability. In particular, we empirically aim to answer the following research questions:

RQ1: How does the test readability metric compare to code readability metrics?

RQ2: Can our test readability metric guide improvement of generated unit tests?

RQ3: Do humans prefer readability optimized tests?

RQ4: Does readability optimization improve human understanding of tests?

4.1 Experimental Setup

4.1.1 Unit Test Generation Tool

We have implemented the algorithm described in Section 3. in the EVOSUITE [20] tool for automatic unit test generation. EVOSUITE uses search-techniques to derive test cases with the aim to maximize coverage of a chosen target criterion (e.g., line coverage or branch coverage). After the generation, EVOSUITE applies several post-processing steps to improve readability: For each individual coverage objective (e.g., branch) a minimized test case is generated; that is, removing any statement from the test will lead to the coverage objective no longer being satisfied. In these minimized tests, primitive values are inlined to reduce the number of variables, and

then the primitive values are minimized (i.e., strings are shortened, and numbers are decremented as close as possible to 0 without violating the coverage objective). Finally, assertions are added to the tests, and minimized using an approach based on mutation analysis [22]. The readability optimization algorithm from Section 3. was integrated as a further step of this chain of optimizations.

4.1.2 Experiment Procedure

For RQ1, we used the public dataset by Buse and Weimer [7], our dataset of 450 annotated test cases (Section 2.), and the set of 200 test pairs used to support feature selection (Section 2.), and measured the correlation and agreement of different readability models.

For RQ2-4, we manually selected 30 classes from open source projects, with the criteria that they (1) are testable by EVOSUITE with at least 80% code coverage, (2) do not exhibit features currently not handled by EVOSUITE's default configuration such as GUI components, and (3) have less than 500 non-comment source statements (NCSS) and few dependencies, such that they are non-trivial yet understandable in a reasonable amount of time. For each of the chosen classes we generated 10 tests for each coverage objective (i.e., branch) to account for the randomness of the test generation approach, with and without the readability optimization introduced in this paper. Furthermore, we generated an additional 10 test cases per branch per class with both configurations, but modified EVOSUITE to generate failing assertions (i.e., during the assertion generation using mutation analysis [22] the assertions were chosen to pass on the mutants rather than the original class). To answer RQ2, we compare the default and optimized tests in terms of their readability score as predicted by our test readability model. We used the Wilcoxon-Mann-Whitney statistical symmetry test, and the Vargha-Delaney $\hat{A}_{ab}$ statistics to evaluate the significance of the optimization [4].

For RQ3, we selected three random pairs of tests for each class from the RQ2 dataset; each pair consisting of one test generated with and one without the readability optimization, both cover the same branch, and they differ in readability score. This resulted in a total of 90 pairs of tests, and we used a forced-choice questionnaire on Amazon Mechanical Turk (see Section 2.) to determine for each pair which test is preferred by users.

For RQ4, we selected 10 out of the 30 classes with large differences in readability of its tests, and for each class chose either a pair of passing or failing tests. (Note that our procedure to generate failing tests did not guarantee failing tests, hence the pass or fail status within pairs is not always identical.) To recruit students for RQ4, we invited all computer science students (undergraduate and postgraduate) at the University of Sheffield and asked them to perform a pre-qualification quiz. This quiz consisted of the four questions from the Mechanical Turk qualification plus one JUnit specific question, and we selected 30 students who answered at least 3 questions correctly. The experiment was conducted in the computer lab of the university's Department of Computer Science. All 30 selected participants received a short introduction to the experiment, and then answered 10 questions in a web browser based quiz. Each question showed a test case and provided the source code of all classes required by the test case, and asked the students to select if the test would pass or fail. After 60 minutes the students were asked to submit the answers they had produced up to that point, filled in a short survey, and were paid a fee of GBP10.

4.1.3 Threats to Validity

Construct: For RQ4, we use time and correctness of pass/fail decision to measure understanding. It is possible that using a different task that requires understanding would give a different result. For example, Ceccato et al. [13] reported a positive effect when using random tests during debugging.

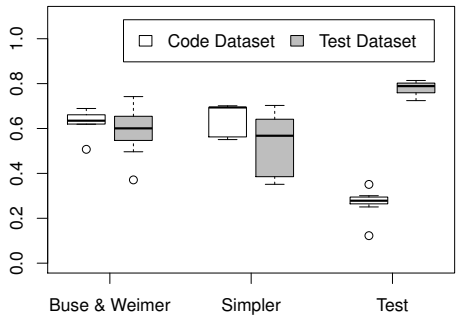


Figure 4: 10-fold cross-validation of code and test readability models using different learners and data sets.

Internal: For all experiments involving humans, the tests were assigned randomly. To avoid learning effects for RQ4, we ensured that no two tests shown to one participant originate from the same project. Participants without sufficient knowledge of Java and JUnit may affect the results; to avoid this problem we only accepted subjects who passed a qualification test. Experiment objectives may have been unclear to participants, at least for RQ4; to counter this threat we tested and revised all our material on a pilot study, and interacted with the students during experiment to ensure they understood the objectives.

