Learning a Metric for Code Readability

Raymond P.L. Buse, Westley Weimer

Abstract—In this paper, we explore the concept of code readability and investigate its relation to software quality. With data collected from 120 human annotators, we derive associations between a simple set of local code features and human notions of readability. Using those features, we construct an automated readability measure and show that it can be 80% effective, and better than a human on average, at predicting readability judgments. Furthermore, we show that this metric correlates strongly with three measures of software quality: code changes, automated defect reports, and defect log messages. We measure these correlations on over 2.2 million lines of code, as well as longitudinally, over many releases of selected projects. Finally, we discuss the implications of this study on programming language design and engineering practice. For example, our data suggests that comments, in of themselves, are less important than simple blank lines to local judgments of readability.

Index Terms—software readability, program understanding, machine learning, software maintenance, code metrics, FindBugs

1 Introduction

We define *readability* as a human judgment of how easy a text is to understand. The readability of a program is related to its maintainability, and is thus a key factor in overall software quality. Typically, maintenance will consume over 70% of the total lifecycle cost of a software product [4]. Aggarwal claims that source code readability and documentation readability are both critical to the maintainability of a project [1]. Other researchers have noted that the act of reading code is the most time-consuming component of all maintenance activities [8], [33], [35]. Readability is so significant, in fact, that Elshoff and Marcotty, after recognizing that many commercial programs were much more difficult to read than necessary, proposed adding a development phase in which the program is made more readable [10]. Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability [22] to ensure maintainability, portability, and reusability of the code. Haneef proposed adding a dedicated readability and documentation group to the development team, observing that, "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [16].

We hypothesize that programmers have some intuitive notion of this concept, and that program features such as indentation (e.g., as in Python [40]), choice of identifier names [34], and comments are likely to play a part. Dijkstra, for example, claimed that the readability of a program depends largely upon the simplicity of its sequencing control (e.g., he conjectured that `goto` unnecessarily complicates program understanding), and employed that notion to help motivate his top-down approach to system design [9].

We present a descriptive model of software readability based on simple features that can be extracted automatically from programs. This model of software readability correlates strongly with human annotators and also with external (widely available) notions of software quality, such as defect detectors and software changes.

To understand why an empirical and objective model of *software* readability is useful, consider the use of readability metrics in natural languages. The Flesch-Kincaid Grade Level [11], the Gunning-Fog Index [15], the SMOG Index [28], and the Automated Readability Index [21] are just a few examples of readability metrics for ordinary text. These metrics are all based on simple factors such as average syllables per word and average sentence length. Despite this simplicity, they have each been shown to be quite useful in practice. Flesch-Kincaid, which has been in use for over 50 years, has not only been integrated into popular text editors including Microsoft Word, but has also become a United States governmental standard. Agencies, including the Department of Defense, require many documents and forms, internal and external, to meet have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). Defense contractors also are often required to use it when they write technical manuals.

These metrics can help organizations gain some confidence that their documents meet goals for readability very cheaply, and have become ubiquitous for that reason. We believe that similar metrics, targeted specifically at source code and backed with empirical evidence for effectiveness, can serve an analogous purpose in the software domain. Readability metrics for the niche areas such as computer generated math [26], treemap layout [3], and hypertext [17] have been found useful. We describe the first general readability metric for source code.

• *Buse and Weimer are with the Department of Computer Science at The Universiy of Virginia, Charlottesville, VA 22904.*
E-mail: {buse, weimer}@cs.virginia.edu
This research was supported in part by, but may not reflect the positions of, National Science Foundation Grants CNS 0716478 and CNS 0905373, Air Force Office of Scientific Research grant FA9550-07-1-0532, and Microsoft Research gifts.

It is important to note that readability is not the same as complexity, for which some existing metrics have been empirically shown useful [41]. Brooks claims that complexity is an "essential" property of software; it arises from system requirements, and cannot be abstracted away [12]. In the Brooks model, readability is "accidental" because it is not determined by the problem statement. In principle, software engineers can only control accidental difficulties: implying that readability can be addressed more easily than intrinsic complexity.

While software complexity metrics typically take into account the size of classes and methods, and the extent of their interactions, the readability of code is based primarily on local, line-by-line factors. Our notion of readability arises directly from the judgments of actual human annotators who do not have context for the code they are judging. Complexity factors, on the other hand, may have little relation to what makes code understandable to humans. Previous work [31] has shown that attempting to correlate artificial code complexity metrics directly to defects is difficult, but not impossible. Although both involve local factors such as indentation, readability is also distinct from coding standards (e.g., [2], [6], [38]), conventions primarily intended to facilitate collaboration by maintaining uniformity between code written by different developers.

In this study, we have chosen to target readability directly both because it is a concept that is independently valuable, and also because developers have great control over it. We show in Section 5 that there is indeed a significant correlation between readability and quality. The main contributions of this paper are:

- A technique for the construction of an automatic software readability metric based on local code features.
- A survey of 120 human annotators on 100 code snippets that forms the basis of the metric presented and evaluated in this paper. We are unaware of any published software readability study of comparable size (12,000 human judgments). We directly evaluate the performance of our model on this data set.
- A set of experiments which reveal significant correlations between our metric for readability and external notions of software quality including defect density.
- A discussion of the features involved in our metric and their relation to software engineering and programming language design.

Some of these main points were previously presented [5]. This article also includes:

- A broader base of benchmark programs for empirical experiments linking our readability metric with notions of software quality. In particular, we evaluate on over two million lines of code compared to over one million in previous work.
- A reporting of the primary results of our analysis in terms of five correlation statistics as compared to one in the previous presentation.
- An additional experiment correlating our readability metric with a more natural notion of software quality and defect density: explicit human mentions of bug repair. Using software version control repository information we coarsely separate changes made to address bugs from other changes. We find that low readability correlates more strongly with this direct notion of defect density than it does with the previous approach of using potential bugs reported by static analysis tools.
- An additional experiment which compares readability to cyclomatic complexity. This experiment serves to validate our claim that our notion of readability is largely independent from traditional measures of code complexity.
- An additional longitudinal experiment showing how changes in readability can correlate with changes in defect density as a program evolves. For ten versions of each of five programs we find that projects with unchanging readability have similarly unchanging defect densities, while a project that experienced a sharp drop in readability was subject to a corresponding rise in defect density.

There are a number of possible uses for an automated readability metric. It may help developers to write more readable software by quickly identifying code that scores poorly. It can assist project managers in monitoring and maintaining readability. It can serve as a requirement for acceptance. It can even assist inspections by helping to target effort at parts of a program that may need improvement. Finally, it can be used by other static analyses to rank warnings or otherwise focus developer attention on sections of the code that are less readable and thus more likely to contain bugs. The readability metric presented in this paper has already been used successfully to aid a static specification mining tool [24] — in that setting, knowing whether a code path was readable or not had over twice the predictive power (as measured by ANOVA F-score) as knowing whether a code path was feasible or not.

The structure of this paper is as follows. In Section 2 we present a study of readability involving 120 human annotators. We present the results of that study in Section 3, and in Section 4 we determine a small set of features that is sufficient to capture the notion of readability for a majority of annotators. In Section 5 we discuss the correlation between our readability metric and external notions of software quality. We discuss some of the implications of this work on programming language design in Section 6, discuss potential threats to validity in Section 7, discuss possibilities for extension in Section 8, and conclude in Section 9.

2 Study Methodology

A consensus exists that readability is an essential determining characteristic of code quality [1], [4], [8], [9],

[10], [16], [31], [32], [33], [34], [35], [41], but not about which factors contribute to human notions of software readability the most. A previous study by Tenny looked at readability by testing comprehension of several versions of a program [39]. However, such an experiment is not sufficiently fine-grained to extract precise features. In that study, the code samples were large, and thus the perceived readability arose from a complex interaction of many features, potentially including the purpose of the code. In contrast, we choose to measure the software readability of small (7.7 lines on average) selections of code. Using many short code selections increases our ability to tease apart which features are most predictive of readability. We now describe an experiment designed to extract a large number of readability judgments over short code samples from a group of human annotators.