External: All our experiments are based on either Amazon Mechanical Turk users, or students, and thus may not generalize to all developers [27, 37]. The set of target classes used in the experiment is the result of a manual but systematic selection process, aiming to find classes that are understandable in the short duration of the expriment (RQ4). The chosen classes are not small, but it may be that the readability optimization is more important for classes with more dependencies. Thus, to which extent our findings can be generalised to arbitrary programming and testing tasks remains an open question. We used EVOSUITE for experiments and to support the generation of our data set, and tests produced by EVOSUITE and other tools may lead to different results. However, the output of EVOSUITE is similar to that of other tools aiming at code coverage.

Conclusion: The human study to answer RQ4 involved 30 human subjects, which resulted in significance in only two out of the 10 test pairs. However, obtaining more responses per test pair by reducing the number of pairs was not possible, as this would have implied that students would have to answer several questions related to the same class, which would have led to undesired learning effects.

4.2 RQ1: Test vs. Code Readability

As a baseline for the success of our domain specific readability model, we used the code readability model by Buse and Weimer [7], as well as the extended version by Posnett et al. [34]. Both models are originally classification models, and we replicated them as regression models. We created two versions for each model, one trained with our dataset of 450 test cases with human annotations, and Buse and Weimer's original dataset of 100 code snippets with 1,200 human judgments. This allows us to distinguish between the effects of the choice of features and the training data.

Figure 4 shows the performance using seven different learners (Linear Regression, Multilayer Perceptron, SMOreg, M5Rules, M5P, Additive Regression, Bagging) on the *Buse & Weimer* code readability model, Posnett et al.'s *Simpler* model (which includes Halstead- and entropy-based features), and our *Test* readability model, in terms of the correlation using 10-fold cross validation. The Buse & Weimer and Simpler models both perform better on code snippets than on test snippets. In contrast, our Test model shows a poor performance on the code dataset while achieving the overall highest correlation on the test dataset with a median value 0.79, which

Table 2: Model prediction agreement with user choices

Each model is trained with two available datasets and tested with 200 pairs of test cases. Agreement shows the percentage of choices predicted by model agreeing with the overall user choice. Cohen's kappa [11] shows inter-rater agreement between the model prediction and user choices.

	Code dataset			Test dataset		
Model	Cohen's kappa	*p-value*	Agreement	Cohen's kappa	*p-value*	Agreement
Buse & Weimer	0.37	<0.01	0.685	0.33	<0.01	0.665
Simpler	0.57	<0.01	0.790	0.38	<0.01	0.695
Test Readability	0.31	<0.01	0.655	0.73	0	0.865

suggests that our choice of features is well adapted to the specific details influencing test readability. These results demonstrate the importance of our domain-specific model of software readability.

To compare these models with respect to their agreement with human judgement, we used the dataset of 200 pairs (Section 2.2) and measured the agreement between each model and the majority preference of the human annotators (i.e., percentage of matching choices). Table 2 shows the tests used to calculate the model–user agreement. Our test readability model trained with test case snippets achieves a high inter-rater agreement (kappa=0.73, *p-value*=0), with a correctness ratio of 0.865 over 200 test pairs. As the Table shows, our test readability model significantly outperform previous code readability models at this task.

RQ1: Our test readability model performs better on test snippet datasets, achieving a higher agreement with human annotators than previous work (kappa +28%).

4.3 RQ2: Improved Test Generation

To evaluate the success of the test optimization technique, we applied test generation to 30 classes with 10 repetitions each, with and without optimization, and compared the tests per branch in terms of their readability score. That is, each pair of tests is generated for the same coverage objective, but differs in readability score. We find that 56% of test cases on which optimization was applied had at least one alternative to choose from; for these tests, on average there were 5.1 alternatives. Table 3 summarizes the overall results: On all but three classes there is a significant increase of the readability score, by an average of 1.9% (2.5% when considering only tests that had alternatives). This increase may seem small, but recall that readability scores have low variance (cf. the narrow range of values between 3.0 and 4.0 in Figure 2), and the syntactic differences for improvement steps are small and incremental.

The largest improvement is observable for classes where the largest numbers of alternatives can be generated. As our algorithm (see Algorithm 1) is based on varying method and constructor calls, generation of alternatives works best when the dependency classes have many different constructors and parameters. Furthermore, we observe that more alternatives are generated for classes with more and simpler methods. For example, `math3.complex.Complex` has 196 branches, but these are spread over 48 methods, and this results in the overall largest readability improvement (+8.8%), with 92 alternatives per test on average. Similar examples are the class `org.joda.time.Months` and `beanutils.locale.converters.DateLocaleConverter`. The first one has five constructors plus many methods that also return `Months` instances, and the second has twelve different constructors. This leads to a large number of alternatives (39 for `Months` and 64 for `DateLocaleConverter`), and readability improvements of +3.2% and +4.9%, respectively.