Formally, we can characterize software readability as a mapping from a code sample to a finite score domain. In this experiment, we presented human annotators with a sequence of short code selections, called *snippets*, through a web interface. The annotators were asked to individually score each snippet based on their personal estimation of readability. We now discuss three important parameters: snippet selection policy (Section 2.1), snippet scoring (Section 2.2), and participation (Section 2.3).

2.1 Snippet Selection Policy

We claim that the readability of code is very different from that of natural languages. Code is highly structured and consists of elements serving different purposes, including design, documentation, and logic. These issues make the task of snippet selection an important concern. We have designed an automated policy-based tool that extracts snippets from Java programs.

First, snippets should be relatively short to aid feature discrimination. However, if snippets are too short, then they may obscure important readability considerations. Second, snippets should be logically coherent to allow annotators the context to appreciate their readability. We claim that they should not span multiple method bodies and that they should include adjacent comments that document the code in the snippet. Finally, we want to avoid generating snippets that are "trivial." For example, we are uninterested in evaluating the readability of a snippet consisting entirely of boilerplate import statements or entirely of comments.

Consequently, an important tradeoff exists such that snippets must be as short as possible to adequately support analysis by humans, yet must be long enough to allow humans to make significant judgments on them. Note that it is *not* our intention to "simulate" the reading process, where context may be important to understanding. Quite the contrary: we intend to eliminate context and complexity to a large extent and instead focus on the "low-level" details of readability. We do not imply that context is unimportant; we mean only to show that there exists a set of local features that, by themselves, have a strong impact on readability and, by extension, software quality.

With these considerations in mind, we restrict snippets for Java programs as follows. A snippet consists of precisely three consecutive simple statements [14], the most basic unit of a Java program. Simple statements include field declarations, assignments, function calls, breaks, continues, throws and returns. We find by experience that snippets with fewer such instructions are sometimes too short for a meaningful evaluation of readability, but that three statements are generally both adequate to cover a large set of local features and sufficient for a fine-grained feature-based analysis.

A snippet *does* include preceding or in-between lines that are not simple statements, such as comments, function headers, blank lines, or headers of compound statements like `if-else`, `try-catch`, `while`, `switch`, and `for`. Furthermore, we do not allow snippets to cross scope boundaries. That is, a snippet neither spans multiple methods nor starts inside a compound statement and then extends outside it (however, we do permit snippets to start outside of a compound statement but end before the statement is complete). We find that with this set of policies, over 90% of statements in all of the programs we considered (see Section 5) are candidates for incorporation in some snippet. The few non-candidate lines are usually found in functions that have fewer than three statements. This snippet definition is specific to Java but extends to similar languages like C and C++. We find the size distribution (in number of characters) for the 100 snippets generated for this study to be approximately normal, but with a positive skew (mean of 278.94, median of 260, minimum of 92 and maximum of 577).

The snippets were generated from five open source projects (see Figure 10). They were chosen to include varying levels of maturity and multiple application domains to keep the model generic and widely-applicable. We discuss the possibility of domain-specific models in Section 8.

2.2 Readability Scoring

Prior to their participation, our volunteer human annotators were told that they would be asked to rate Java code on its readability, and that their participation would assist in a study of that aspect of software quality. Responses were collected using a web-based annotation tool (shown in Figure 1) that users were permitted to access at their leisure. Participants were presented with a sequence of snippets and buttons labeled 1–5 [25]. Each participant was shown the same set of one hundred snippets in the same order. Participants were graphically reminded that they should select a number near five for "more readable" snippets and a number near one for "less readable" snippets, with a score of three indicating neutrality. Additionally, there was an option to skip the current snippet; however, it was used very infrequently

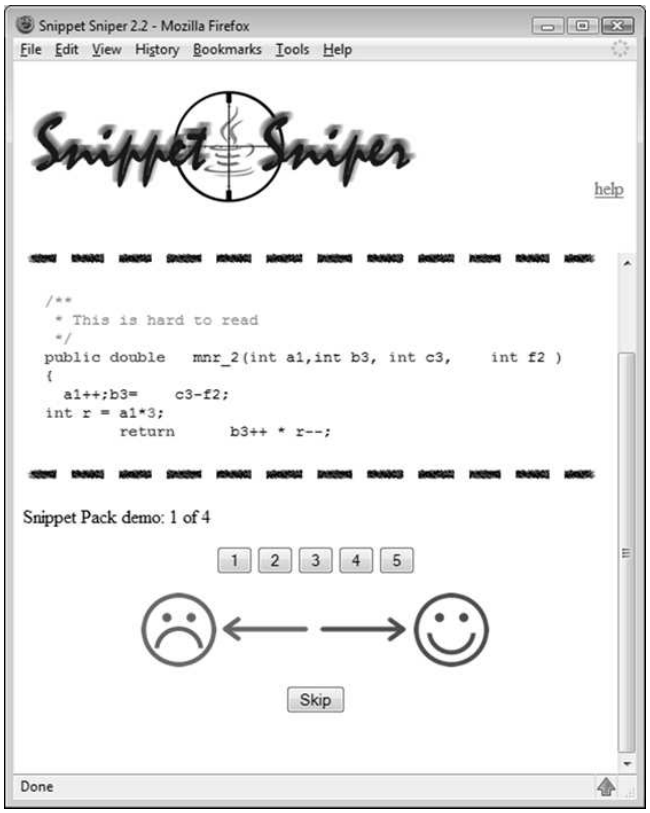

Fig. 1. Web-based tool for annotating the readability of code snippets used for this study.

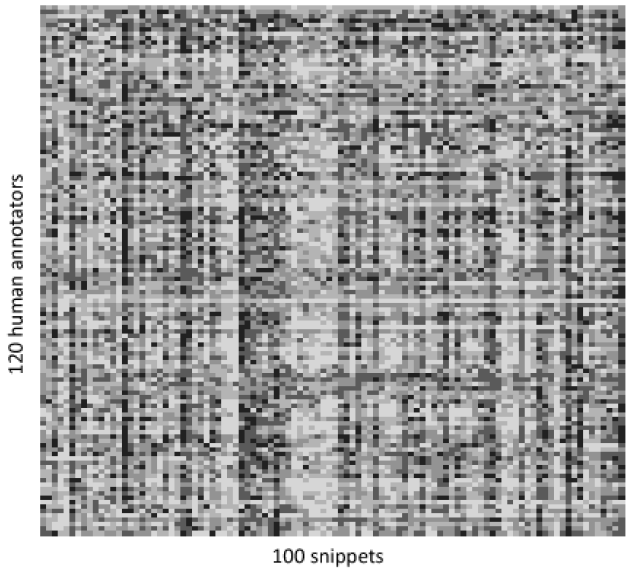

Fig. 2. The complete data set obtained for this study. Each box corresponds to a judgment made by a human annotator. Darker colors correspond to lower readability scores (e.g., 1 and 2) the lighter ones correspond to higher scores. The vertical bands, which occur much more frequently here than in a figure with random coloration, indicate snippets that were judged similarly by many annotators. Our metric for readability is derived from these 12,000 judgments.

(15 times in 12,000). Snippets were not modified from the source, but they were syntax highlighted to better simulate the way code is typically viewed.[1] Finally, clicking on a "help" link reminded users that they should score the snippets "based on [their] estimation of readability" and that "readability is [their] judgment about how easy a block of code is to understand." Readability was intentionally left formally undefined in order to capture the unguided and intuitive notions of participants.

2.3 Study Participation

The study was advertised at several computer science courses at *The University of Virginia*. As such, participants had varying experience reading and writing code: 17 were taking first-year courses, 63 were taking second year courses, 30 were taking third or fourth year courses, and 10 were graduate students. In total, 120 students participated. The study ran from Oct 23 to Nov 2, 2008. Participants were told that all respondents would receive $5 USD, but that the fifty people who *started* (not finished) the survey the earliest would receive $10 USD. The use of start time instead of finish time encouraged participation

1. Both syntax highlighting and the automatic formatting of certain IDEs (e.g., Eclipse) may change the perceived readability of code. In our opinion, the vast majority of editors feature syntax highlighting, but while automatic formatting is often available and is prevalent in some editors, it is not as universally used. We thus present snippets with syntax highlighting but without changing the formatting.

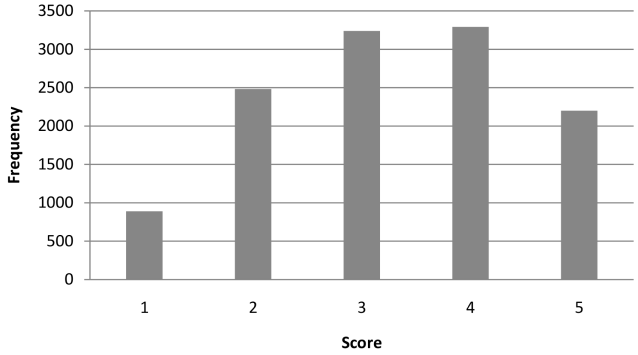

Fig. 3. Distribution of readability scores made by 120 human annotators on code snippets taken from several open source projects (see Figure 10).

without placing any time pressure on the activity itself (e.g., there was no incentive to make rushed readability judgments); this setup was made clear to participants. All collected data was kept carefully anonymous, and participants were aware of this fact: completing the survey yielded a randomly-generated code that could be monetarily redeemed. In Section 4.2 we discuss the effect of experience on readability judgments. In Section 7 we discuss the implications of our participant pool on experiment validity.

Statistic	Avg—Humans	Avg—Model
Cohen's κ	0.18 (p=0.0531)	0.07 (p=0.0308)
Weighted κ	0.33 (p=0.0526)	0.27 (p=0.0526)
Kendall's τ	0.44 (p=0.0090)	0.53 (p<0.0001)
Pearson's r	0.56 (p=0.0075)	0.63 (p<0.0001)
Spearman's ρ	0.55 (p=0.0089)	0.71 (p<0.0001)

Fig. 4. Five statistics for inter-annotator agreement. The "Avg—Humans" Column gives the average value of the statistic when applied between human annotator scores and the average human annotator score (or mode in the case of κ). The "Avg—Model" column show the value of the statistic between our model's output and the average (or mode) human readability scores. In both cases we give the corresponding p-values for the null-hypothesis (the probability that the correlation is random).

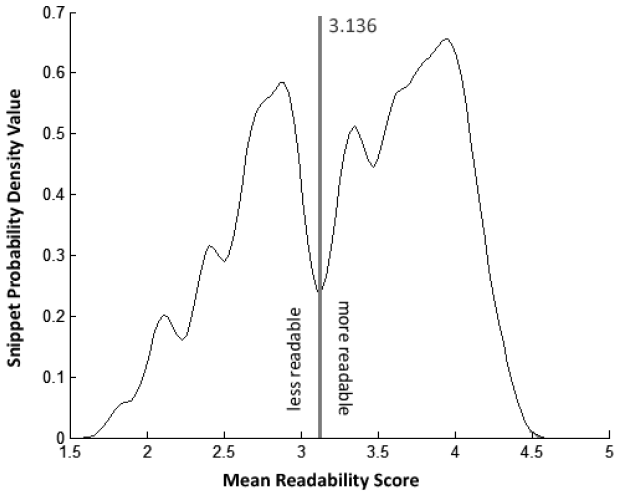

Fig. 5. Distribution of the average readability scores across all the snippets. The bimodal distribution presents us with a natural cutoff point from which we can train a binary classifier. The curve is a probability-density representation of the distribution with a window size of 0.8.

3 Study Results

Our 120 annotators each scored 100 snippets for a total of 12,000 distinct judgments. Figure 2 provides a graphical representation of this publicly-available data.[2] The distribution of scores can be seen in Figure 3.

First, we consider inter-annotator agreement, and evaluate whether we can extract a single coherent model from this data set. For the purpose of measuring agreement, we consider several correlation statistics. One possibility is Cohen's κ which is often used as a measure of inter-rater agreement for categorical items. However, the fact that our judgment data is ordinal (i.e., there is a qualitative relationship and total order between categories) is an important statistical consideration. Since the annotators did not receive precise guidance on how to score snippets, absolute score differences are not as important as relative ones. If two annotators both gave snippet X a higher score than snippet Y, then we consider them to be in agreement with respect to those two snippets, even if the actual numerical score values differ. Thusly, in this study we tested a linear-weighted version of κ (which conceptually gives some credit for rankings that are "close"). In addition, we considered Kendall's τ (i.e., the number of bubble-sort operations to order one list in the same way as a second), Pearson's r (measures the degree of linear dependence) and Spearman's ρ (the degree of dependence with any arbitrary monotonic function, closely related to Pearson's r) [37]. For these statistics, a correlation of 1 indicates perfect correlation, and 0 indicates no correlation (i.e., uniformly random scoring with only random instances of agreement). In the case of Pearson a correlation of 0.5 would arise, for example, if two annotators scored half of the snippets exactly the same way, and then scored the other half randomly.

We can combine our large set of judgments into a single model simply by averaging them. Because each of the correlation statistics compares the judgments of two annotators at a time. We extend it by finding the average correlation between our unified model and each annotator, the results are tabulated in Figure 4. In the case of κ and Weighted κ we use the *mode* of the scores because discrete values are expected. Translating this sort of statistic into qualitative terms is difficult, but correlation greater than 0.5 (Pearson/Spearman) is typically considered to be moderate to strong for a human-related study[13]. We use this unified model in our subsequent experiments and analyses. We employ Spearman's ρ throughout this study as our principle measure of agreement because it is the most general, and it appears to model the greatest degree of agreement. Figure 5 shows the range of agreements.

This analysis seems to confirm the widely-held belief that humans agree significantly on what readable code looks like, but not to an overwhelming extent. One implication is that there are, indeed, underlying factors that influence readability of code. By modeling the average score, we can capture most of these common factors, while simultaneously omitting those that arise largely from personal preference.

4 Readability Model

We have shown that there is significant agreement between our group of annotators on the relative readability of snippets. However, the processes that underlie this correlation are unclear. In this section, we explore the extent to which we can mechanically predict human readability judgments. We endeavor to determine which code features are predictive of readability, and construct a model (i.e., an automated software readability metric) to analyze other code.

2. The dataset and our tool are available at http://www.cs.virginia.edu/~weimer/readability

Avg.	Max.	Feature Name
✓	✓	line length (# characters)
✓	✓	# identifiers
✓	✓	identifier length
✓	✓	indentation (preceding whitespace)
✓	✓	# keywords
✓	✓	# numbers
✓		# comments
✓		# periods
✓		# commas
✓		# spaces
✓		# parenthesis
✓		# arithmetic operators
✓		# comparison operators
✓		# assignments (=)
✓		# branches (if)
✓		# loops (for, while)
✓		# blank lines
	✓	# occurrences of any single character
	✓	# occurrences of any single identifier

Fig. 6. The set of features considered by our metric. Read "#" as "number of . . . "

4.1 Model Generation

First, we form a set of features that can be detected statically from a snippet or other block of code. We have chosen features that are relatively simple, and that intuitively seem likely to have some effect on readability. They are factors related to structure, density, logical complexity, documentation, and so on. Importantly, to be consistent with our notion of readability as discussed in Section 2.1, each feature is independent of the size of a code block. Figure 6 enumerates the set of code features that our metric considers when judging code readability. Each feature can be applied to an arbitrary sized block of Java source code, and each represents either an average value per line, or a maximum value for all lines. For example, we have a feature that represents the average number of identifiers in each line, and another that represents the maximum number in any one line. The last two features listed in Figure 6 detect the character and identifier that occur most frequently in a snippet, and return the number of occurrences found. Together, these features create a mapping from snippets to vectors of real numbers suitable for analysis by a machine-learning algorithm.

Earlier, we suggested that human readability judgments may often arise from a complex interaction of features, and furthermore that the important features and values may be hard to locate. As a result, simple methods for establishing correlation may not be sufficient. Fortunately, there are a number of machine learning algorithms designed precisely for this situation. Such algorithms typically take the form of a *classifier* which operates on *instances* [30]. For our purposes, an instance is a feature vector extracted from a single snippet. In the training phase, we give a classifier a set of instances along with a labeled "correct answer" based on the readability data from our annotators. The labeled correct answer is a binary judgment partitioning the snippets into "more readable" and "less readable" based on the human annotator data. We designate snippets that received an average score below 3.14 to be "less readable" based on the natural cutoff from the bimodal distribution in Figure 5. We group the remaining snippets and consider them to be "more readable." Furthermore, the use of binary classifications also allows us to take advantage of a wider variety of learning algorithms.