On the other hand, class `cli.Option` has three constructors and methods that mainly take primitive values as parameters, and thus results in just 0.79 alternative tests on average (with a significant readability improvement of +0.4%). An extreme example is given by class `codec.language.Metaphone`, where the number of alternatives is close to 0.0 (see Table 3). `codec.language.Metaphone` is one of the largest classes, with 211 branches, but 184 of those branches are in the same method `public String metaphone(String txt){...}`, which receives and returns a String object. Therefore, there is no other alternative of creating a metaphone string without calling that method. For this specific class, a better way to optimize readability may be by directly optimizing the strings using a language model [1].

RQ2: Alternatives were generated for 56% of the unit tests, resulting in a readability improvement of +1.9% on average.

4.4 RQ3: Do Humans Prefer Readability Optimized Tests?

To evaluate whether humans prefer the readability optimized test cases to the default tests generated by EvoSuite, we applied test generation to 30 classes, with and without optimization. For each class we chose the three pairs of tests with the largest difference in readability score, and used a forced-choice survey to let human annotators select which test cases they think are more readable. That is, for each pair, both tests cover the same branch of the same class, but differ only in their readability score. Table 4 shows the details of the pairs used for this experiment, and shows the joint probability of agreement, which is 69% overall. On average, for all 30 classes there is a preference for the optimized tests. The average pair-wise agreement (fraction of pairs rated by both raters on which they agree) is 0.58; a one-sided Wilcoxon signed-rank shows that this is significantly better ($p < 0.001$) than a random choice (assuming agreement of 0.5 for random choices).

The highest agreement is observed for the first pair for class `net.n3.nanoxml.XMLElement` (87%). Here, the optimized test expects an exception and has only three statements. In contrast, the default test has four statements and six assertions, expects no exception, and uses random strings (e.g., `"7I%d7W5Y(Ta+"`).

In class `org.magee.math.Rational` there are two pairs for which the users prefer the default test case. For pair 2, the optimized test expects a `NullPointerException`, and we have generally observed that exceptional tests tend to get slightly higher readability scores. However, in this case the users disagree, possibly influenced by the rather meaningless comment in the catch block stating that there is "no message". For pair 3, the tests are very similar, and it is possible that domain knowledge influences the choice: The default test subtracts a number from itself, whereas the optimized test performs a syntactically similar, but mathematically slightly more complex calculation.

RQ3: Our experiment showed an agreement of 69% between human annotators and the readability optimization.

4.5 RQ4: Does Readability Optimization Improve Human Understanding of Tests?

Table 5 summarizes the results of the controlled experiment to answer RQ4. 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 tests. The average time spent on the non-optimized tests was 4.7 minutes, compared with 4 minutes for the optimized tests. Overall, this suggests that improved readability helps when making this software maintenance decision.

On the other hand, there are five classes where the ratio of correct responses increases, and five where the ratio decreases, suggesting that there are other factors influencing the difficulty of understanding a test that are not captured by our readability model. For example, the tests for class `cli.Option` have a substantial difference in readability (3.55 default, 4.01 optimized), but the default one con-

Table 3: Readability value for the 30 classes selected based on 10 runs per branch.

For each class we report its version, the number of branches, and the readability value using *default* configuration. For the *optimized* configuration we also report its readability value, the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *default* configuration, and average number of optimized alternatives generated. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold.