When the training is complete, we apply the classifier to an instance it has not seen before, obtaining an estimate of the probability that it belongs in the "more readable" or "less readable" class. This allows us to use the probability that the snippet is "more readable" as a score for readability. We used the Weka [18] machine learning toolbox.

We build a classifier based on a set of features that have predictive power with respect to readability. To help mitigate the danger of over-fitting (i.e., of constructing a model that fits only because it is very complex in comparison the amount of data), we use 10-fold cross validation [23] (in Section 4.2 we discuss the results of a principle component analysis designed to help us understand the true complexity of the model relative to the data). 10-fold cross validation consists of randomly partitioning the data set into 10 subsets, training on 9 of them and testing on the last one. This process is repeated 10 times, so that each of the 10 subsets is used as the test data exactly once. Finally, to mitigate any bias arising from the random partitioning, we repeat the entire 10-fold validation 10 times and average the results across all of the runs.

4.2 Model Performance

We now test the hypothesis that local textual surface features of code are sufficient to capture human notions of readability. Two relevant success metrics in an experiment of this type are recall and precision. Here, recall is the percentage of those snippets judged as "more readable" by the annotators that are classified as "more readable" by the model. Precision is the fraction of the snippets classified as "more readable" by the model that were also judged as "more readable" by the annotators. When considered independently, each of these metrics can be made perfect trivially (e.g., a degenerate model that always returns "more readable" has perfect recall). We thus weight them together using the f-measure statistic, the harmonic mean of precision and recall [7]. This, in a sense, reflects the accuracy of the classifier with respect to the "more readable" snippets. We also consider the overall accuracy of the classifier by finding the percentage of correctly classified snippets.

We performed this experiment on ten different classifiers. To establish a baseline, we trained each classifier

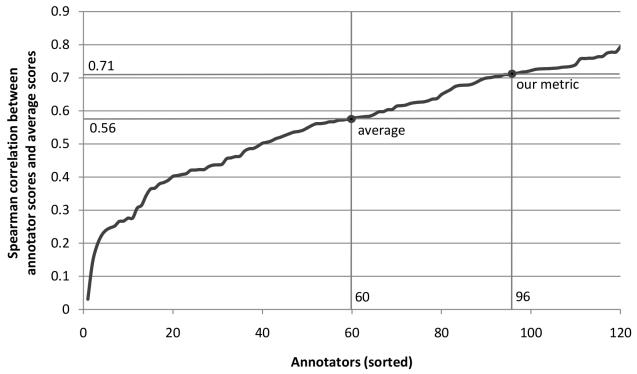

Fig. 7. Inter-annotator agreement. For each annotator (in sorted order), the value of Spearman's ρ between the annotator's judgments and the average judgments of all the annotators (the model that we attempt to predict). Also plotted is Spearman's ρ for our metric as compared to the average of the annotators.

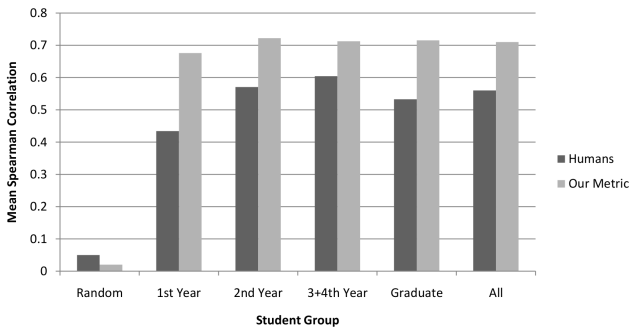

Fig. 8. Annotator agreement by experience group. "Random" represents a baseline of uniformly distributed random annotations.

on the set of snippets with randomly generated score labels. Guessing randomly yields an f-measure of 0.5 and serves as a baseline, while 1.0 represents a perfect upper bound. None of the classifiers were able to achieve an f-measure of more than 0.61 (note, however, that by always guessing 'more readable' it would actually be trivial to achieve an f-measure of 0.67). When trained on the average human data (i.e., when not trained randomly), several classifiers improved to over 0.8. Those models included the multilayer perceptron (a neural network), the Bayesian classifier (based on conditional probabilities of the features), a Logistic Regression, and the Voting Feature Interval approach (based on weighted "voting" among classifications made by each feature separately). On average, these three best classifiers each correctly classified between 75% and 80% of the snippets. We view a model that is well-captured by multiple learning techniques as an advantage: if only one classifier could agree with our training data, it would have suggested a lack of generality in our notion of readability.

While an 80% correct classification rate seems reasonable in absolute terms, it is perhaps simpler to appreciate in relative ones. When we compare continuous the output of the Bayesian classifier (i.e., we use a probability estimate of "more readable" rather a binary classification) to the average human score model it was trained against, we obtain a Spearman correlation of 0.71. As shown in Figure 7, that level of agreement is 16% than what the average human in our study produced. Column 3 of Figure 4 presents the results in terms of four other statistics. While we could attempt to employ more exotic classifiers or investigate more features to improve this result, it is not clear that the resulting model would be any "better" since the model is already well within the margin of error established by our human annotators. In other words, in a very real sense, this metric is "just as good" as a human. For performance we can thus select any classifier in that equivalence class, and we choose to adopt the Bayesian classifier for the experiments in this paper because of its run-time efficiency.

We also repeated the experiment separately with each annotator experience group (e.g., first year CS students, second year CS students). Figure 8 shows the mean Spearman correlations. The dark bars on the left show the average agreement between humans and the average score vector for their group (i.e., inter-group agreement). For example, third and fourth year CS students agree with each other more often (Spearman correlation of approximately 0.6) than do first year CS students (correlation under 0.5). The light bar on the right indicates the correlation between our metric (trained on the annotator judgments for that group) and the average of all annotators in the group. Three interesting observations arise. First, for all groups, our automatic metric agrees with the human average more closely than the humans agree. Second, we see a gradual trend toward increased agreement with experience, except for graduate students. We suspect that the difference with respect to graduates may a reflection of the more diverse background of the graduate student population, their more sophisticated opinions, or some other external factor. And third, the performance of our model is very consistent across all four groups, implying that to some degree it is robust to the source of training data.

We investigated which features have the most predictive power by re-running our all-annotators analysis using only one feature at a time. The relative magnitude of the performance of the classifier is indicative of the comparative importance of each feature. Figure 9 shows the result of that analysis with the magnitudes normalized between zero and one.

We find, for example, that factors like 'average line length' and 'average number of identifiers per line' are very important to readability. Conversely, 'average identifier length' is not, in itself, a very predictive factor; neither are `if` constructs, loops, or comparison operators. Section 6 includes a discussion of some of the possible implications of this result. A few of these figures are similar to those used by automated complexity metrics [27]: in Section 5.2 we investigate the overlap

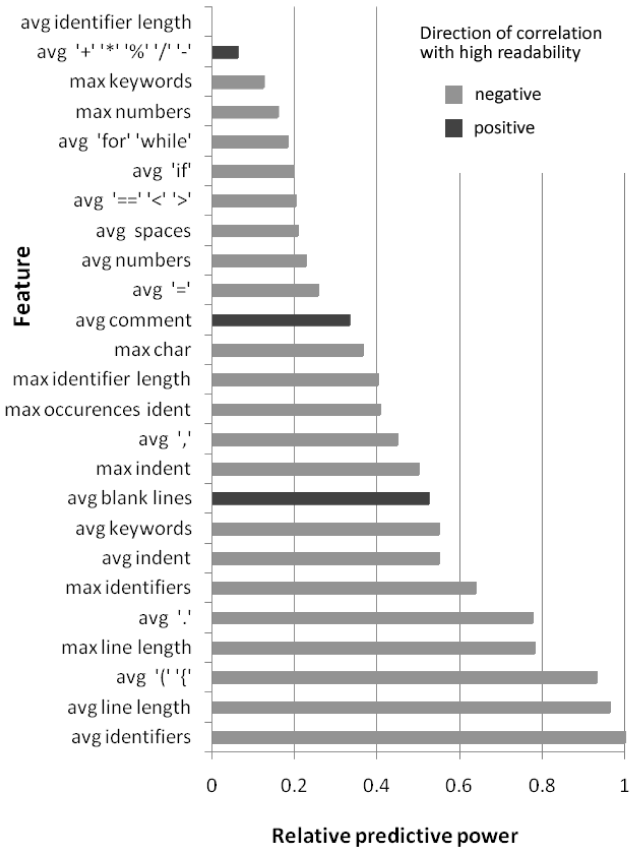

Fig. 9. Relative power of features as determined by a singleton (one-feature-at-a-time) analysis. The direction of correlation for each is also indicated.

between readability and complexity.

We prefer this singleton feature analysis to a leave-one-out analysis (which judges feature power based on decreases in classifier performance) that may be misleading due to significant feature overlap. This occurs when two or more features, though different, capture the same underlying phenomena. As a simple example, if there is exactly one space between every two words then a feature that counts words and a feature that counts spaces will capture essentially the same information and leaving one of them out is unlikely to decrease accuracy. A principal component analysis (PCA) indicates that 95% of the total variability can be explained by 8 principal components, thus implying that feature overlap is significant. The total cumulative variance explained by the first 8 principal components is as follows {41%, 60%, 74%, 81%, 87%, 91%, 93%, 95%}.

5 Correlating Readability with Software Quality

In the previous section we constructed an automated model of readability that mimics human judgments. We implemented our model in a tool that assesses the readability of programs. In this section we use that tool to test the hypothesis that readability (as captured by our model) correlates with external conventional metrics of software quality. Specifically, we first test for a correlation between readability and FindBugs, a popular static bug-finding tool [19]. Second, we test for a similar correlation with changes to code between versions of several large open source projects. Third, we do the same for version control log messages indicating that a bug has been discovered and fixed. Then, we examine whether readability correlates with Cyclomatic complexity [27] to test our earlier claim that our notion of readability is largely independent of inherent complexity. Finally, we look for trends in code readability across software projects.

The set of open source Java programs we have employed as benchmarks can be found in Figure 10. They were selected because of their relative popularity, diversity in terms of development maturity and application domain, and availability in multiple versions from *SourceForge*, an open source software repository. Maturity is self reported, and categorized by *SourceForge* into 1-planning, 2-pre-alpha, 3-alpha, 4-beta, 5-production/stable, 6-mature, 7-inactive. Note that some projects present multiple releases at different maturity levels; in such cases we selected the release for the maturity level indicated.

Running our readability tool (including feature detection and the readability judgment) was quite rapid. For example, the 98K lines of code in SoapUI took less than 16 seconds to process (about 6K LOC per second) on a machine with a 2GHz processor and disk with a maximum 150 MBytes/sec transfer rate.

5.1 Readability Correlations

Our first experiment tests for a correlation between defects detected by FindBugs with our readability metric at the function level. We first ran FindBugs on the benchmark, noting defect reports. Second, we extracted all of the functions and partitioned them into two sets: those containing at least one reported defect, and those containing none. To avoid bias between programs with varying numbers of reported defects, we normalized the function set sizes. We then ran the already-trained classifier on the set of functions, recording an f-measure for "contains a bug" with respect to the classifier judgment of "less readable." The purpose of this experiment is to investigate the extend to which our model correlates with an external notion of code quality in aggregate.

Our second experiment tests for a similar correlation between "code churn" and readability. Version-to-version changes capture another important aspect of code quality. This experiment used the same setup as the first, but used readability to predict which functions will be modified between two successive releases of a program. For this experiment, "successive release" means the two most recent stable versions. In other words, instead of "contains a bug" we attempt to predict

Project Name	KLOC	Maturity	Description
Azureus: Vuze 4.0.0.4	651	5	Internet File Sharing
JasperReports 2.04	269	6	Dynamic Content
Hibernate* 2.1.8	189	6	Database
jFreeChart* 1.0.9	181	5	Data representation
FreeCol* 0.7.3	167	3	Game
TV Browser 2.7.2	162	5	TV Guide
jEdit* 4.2	140	5	Text editor
Gantt Project 3.0	130	5	Scheduling
SoapUI 2.0.1	98	6	Web services
Data Crow 3.4.5	81	5	Data Management
Xholon 0.7	61	4	Simulation
Risk 1.0.9.2	34	4	Game
JSch 0.1.37	18	3	Security
jUnit* 4.4	7	5	Software development
jMencode 0.64	7	3	Video encoding

Fig. 10. Benchmark programs used in our experiments. The "Maturity" column indicates a self-reported *SourceForge* project status. *Used as a snippet source.

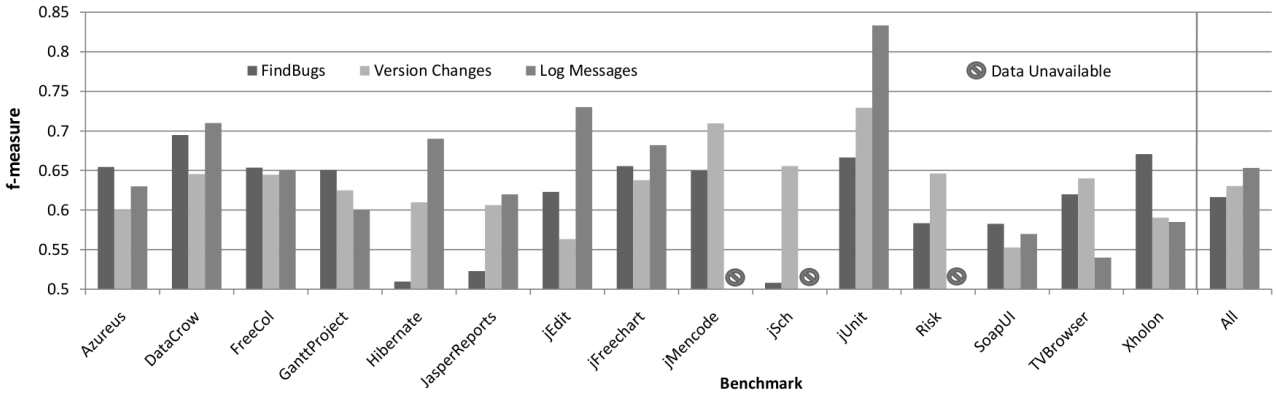

Fig. 11. f-measure for using readability to predict functions that: show a FindBugs defect, have change between releases, and have a defect referenced in a version control log message. For log messages, "Data Unavailable" indicates that not enough version control information was available to conduct the experiment on that benchmark.

"is going to change soon." We consider a function to have changed in any case where the text is not exactly the same, including changes to whitespace. Whitespace is normally ignored in program studies, but since we are specifically focusing on readability we deem it relevant.

While our first experiment looks at output from a bug finder, such output may not constitute true defects in code. Our third experiment investigates the relationship between readability and defects that have actually been noted by developers. It has become standard software engineering practice to use version control repositories to manage modifications to a code base. In such a system, when a change is made, the developer typically includes a log message describing the change. Such log messages may describe a software feature addition, an optimization, or any number of other potential changes. In some cases log messages even include a reference to a "bug" — often, but not always, using the associated bug tracking number from the project's bug database. In this third experiment we use our metric for readability to predict whether a function has been modified with a log message that mentions a bug.

Twelve of our fifteen benchmarks feature publicly accessible version control systems with at least 500 revisions (most have thousands). For the most recent version of each program, for each line of code, we determine the last revision to have changed that line (e.g., this is similar to the common `cvs blame` tool). We restrict our evaluation to the most recent 1000 revisions of each benchmark and find 5087 functions with reported bugs out of 111,670 total. We then inspect the log messages for the word "bug" to determine whether a change was made to address a defect or not. We partition all functions into two sets: those where at least one line was last changed to deal with a bug, and those where no lines were last changed to deal with a bug.