Class	Version	Branches	Readability		$\hat{A}_{12}$	Relative Improvement	Average Number of Alternatives
			Default	Optimized			
java2.util2.BitSet	-	288	3.8701	3.9091	**0.54**	+1.0%	2.03
net.n3.nanoxml.StdXMLReader	2.2.1	96	3.4257	3.4347	**0.52**	+0.2%	1.14
net.n3.nanoxml.XMLElement	2.2.1	165	3.6655	3.7376	**0.70**	+1.9%	9.72
net.xineo.xml.handler.ObjectHandlerAdapter	1.1.0	23	3.4762	3.5492	**0.62**	+2.0%	17.42
nu.xom.Attribute	1.2.10	201	3.7734	3.7882	**0.63**	+0.3%	9.79
org.apache.commons.beanutils.locale.converters.DateLocaleConverter	1.9.2	51	3.6238	3.8021	**0.78**	+4.9%	64.03
org.apache.commons.chain.impl.ChainBase	1.2	33	3.6404	3.6754	**0.56**	+0.9%	25.62
org.apache.commons.cli.CommandLine	1.2	48	3.6985	3.7176	**0.57**	+0.5%	0.68
org.apache.commons.cli.Option	1.2	97	3.7459	3.7639	**0.54**	+0.4%	0.79
org.apache.commons.cli.PosixParser	1.2	46	3.5414	3.5912	**0.64**	+1.4%	20.62
org.apache.commons.codec.language.Metaphone	1.1	211	3.7523	3.7539	0.50	+0.0%	0.0044
org.apache.commons.codec.language.Soundex	1.1	40	3.7959	3.8312	**0.65**	+0.9%	1.74
org.apache.commons.collections4.comparators.ComparatorChain	4	52	3.4294	3.5163	**0.68**	+2.5%	11.85
org.apache.commons.collections4.comparators.FixedOrderComparator	4	73	3.1959	3.2772	**0.62**	+2.5%	11.01
org.apache.commons.collections4.iterators.FilterIterator	4	21	3.7577	3.8228	**0.65**	+1.7%	3.12
org.apache.commons.collections4.iterators.FilterListIterator	4	51	3.5880	3.6604	**0.60**	+2.0%	3.92
org.apache.commons.collections.primitives.ArrayIntList	1	28	3.7554	3.7707	0.52	+0.4%	1.56
org.apache.commons.configuration.tree.MergeCombiner	1.1	29	3.2433	3.3241	**0.67**	+2.4%	10.66
org.apache.commons.digester3.plugins.PluginRules	3.2	47	3.7284	3.7878	**0.56**	+1.5%	8.49
org.apache.commons.digester3.RulesBase	3.2	39	3.5806	3.6379	**0.60**	+1.6%	14.79
org.apache.commons.lang3.CharRange	3.3.2	58	3.9201	3.9363	**0.62**	+0.4%	4.80
org.apache.commons.math3.complex.Complex	3.4.1	196	3.6501	3.9748	**0.83**	+8.8%	91.89
org.apache.commons.math3.fraction.Fraction	3.4.1	118	3.7660	3.8892	**0.74**	+3.2%	34.66
org.apache.commons.math3.genetics.ListPopulation	3.4.1	25	3.6245	3.6353	**0.53**	+0.2%	1.26
org.apache.commons.math3.stat.clustering.DBSCANClusterer	3.4.1	32	3.1619	3.1763	0.53	+0.4%	2.63
org.jdom2.Attribute	2.0.5	74	3.7439	3.8507	**0.71**	+2.8%	41.93
org.jdom2.DocType	2.0.5	21	3.8127	3.9344	**0.67**	+3.1%	9.62
org.joda.time.Months	2.7	69	3.8716	3.9982	**0.68**	+3.2%	39.38
org.joda.time.YearMonthDay	2.7	71	3.5456	3.6278	**0.68**	+2.3%	32.69
org.magee.math.Rational	2005-11-19	36	3.7897	3.9391	**0.85**	+3.9%	9.84
Average			3.6953	3.7284	0.63	+1.9%	5.09

Table 4: Readability value for the 30 classes selected based on the top three pairs that maximise the difference between *default* and *optimized* configurations.

For each class and pair we report the readability value of each configuration and the percentage of users that agree *optimized* test cases are better in terms of readability.

Class	Pair 1			Pair 2			Pair 3			Average Agree
	Readability			Readability			Readability			
	Default	Optimized	Agree	Default	Optimized	Agree	Default	Optimized	Agree	
java2.util2.BitSet	3.78	3.92	71.25%	3.73	3.88	62.79%	3.83	3.90	60.71%	64.92%
net.n3.nanoxml.StdXMLReader	3.15	3.87	70.00%	3.17	3.69	67.02%	3.34	3.81	69.44%	68.82%
net.n3.nanoxml.XMLElement	3.34	3.80	87.18%	3.33	3.73	73.00%	3.40	3.78	63.00%	74.39%
net.xineo.xml.handler.ObjectHandlerAdapter	3.48	3.78	71.95%	3.39	3.67	70.45%	3.22	3.56	78.41%	73.60%
nu.xom.Attribute	3.33	3.81	69.57%	3.24	3.83	75.00%	3.24	3.75	61.96%	68.84%
org.apache.commons.beanutils.locale.converters.DateLocaleConverter	3.22	3.79	81.25%	3.34	3.85	65.69%	3.38	3.74	68.09%	71.67%
org.apache.commons.chain.impl.ChainBase	2.99	3.75	74.39%	3.27	3.86	73.81%	3.31	3.86	68.57%	72.26%
org.apache.commons.cli.CommandLine	3.34	3.75	75.00%	3.49	3.82	82.14%	3.41	3.76	69.74%	75.63%
org.apache.commons.cli.Option	3.48	4.01	70.00%	3.53	3.87	68.18%	3.59	3.92	57.14%	65.11%
org.apache.commons.cli.PosixParser	3.39	3.68	67.39%	3.35	3.72	73.81%	3.49	3.73	63.41%	68.21%
org.apache.commons.codec.language.Metaphone	3.59	3.88	77.55%	3.61	3.87	72.34%	3.67	3.82	64.58%	71.49%
org.apache.commons.codec.language.Soundex	3.76	3.92	51.96%	3.63	3.92	68.89%	3.60	3.89	75.00%	65.28%
org.apache.commons.collections4.comparators.ComparatorChain	3.27	3.92	72.22%	3.36	3.79	69.44%	3.51	3.92	66.67%	69.44%
org.apache.commons.collections4.comparators.FixedOrderComparator	2.70	3.34	69.44%	2.80	3.44	72.34%	2.63	3.12	74.42%	72.07%
org.apache.commons.collections4.iterators.FilterIterator	3.56	3.99	71.62%	3.33	3.83	67.35%	3.64	3.95	65.85%	68.27%
org.apache.commons.collections4.iterators.FilterListIterator	3.36	3.84	80.21%	3.08	3.69	84.21%	3.12	3.69	81.71%	82.04%
org.apache.commons.collections.primitives.ArrayIntList	3.35	3.72	70.41%	3.79	3.94	53.06%	3.58	3.79	66.00%	63.16%
org.apache.commons.configuration.tree.MergeCombiner	3.32	3.68	73.17%	3.28	3.55	72.50%	3.32	3.59	55.41%	67.03%
org.apache.commons.digester3.plugins.PluginRules	2.49	3.46	67.44%	3.09	3.73	57.69%	3.08	3.64	61.54%	62.22%
org.apache.commons.digester3.RulesBase	3.23	3.88	78.57%	3.13	3.71	76.60%	3.17	3.73	67.59%	74.25%
org.apache.commons.lang3.CharRange	3.82	4.04	77.27%	3.80	3.97	63.27%	3.91	4.05	76.19%	72.24%
org.apache.commons.math3.complex.Complex	3.39	3.97	78.05%	3.38	3.94	60.47%	3.54	4.01	68.18%	68.90%
org.apache.commons.math3.fraction.Fraction	2.98	3.60	63.04%	3.65	3.83	66.67%	3.53	3.88	75.53%	68.41%
org.apache.commons.math3.genetics.ListPopulation	3.43	3.83	75.00%	3.44	3.75	67.05%	3.48	3.83	72.34%	71.46%
org.apache.commons.math3.stat.clustering.DBSCANClusterer	3.28	3.65	65.91%	3.49	3.65	67.44%	3.13	3.36	76.09%	69.81%
org.jdom2.Attribute	3.57	3.86	73.40%	3.78	3.86	73.33%	3.66	3.86	63.00%	69.91%
org.jdom2.DocType	3.49	3.75	73.86%	3.71	3.85	57.45%	3.57	3.85	67.86%	66.39%
org.joda.time.Months	3.35	4.05	64.00%	3.44	4.02	61.70%	3.46	3.99	62.16%	62.62%
org.joda.time.YearMonthDay	3.25	3.81	78.95%	3.32	3.81	71.11%	3.15	3.70	75.00%	75.02%
org.magee.math.Rational	3.86	3.87	61.54%	3.68	3.95	45.00%	3.67	3.83	50.00%	52.18%
Average										69.19%