Figure 11 summarizes the results of these three experiments. The average f-measure over our benchmarks

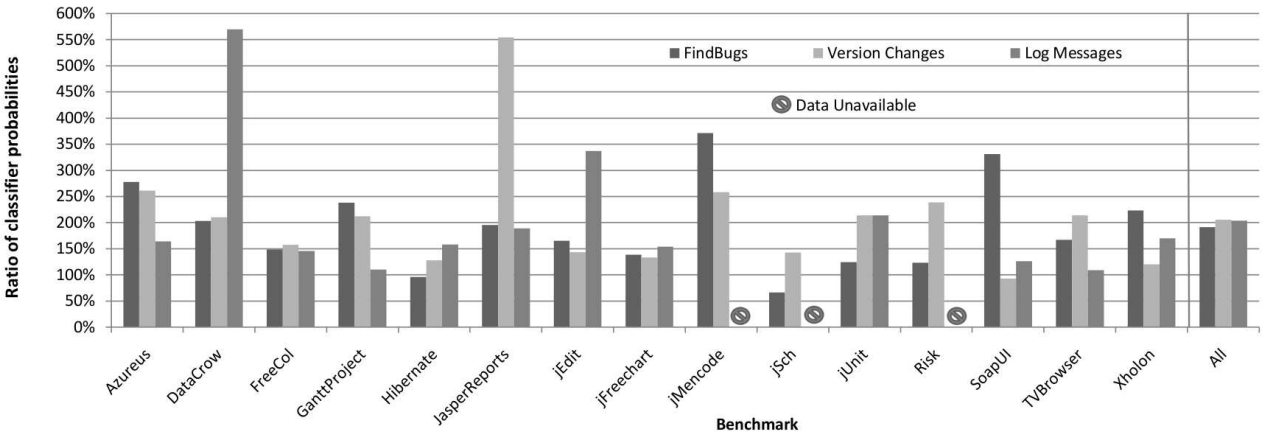

Fig. 12. Mean ratio of the classifier probabilities (predicting 'less readable) assigned to functions that contained a FindBugs defect, that will change in the next version, or that had a defect reported in a version control log. For example, FreeCol functions with FindBug errors were assigned a probability of 'less readable' that was nearly 150% greater on average than the probabilities assigned to functions without such defects.

for the FindBugs correlation is 0.62 (precision=0.90, recall=0.48), for version changes it is 0.63 (precision=0.89, recall=0.49), and for log messages indicating a bug fix in the twelve applicable benchmarks, the average f-measure is 0.65 (precision=0.92, recall=0.51). It is important to note that our goal is *not* perfect correlation with FindBugs or any other source of defect reports: projects can run FindBugs directly rather than using our metric to predict its output. Instead, we argue that our readability metric has general utility and is correlated with multiple notions of software quality. This is best shown by our strong performance when correlating with bug-mentioning log messages: code with low readability is significantly more likely to be changed by developers later for the purposes of fixing bugs.

A second important consideration is the *magnitude* of the difference. We claim that classifier probabilities (i.e., continuous output versus discrete classifications) are useful in evaluating readability. Figure 12 presents this data in the form of a ratio, the mean probability assigned by the classifier to functions positive for FindBugs defects or version changes to functions without these features. A ratio over 1 (i.e., $> 100\%$) for many of the projects indicates that the functions with these features tend to have lower readability scores than functions without them. For example, in the jMencode and SoapUI projects, functions judged less readable by our metric were dramatically more likely to contain FindBugs defect reports, and in the JasperReports project less-readable methods were very likely to change in the next version.

As a brief note: we intentionally do not report standard deviations for readability distributions. Both the underlying score distribution that our metric is based on (see Figure 5) and the output of our tool itself are bimodal. In fact, the tool output on our benchmarks more closely approximates a bathtub or uniform random distribution than a normal one. As a result, standard inferences about the implications of the standard deviation do not apply. However, the mean of such a distribution does well-characterize the ratio of methods from the lower half of the distribution to the upper half (i.e., it characterizes the population of 'less-readable' methods compared to 'more-readable' ones).

For each of these three external quality indicators we found that our tool exhibits a substantial degree of correlation. Predicting based on our readability metric yields an f-measure over 0.8 in some case. Again, our goal is not a perfect correlation with version changes and code churn. These moderate correlations do, however, add support to the hypothesis that a substantial connection exists between code readability, as described by our model, and defects and upcoming code changes

5.2 Readability and Complexity

In this paper, we defined readability as the "accidental" component of code understandability, referring to the idea that it is an artifact of writing code and not closely related to the complexity of the problem that the code is meant to solve. Our short snippet selection policy masks complexity, helping to tease it apart from readability. We also selected a set of surface features designed to be agnostic to the length of any selection of code. Nonetheless, some of the features still may capture some amount of complexity. We now test the hypothesis that readability and complexity are *not* closely related.

We use McCabe's Cyclomatic measure [27] as an indicative example of an automated model of code complexity. We measured the Cyclomatic complexity and readability of each method in each benchmark. Since both quantities are ordinal (as opposed to binary as in our three previous experiments), we computed Pearson's r between them. We also computed, for use as a baseline, the correlation between Cyclomatic complexity and method length. Figure 13 shows that readability is not

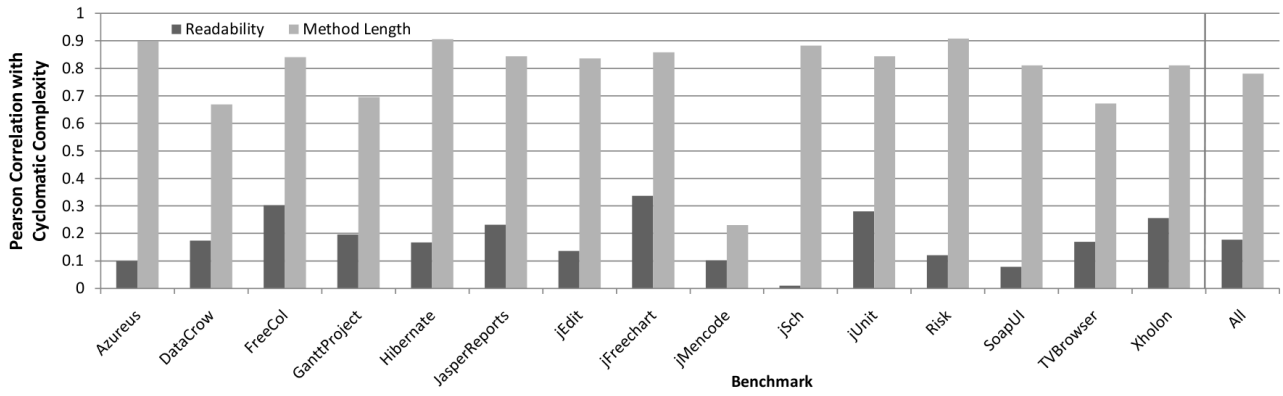

Fig. 13. Pearson Product moment correlation between Cyclomatic complexity and readability as well as between Cyclomatic complexity and method length (number of statements in the method). Readability is at most weakly correlated with complexity in an absolute sense. In a relative sense, compared to method length, readability is effectively uncorrelated with complexity. These results imply that readability captures an aspect of code that is not well modeled by a traditional complexity measure.