Table 5: Human understanding results of tests for the 10 classes randomly selected.

For each class selected we report the branch covered, the test result of each individual (pass/fail), number of asserts or fail keywords, average time to answer, and percentage of correct responses. Effect sizes of statistically significant differences ($p < 0.05$) are shown in bold.

		Oracle		Readability		Assert/Fail		Time (min)			Correct Answers	
Class	Branch	Def.	Optim.	Def.	Optim.	Def.	Optim.	Def.	Optim.	$\hat{A}_{12}$	Def.	Optim.
net.n3.nanoxml.StdXMLReader	67	pass	fail	3.40	3.79	0 / 1	0 / 1	4.35	3.76	0.57	60.00%	61.11%
nu.xom.Attribute	91	pass	pass	3.33	3.81	0 / 1	0 / 1	4.69	5.75	0.43	60.00%	38.46%
org.apache.commons.chain.impl.ChainBase	5	pass	pass	2.99	3.75	1 / 0	1 / 0	4.04	3.80	0.61	76.47%	54.55%
org.apache.commons.cli.Option	67	fail	pass	3.55	4.01	5 / 0	0 / 1	2.21	2.69	0.34	100.00%	75.00%
org.apache.commons.collections4.comparators.FixedOrderComparator	22	pass	pass	2.63	3.30	3 / 0	2 / 0	4.92	3.31	0.70	64.29%	23.08%
org.apache.commons.collections4.iterators.FilterListIterator	4	fail	fail	3.10	3.73	2 / 0	1 / 0	4.75	3.00	**0.73**	71.43%	85.71%
org.apache.commons.digester3.plugins.PluginRules	25	pass	pass	2.49	3.46	1 / 0	1 / 0	6.43	6.04	0.51	72.73%	62.50%
org.apache.commons.digester3.RulesBase	3	pass	pass	2.80	3.76	1 / 0	1 / 0	4.77	3.24	0.68	90.91%	100.00%
org.apache.commons.lang3.CharRange	15	pass	pass	3.82	4.04	2 / 0	1 / 0	4.35	1.93	**0.92**	50.00%	100.00%
org.joda.time.YearMonthDay	66	pass	pass	3.25	3.81	3 / 0	0 / 1	6.06	6.49	0.48	30.77%	61.54%
Average				3.14	3.75			4.66	4.00	0.60	67.66%	66.19%

```
BuddhistChronology buddhistChronology0 =
    BuddhistChronology.getInstance();
YearMonthDay yearMonthDay0 = new YearMonthDay((Chronology
    ) buddhistChronology0);
YearMonthDay.Property yearMonthDay_Property0 = new
    YearMonthDay.Property(yearMonthDay0, 0);
YearMonthDay yearMonthDay1 = yearMonthDay_Property0.
    addWrapFieldToCopy(0);
assertEquals("2557-02-14", yearMonthDay1.toString());
assertEquals("2557-02-14", yearMonthDay0.toString());
assertEquals(-292268511, yearMonthDay_Property0.
    getMinimumValueOverall());
```

```
YearMonthDay yearMonthDay0 = null;
try {
  yearMonthDay0 = new YearMonthDay(0, 0, 0);
  fail("Expecting exception: IllegalFieldValueException");

} catch(IllegalFieldValueException e) {
  //
  // Value 0 for monthOfYear must not be smaller than 1
  //
}
```