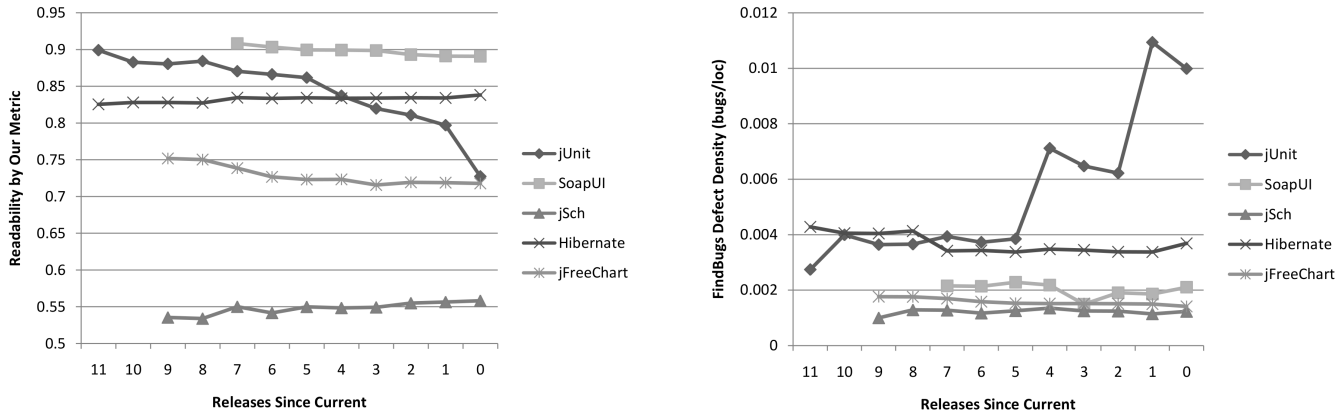

Fig. 14. The left graph shows the average readability metric of all functions in a project as a function of project lifetime. The right graph shows the FindBugs defect density in a project as a function of project lifetime for the same projects and releases. Note that the projects with flat readabilities have corresponding flat defect densities, while jUnit (described in text) becomes significantly less readable and significantly more buggy starting at release 5.

closely related to this traditional notion of complexity. Method length, on the other hand, is much more strongly related to complexity. We thus conclude that while our notion of readability is not orthogonal to complexity, it is in large part modeling a distinct phenomena.

5.3 Software Lifecycle

To further explore the relation of our readability metric to external factors, we investigate changes over long periods of time. We hypothesize that an observed trend in the readability of a project will manifest as a similar trend in project defects. Figure 14 shows a longitudinal study of how readability and defect rates tends to change over the lifetime of a project. To construct this figure we selected several projects with rich version histories and calculated the average readability level over all of the functions in each. Each of the projects shows a linear relationship at a statistical significance level (p-value) of better than 0.05 except for defect density with jFreeChart.

Note that newly-released versions for open source projects are not always more stable than their predecessors. Projects often undergo major overhauls or add additional cross cutting features. Consider jUnit, which has recently adopted a "completely different API ...[that] depends on new features of Java 5.0 (annotations, static import...)" [20].

The rightmost graph in Figure 14 plots FindBugs-reported defect density over multiple project releases. Most projects, such as Hibernate and SoapUI, show a relatively flat readability profile. These projects experience a similarly flat defect density. The jUnit project, however, shows a sharp decline in readability from releases 5 to 0 (as the new API is introduced) and a corresponding increase in defect density from releases 5 to 0. For JUnit, we observe that 58.9% of functions have readability

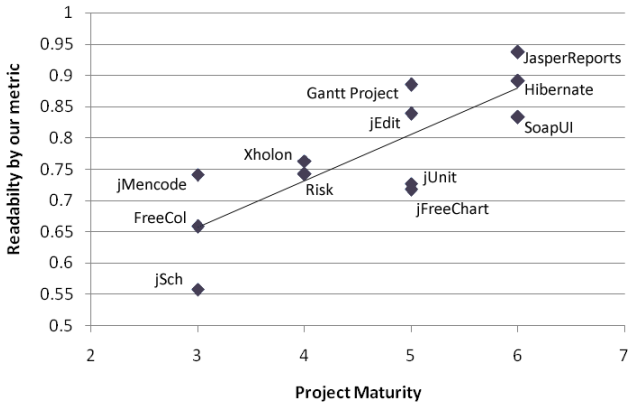

Fig. 15. Average readability metric of all functions in a project as a function of self-reported project maturity with best fit linear trend line. Note that projects of greater maturity tend to exhibit greater readability.

below 0.5 in the earliest version we tested, and 71.4% in the most recent (an increase of 12.5%). Of the remaining four systems, SoapUI had the greatest change: 44.3% below 0.5 to 48.1% (an increase of 3.8%). In this case study, our readability metric correlates strongly with software quality metrics both between different projects and also between different releases of the same project.

We conducted one additional study to measure readability against maturity and stability. Figure 15 plots project readability against project maturity, as self-reported by developers. The data shows a noisy, but statistically significant (Pearson's $r = 0.80$ with p$= 0.00031$), upward trend implying that projects that reach maturity tend to be more readable. For example, every "6–mature" project is more readable than every "3–alpha" project.

6 Discussion

This study includes a significant amount of empirical data about the relation between local code features and readability. We believe that this information may have implications for the way code should be written and evaluated, and for the design of programming languages. However, we caution that this data may only be truly relevant to our annotators; it should not be interpreted to represent a comprehensive or universal model for readability. Furthermore, by the nature of a descriptive model, it may not be suitable for directly prescribing coding practices. However, we believe it can be useful to identify aspects of code readability which should be more carefully considered.

To start, we found that the length of identifier names constitutes almost no influence on readability (0% relative predictive power). This observation fails to support the common belief that "single-character identifiers ...[make the] ...maintenance task much harder" [22]. An observation which perhaps contributed to a significant movement toward "self documenting code" which is often characterized by long and descriptive identifier names and few abbreviations. The movement has had particular influence on the Java community. Furthermore, naming conventions, like the "Hungarian" notation which seeks to encode typing information into identifier names, should be considered carefully [36]. In our study, the average identifier length had near zero predictive power, while maximum identifier length was much more useful as a negative predictor. While we did not include any features to detect encoded type information or other variable naming conventions, paradigms that result in longer identifiers without conveying additional information may negatively impact readability.

Unlike identifiers, comments are a very direct way of communicating intent. One might expect their presence to increase readability dramatically. However, we found that comments were are only moderately well-correlated with our annotators' notion of readability (33% relative power). One conclusion may be that while comments can enhance readability, they are typically used in code segments that started out less readable: the comment and the unreadable code effectively balance out. The net effect would appear to be that comments are not always, in and of themselves, indicative of high or low readability.

The number of identifiers and characters per line has a strong influence on our readability metric (100% and 96% relative power respectively). It would appear that just as long sentences are more difficult to understand, so are long lines of code. Our findings support the conventional wisdom that programmers should keep their lines short, even if it means breaking up a statement across multiple lines.

When designing programming languages, readability is an important concern. Languages might be designed to force or encourage improved readability by considering the implications of various design and language features on this metric. For example, Python enforces a specific indentation scheme in order to aid comprehension [29], [40]. In our experiments, the importance of character count per line suggests that languages should favor the use of constructs, such as switch statements and pre- and post-increment, that encourage short lines. Our data suggests that languages should add additional keywords if it means that programs can be written with fewer new identifiers.

It is worth noting that our model of readability is *descriptive* rather than *normative* or *prescriptive*. That is, while it can be used to predict human readability judgments for existing software, it cannot be directly interpreted to prescribe changes that will improve readability. For example, while "average number of blank lines" is a powerful feature in our metric that is positively correlated with high readability, merely inserting five blank lines after every existing line of code need not improve human judgments of that code's readability. Similarly, long identifiers contribute to lower readability scores in our model, but replacing all identifiers with random

two-letter sequences is unlikely to help. We might tease apart such relationships and refine our model by applying model-prescribed changes to pieces of code and then evaluating their actual change in readability via a second human study: differences between predicted changes and observed changes will help illuminate confounding variables and imprecise features in our current model. Learning such a normative model of software readability remains as future work.

Finally, as language designers consider new language features, it might be useful to conduct studies of the impact of such features on readability. The techniques presented in this paper offer a framework for conducting such experiments.

7 Threats to Validity

One potential threat to validity concerns the pool of participants for our study. In particular, our participants were taken largely from introductory and intermediate computer science courses, implying that they have had little previous experience reading or writing code as compared to industrial practitioners. Furthermore, they may possess some amount of bias to certain coding practices or idioms resulting from a uniform instruction at the same institution.

Because our model is built only upon the opinions of computer science students at *The University of Virginia*, it is only a model of readability according to them. Nonetheless, the metric we present shows significant correlation with three separate external notions of software quality. Furthermore, our study shows that even graduate students, who have widely varying educational and professional backgrounds, show a strong level of agreement with each other and with our metric. This indicates that annotator bias of this type may be small; however, we did not study it directly in this paper. Finally, rather than presenting a final or otherwise definitive metric for readability, we present an initial metric coupled with a methodology for constructing further metrics from an arbitrary population.