Figure 5: Default and optimized test case for class `org.joda.time.YearMonthDay`.

tains regular assertions, whereas the optimized one expects an exception to be thrown. While all responses for the non-optimized test were correct, only 75% of the responses for the optimized test were correct, and the average time for responses increased from 2.21min to 2.69min. The likely explanation for this is that, even though the readability model suggests that exceptions improve readability, it may be more difficult to understand exceptional control flow.

This conjecture is supported by the tests for class `org.joda.time.YearMonthDay` (see Figure 5), where again the optimized test leads to an expected exception. Here, the time to response increases from 6.06min to 6.49min on average. However, in contrast to `cli.Option`, the percentage of correct responses increases by 31%. Possibly, this improvement is influenced by the error message of the expected exception, which is included as a comment: The optimized test calls the constructor of class `YearMonthDay(day, month, year)` with value 0 and the exception message was *"Value 0 for monthOfYear must not be smaller than 1"*.

The third class with an increase in time, `nu.xom.Attribute`, tests exceptional behavior in both versions. Here, the percentage of correct responses is only 39%, compared to 60% in the default version. This reduction may again be related to the specific error message, as the default test complains about the use of an *"Illegal initial scheme character"* in a URI parameter, which apparently is easier to understand than the *"Missing scheme in absolute URI reference"* message of the optimized test.

Both default and optimized test expect a null pointer exception for class `net.n3.nanoxml.StdXMLReader`. Although the

```
CatalogFactoryBase catalogFactoryBase0 = (
    CatalogFactoryBase)CatalogFactory.getInstance();
DispatchLookupCommand dispatchLookupCommand0 = new
    DispatchLookupCommand((CatalogFactory)
    catalogFactoryBase0);
HashMap<Object, CopyCommand> hashMap0 = new HashMap<
    Object, CopyCommand>();
ContextBase contextBase0 = new ContextBase((Map) hashMap0
    );
Set set0 = contextBase0.keySet();
ChainBase chainBase0 = new ChainBase((Collection) set0);
RemoveCommand removeCommand0 = new RemoveCommand();
Command[] commandArray0 = new Command[4];
commandArray0[0] = (Command) removeCommand0;
commandArray0[1] = (Command) dispatchLookupCommand0;
commandArray0[2] = (Command) dispatchLookupCommand0;
commandArray0[3] = (Command) chainBase0;
ChainBase chainBase1 = new ChainBase(commandArray0);
assertFalse(chainBase1.equals((Object)chainBase0));
```

```
ChainBase chainBase0 = new ChainBase();
Command[] commandArray0 = chainBase0.getCommands();
ChainBase chainBase1 = new ChainBase(commandArray0);
assertNotSame(chainBase1, chainBase0);
```

Figure 6: Default and optimized test case for class `chain.impl.ChainBase`.

rate of correct responses is comparable, here the time spent on the optimized test is lower on average (4.35min default vs, 3.76min optimized). This is likely related to the difference in size; the default test has seven statements, the optimized one only two.

For `comparators.FixedOrderComparator`, the reduction of correct responses may be a result of uncertainty arising from how null-values are handled in the maps underlying the class; the default test case does not use null, the optimized one does. The same may also hold for `digester3.plugins.PluginRules`, where the optimized test case uses several null values.

These results are reflected by the participants' opinions. In free response questions at the end of our survey, almost every user stated that a readable test case must be (1) minimal (no more than 5 lines if possible); (2) have short lines; (3) not dependent on too many classes, as for example indicated by the comment that "due to deep inheritance and lots of underlying methods to construction, it was somewhat hard to understand some classes under test."

For `chain.impl.ChainBase` the slight reduction in correct responses is surprising, as the optimized test case has a trivially true assertion (`assertNotSame` with two objects resulting from two different constructor invocations; cf. Figure 6). However, this assertion form is less common and might have been interpreted by several participants as the more common `assertNotEquals`, which may contribute to the increase in wrong responses.

Finally, for `FilterListIterator` and `CharRange` there are large improvements in the response time and in the correctness

```
CharRange charRange0 = CharRange.isNot('#');
Character character0 = Character.valueOf('#');
CharRange charRange1 = CharRange.isNotIn('\"', (char)
    character0);
char char0 = charRange1.getStart();
assertEquals('\"', char0);

boolean boolean0 = charRange0.contains('\"');
assertTrue(boolean0);
```

```
CharRange charRange0 = CharRange.is(' ');
boolean boolean0 = charRange0.contains(' ');
assertTrue(boolean0);
```

Figure 7: Default and optimized test case for `lang3.CharRange`.

of responses. For `FilterListIterator` the default test case uses a confusing chain of calls to construct the instance of the class under test; for the `CharRange` default test case this also holds, but here maybe a different factor also plays a role. As we can see in Figure 7 the default test case starts by constructing a negated `CharRange` over a single character *"#"*, and later checks whether a different character is contained in the `CharRange`, whereas the optimized test case uses the same character twice, when setting up the `CharRange` and when querying `contains`.

RQ4: In our experiments, the optimized tests reduced the response time by 14%, but did not directly influence participant accuracy.

5. RELATED WORK

Readability Metrics. Buse and Weimer [7] introduced a metric for code readability based on human judgements. They collected human annotation data for code snippets and trained a classifier based on those scores. Our work is based on the same readability metric concept, but produces a domain-specific model for unit tests, using dedicated test features and data, resulting in an overall better performance and prediction power. Posnett et al. [34] used the same dataset to learn a simpler model of code readability, using fewer features based on size, Halstead metrics, and entropy. We replicated this model as a regression model and applied it to test data, and saw that the performance of our test-specific model is still better.

Readability Optimization of Tests. The problem of understanding tests is well known. The most common scenario where understanding a test is necessary is if a test fails. To support developers with debugging failing tests, Leitner et al. [30] and Lei and Andrews [29] minimised failing (randomly generated) tests using delta-debugging in order to simplify debugging the failure cause. Zhang et al. [42] presented an approach to synthesise natural language documentation to explain the failure.

Often, a failing test is not a problem in the program that needs fixing, but a maintenance problem in the test, requiring a fix in the test code. Robinson et al. [35] discuss a range of optimizations to apply during test generation in order to reduce the number of false failures produced by tests and also improve their readability. Daniel et al. [14] and Mirzaaghaei et al. [33] provide automated techniques to fix failing tests, and Hao et al. [25] use machine learning to predict whether a test failure is due to a code or test problem.

To avoid creating maintainability problems when writing tests, there are established standard patterns for test design [32], and there are certain test *smells* [17] that help to detect problematic tests. However, the most discussed scenario related to understandability of tests is when the tests are generated automatically. Automated test generation typically uses systematic approaches, for example to derive tests satisfying coverage criteria. In order to find faults, automatically generated tests require a test oracle [5]; that is, some mechanism that decides whether the test execution revealed a failure or matched the expected behavior.

Common approaches to address the test oracle problem include an optimization of the number of tests generated in the first place [26]. However, there have also been attempts to improve the readability of tests systematically: Fraser and Zeller [21] learn models of common object usage from existing code and tests, such that newly generated tests match the developer expectations. Afshan et al. [1] apply a natural language model to optimize textual input values, which are often simply random characters, to more English-like text. We included this model as one of our features, but found low predictive power for our dataset; possibly including more string-related tests would change this. Zhang [41] describes a transformation that replaces variables with the aim to reduce the number of statements, which works best on un-minimized tests. In contrast, our optimization algorithm works with already minimized tests and integrates test generation mechanisms to replace calls and satisfy new dependencies. Fraser and Zeller [22] reduce the number of assertions in a test using mutation analysis. Xuan and Monperrus [40] split tests into one per assertion; while the aim of this approach is to improve fault localization, this likely also has an influence on readability.

6. CONCLUSIONS

A unit test may be written once, but it is read and interpreted by developers many times. If a test is not readable, then it may be more difficult to understand it. This problem is exacerbated for unit tests generated automatically by tools, which in principle are intended to support developers in generating high coverage test suites, but in practice tend to generate tests that do not look as nice as manually-written ones. To address this problem, we have built a predictive model of test readability based on data of how humans rate the readability of unit tests. We have applied this model in an automated unit test generation tool, and validated that users prefer our readability optimized tests to non-optimized tests.

Our technique to increase the readability of unit tests is still quite limited in the scope of its changes to test appearance. For example, variable names are chosen according to a fixed strategy (i.e., class name in camel case, with lower caps first letter, and numeric ID attached at the end). Our feature analysis suggests that identifiers have a very strong 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 importance of identifier names in source code [3, 6, 9, 19, 24, 28, 38], and suggests that future work will need to address this problem for unit test generation, leveraging existing work on improving identifiers (e.g., [2, 9, 10, 16]).

Our experiment about the effects of readability on test understanding has also demonstrated the boundaries between readability and understandability: Not all tests that look nice are also easy to understand. The inclusion of *semantic* features such as code coverage, may lead to an improvement of the readability model. It is even conceivable to use our human maintenance question data to learn a model of understandability, rather than readability.