We also consider our methodology for scoring snippets as a potential threat to validity. To capture an intuitive notion of readability, rather than a constrained one, we did not provide specific guidance to our participants on how to judge readability. It is possible that inter-annotator agreement would be much greater in a similar study that included a precise definition of readability (e.g., a list of factors to consider). In is unclear how our modeling technique would perform under those circumstances. Similarly, both practice and fatigue factors may affect the level of agreement or bias the model. However, we do not observe a significant trend in the variance of the score data across the snippet set (a linear regression on score variance has a slope of 0.000 and R-squared value of 0.001, suggesting that practice effects were minimal).

8 Future Work

The techniques presented in this paper should provide an excellent platform for conducting future readability experiments, especially with respect to unifying even a very large number of judgments into an accurate model of readability.

While we have shown that there is significant agreement between our annotators on the factors that contribute to code readability, we would expect each annotator to have personal preferences that lead to a somewhat different weighting of the relevant factors. It would be interesting to investigate whether a personalized or organization-level model, adapted over time, would be effective in characterizing code readability. Furthermore, readability factors may also vary significantly based on application domain. Additional research is needed to determine the extent of this variability, and whether specialized models would be useful.

Another possibility for improvement would be an extension of our notion of local code readability to include broader features. While most of our features are calculated as average or maximum value per line, it may be useful to consider the size of compound statements, such as the number of simple statements within an if block. For this study, we intentionally avoided such features to help ensure that we were capturing readability rather than complexity. However, in practice, achieving this separation of concerns is likely to be less compelling.

Readability measurement tools present their own challenges in terms of programmer access. We suggest that such tools could be integrated into an IDE, such as Eclipse, in the same way that natural language readability metrics are incorporated into word processors. Software that seems readable to the author may be quite difficult for others to understand [16]. Such a system could alert programmers as such instances arise, in a way similar to the identification of syntax errors.

Finally, in line with conventional readability metrics, it would be worthwhile to express our metric using a simple formula over a small number of features (the PCA from Section 4.2 suggests this may be possible). Using only the truly essential and predictive features would allow the metric to be adapted easily into many development processes. Furthermore, with a smaller number of coefficients the readability metric could be parameterized or modified in order to better describe readability in certain environments, or to meet more specific concerns.

9 Conclusion

In this paper we have presented a technique for modeling code readability based on the judgments of human annotators. In a study involving 120 computer science students, we have shown that it is possible to create a metric that agrees with these annotators as much as they agree with each other by only considering a relatively simple set of low-level code features. In addition, we

have seen that readability, as described by this metric, exhibits a significant level of correlation with more conventional metrics of software quality, such as defects, code churn, and self-reported stability. Furthermore, we have discussed how considering the factors that influence readability has potential for improving the programming language design and engineering practice with respect to this important dimension of software quality. Finally, it is important to note that the metric described in this paper is not intended as the final or universal model of readability.

REFERENCES

[1] K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," *Reliability and Maintainability Symposium*, pp. 235–241, Sep. 2002.

[2] S. Ambler, "Java coding standards," *Softw. Dev.*, vol. 5, no. 8, pp. 67–71, 1997.

[3] B. B. Bederson, B. Shneiderman, and M. Wattenberg, "Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies," *ACM Trans. Graph.*, vol. 21, no. 4, pp. 833–854, 2002.

[4] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[5] R. P. L. Buse and W. R. Weimer, "A metric for software readability," in *International Symposium on Software Testing and Analysis*, 2008, pp. 121–130.

[6] L. W. Cannon, R. A. Elliott, L. W. Kirchhoff, J. H. Miller, J. M. Milner, R. W. Mitze, E. P. Schan, N. O. Whittington, H. Spencer, D. Keppel, , and M. Brader, *Recommended C Style and Coding Standards: Revision 6.0*. Seattle, Washington: Specialized Systems Consultants, Inc., June 1990.

[7] T. Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of the f-measure," in *International Conference on Quality Software*, 2004, pp. 146–153.

[8] L. E. Deimel Jr., "The uses of program reading," *SIGCSE Bull.*, vol. 17, no. 2, pp. 5–14, 1985.

[9] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall PTR, 1976.

[10] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 25, no. 8, pp. 512–521, 1982.

[11] R. F. Flesch, "A new readability yardstick," *Journal of Applied Psychology*, vol. 32, pp. 221–233, 1948.

[12] J. Frederick P. Brooks, "No silver bullet: essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[13] L. L. Giventer, *Statistical Analysis in Public Administration*. Jones and Bartlett Publishers, 2007.

[14] J. Gosling, B. Joy, and G. L. Steele, *The Java Language Specification*, ser. The Java Series. Reading, MA, USA: Addison-Wesley, 1996.

[15] R. Gunning, *The Technique of Clear Writing*. New York: McGraw-Hill International Book Co, 1952.

[16] N. J. Haneef, "Software documentation and readability: a proposed process improvement," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 3, pp. 75–77, 1998.

[17] A. E. Hatzimanikatis, C. T. Tsalidis, and D. Christodoulakis, "Measuring the readability and maintainability of hyperdocuments," *Journal of Software Maintenance*, vol. 7, no. 2, pp. 77–90, 1995.

[18] G. Holmes, A. Donkin, and I. Witten, "Weka: A machine learning workbench," *Australia and New Zealand Conference on Intelligent Information Systems*, 1994.

[19] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[20] jUnit.org, "jUnit 4.0 now available," in *http://sourceforge.net/forum/forum.php?forum_id=541181*, Feb. 2006.

[21] J. P. Kinciad and E. A. Smith, "Derivation and validation of the automated readability index for use with technical materials," *Human Factors*, vol. 12, pp. 457–464, 1970.

[22] J. C. Knight and E. A. Myers, "Phased inspections and their implementation," *SIGSOFT Softw. Eng. Notes*, vol. 16, no. 3, pp. 29–35, 1991.

[23] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," *International Joint Conference on Artificial Intelligence*, vol. 14, no. 2, pp. 1137–1145, 1995.

[24] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[25] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 140, pp. 44–53, 1932.

[26] S. MacHaffie, R. McLeod, B. Roberts, P. Todd, and L. Anderson, "A readability metric for computer-generated mathematics," Saltire Software, http://www.saltire.com/equation.html, Tech. Rep., retrieved 2007.

[27] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[28] G. H. McLaughlin, "Smog grading – a new readability," *Journal of Reading*, May 1969.

[29] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, pp. 861–867, 1983.

[30] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[31] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 284–292.

[32] C. V. Ramamoorthy and W.-T. Tsai, "Advances in software engineering," *Computer*, vol. 29, no. 10, pp. 47–58, 1996.

[33] D. R. Raymond, "Reading source code," in *Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1991, pp. 3–16.

[34] P. A. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," *Empirical Software Engineering, 2005. 2005 International Symposium on*, November 2005.

[35] S. Rugaber, "The use of domain knowledge in program understanding," *Ann. Softw. Eng.*, vol. 9, no. 1-4, pp. 143–192, 2000.

[36] C. Simonyi, "Hungarian notation," *MSDN Library*, November 1999.

[37] S. E. Stemler, "A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability," *Practical Assessment, Research and Evaluation*, vol. 9, no. 4, 2004.

[38] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 2004.

[39] T. Tenny, "Program readability: Procedures versus comments," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1271–1279, 1988.

[40] A. Watters, G. van Rossum, and J. C. Ahlstrom, *Internet Programming with Python*. New York: MIS Press/Henry Holt publishers, 1996.

[41] E. J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Softw. Eng.*, vol. 14, no. 9, pp. 1357–1365, 1988.

Raymond P.L. Buse received a BS degree in computer science from the Johns Hopkins University and an MCS from the University of Virginia, where he is currently a graduate student. His main research interests include static analysis techniques and the underlying human factors related to programming language design and software engineering practice.

Westley R. Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an assistant professor at the University of Virginia. His main research interests include static and dynamic analyses and program transformations to improve software quality.