In summary, we proposed a domain-specific model of test readability and an algorithm for producing more readable tests. We found our model to outperform previous work in term of agreement with humans on test case readability (0.87 vs. 0.79); we found our approach to generate alternate optimized tests that were 2% more readable on average; we found humans to prefer our optimized tests 69% of the time; and we found that humans can answer questions about our tests 14% faster with no change in accuracy.

7. ACKNOWLEDGMENTS

This work was supported by the EPSRC project "EXOGEN" (EP / K030353 / 1) and National Science Foundation grants CCF 1116289, CCF 0954024 and CCF 0905373.

8. REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 352–361, 2013.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning Natural Coding Conventions. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 281–293, 2014.

[3] N. Anquetil and T. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 4–, 1998.

[4] A. Arcuri and L. Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)*, 24(3):219–250, 2014.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering (TSE)*, PP(99), 2014.

[6] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[7] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.

[8] R. P. L. Buse and T. Zimmermann. Information Needs for Software Development Analytics. In *International Conference on Software Engineering (ICSE)*, pages 987–996, 2012.

[9] B. Caprile and P. Tonella. Nomen Est Omen: Analyzing the Language of Function Identifiers. In *Working Conference on Reverse Engineering (WCRE)*, pages 112–, 1999.

[10] B. Caprile and P. Tonella. Restructuring Program Identifier Names. In *International Conference on Software Maintenance (ICSM)*, pages 97–, 2000.

[11] J. Carletta. Assessing Agreement on Classification Tasks: The Kappa Statistic. *Computational Linguistics*, 22(2):249–254, 1996.

[12] S. Cateni, M. Vannucci, M. Vannocci, and V. Colla. *Multivariate Analysis in Management, Engineering and the Sciences*. InTech, 2013-01-09.

[13] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. An Empirical Study About the Effectiveness of Debugging when Random Test Cases Are Used. In *International Conference on Software Engineering (ICSE)*, pages 452–462, 2012.

[14] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting Repairs for Broken Unit Tests. In *International Conference on Automated Software Engineering (ASE)*, pages 433–444, 2009.

[15] M. Dash, K. Choi, P. Scheuermann, and H. Liu. Feature Selection for Clustering - A Filter Solution. In *International Conference on Data Mining (ICDM)*, pages 115–122, 2002.

[16] F. Deißenböck and M. Pizka. Concise and Consistent Naming. *Software Quality Control*, 14(3):261–282, 2006.

[17] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok. Refactoring Test Code. Technical report, 2001.

[18] R. Duangsoithong and T. Windeatt. Relevance and Redundancy Analysis for Ensemble Classifiers. In *Machine Learning and Data Mining in Pattern Recognition*, volume 5632, pages 206–220, 2009.

[19] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An Exploratory Study of Identifier Renamings. In *Working Conference on Mining Software Repositories (MSR)*, pages 33–42, 2011.

[20] G. Fraser and A. Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *European Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[21] G. Fraser and A. Zeller. Exploiting Common Object Usage in Test Case Generation. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–89, 2011.

[22] G. Fraser and A. Zeller. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, March 2012.

[23] Z. P. Fry and W. Weimer. A Human Study of Fault Localization Accuracy. In *International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[24] L. Guerrouj. Normalizing Source Code Vocabulary to Support Program Comprehension and Software Quality. In *International Conference on Software Engineering (ICSE)*, pages 1385–1388, 2013.

[25] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang. Is This a Bug or an Obsolete Test? In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–628. 2013.

[26] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem. In *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 182–191, 2010.

[27] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing User Studies with Mechanical Turk. In *Conference on Human Factors in Computing Systems (CHI)*, pages 453–456, 2008.

[28] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. In *International Conference on Program Comprehension (ICPC)*, pages 3–12, 2006.

[29] Y. Lei and J. H. Andrews. Minimization of Randomized Unit Test Cases. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 267–276, 2005.

[30] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient Unit Test Case Minimization. In *International Conference on Automated Software Engineering (ASE)*, pages 417–420, 2007.

[31] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[32] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[33] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Supporting Test Suite Evolution through Test Case Adaptation. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240, 2012.

[34] D. Posnett, A. Hindle, and P. Devanbu. A Simpler Model of Software Readability. In *Working Conference on Mining Software Repositories (MSR)*, pages 73–82, 2011.

[35] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling Up Automated Test Generation: Automatically

Generating Maintainable Regression Unit Tests for Programs. In *International Conference on Automated Software Engineering (ASE)*, pages 23–32, 2011.

[36] M. Robnik-Šikonja and I. Kononenko. Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning*, 53(1-2):23–69, 2003.

[37] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng. Cheap and Fast—but is It Good?: Evaluating Non-expert Annotations for Natural Language Tasks. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 254–263, 2008.

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

[39] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[40] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM.

[41] S. Zhang. Practical Semantic Test Simplification. In *International Conference on Software Engineering (ICSE)*, pages 1173–1176, 2013.

[42] S. Zhang, C. Zhang, and M. D. Ernst. Automated Documentation Inference to Explain Failed Tests. In *International Conference on Automated Software Engineering (ASE)*, pages 63–72, 2011.