

# A Metric for Software Readability

Raymond P.L. Buse and Westley R. Weimer  
Department of Computer Science  
University of Virginia  
Charlottesville, VA, USA  
{buse, weimer}@cs.virginia.edu

## ABSTRACT

In this paper, we explore the concept of code readability and investigate its relation to software quality. With data collected from 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 two traditional measures of software quality, code changes and defect reports. 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.

## Categories and Subject Descriptors

D.2.9 [Management]: Software quality assurance (SQA);  
D.2.8 [Software Engineering]: Metrics

## General Terms

Measurement, Human Factors

## Keywords

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 critical factor in overall software quality. Typically, maintenance will consume over 70% of the total lifecycle cost of a software product [5]. 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 [29, 36, 38]. Furthermore, maintaining software often means evolving software, and modifying existing code is a large part of modern software engineering [35]. Readability is so significant, in fact, that Elshoff and Marcotty proposed adding a development phase in which the program is made more readable [11]. Knight and Myers suggested that one phase of software inspection should be a check of the source code for readability [26]. Haneef proposed the addition of a dedicated readability and documentation group to the development team [19].

We hypothesize that everyone who has written code has some intuitive notion of this concept, and that program features such indentation (e.g., as in Python [43]), choice of identifier names [37], 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, and employed that notion to help motivate his top-down approach to system design [10].

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 both with human annotators and also with external 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 [12], the Gunning-Fog Index [18], the SMOG Index [31], and the Automated Readability Index [24] are just a few examples of readability metrics that were developed for ordinary text. These metrics are all based on simple factors such as average syllables per word and average sentence length. Despite their relative 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, while far from perfect, 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA '08, July 20–24, 2008, Seattle, Washington, USA.  
Copyright 2008 ACM 978-1-59593-904-3/08/07 ...\$5.00.

specifically at source code and backed with empirical evidence for effectiveness, can serve an analogous purpose in the software domain.

It is important to note that readability is not the same as complexity, for which some existing metrics have been empirically shown useful [44]. Brooks claims that complexity is an “essential” property of software; it arises from system requirements, and cannot be abstracted away [13]. Readability, on the other hand, is purely “accidental.” In the Brooks model, software engineers can only hope to control accidental difficulties: coincidental readability can be addressed far 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. It is not related, for example, to the size of a piece of code. Furthermore, our notion of readability arises directly from the judgments of actual human annotators who need not be familiar with the purpose of the system. Complexity factors, on the other hand, may have little relation to what makes code understandable to humans. Previous work [34] has shown that attempting to correlate artificial code complexity metrics directly to defects is difficult, but not impossible. 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 4 that there is indeed a significant correlation between readability and quality.

The main contributions of this paper are:

- An automatic software readability metric based on local features. Our metric correlates strongly with both human annotators and also external notions of software quality.
- A survey of 120 human annotators on 100 code snippets that forms the basis for our metric. We are unaware of any published software readability study of comparable size (12,000 human judgments).
- A discussion of the features involved in that metric and their relation to software engineering and programming language design.

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.

The structure of this paper is as follows. In Section 2 we investigate the extent to which our study group agrees on what readable code looks like, and in Section 3 we determine a small set of features that is sufficient to capture the notion of readability for a majority of annotators. In Section 4 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 5, place our work in context in Section 6, discuss possibilities for extension in Section 7, and conclude in Section 8.

## 2. HUMAN READABILITY ANNOTATION

A consensus exists that readability is an essential determining characteristic of code quality [1, 5, 10, 11, 19, 29, 34, 35, 36, 37, 38, 44], but not about which factors most contribute to human notions of software readability. A previous study by Tenny looked at readability by testing comprehension of several versions of a program [42]. 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 present a sequence of short code selections, called *snippets*, through a web interface. Each annotator is asked to individually score each snippet based on a personal estimation of how readable it is. There are two important parameters to consider: snippet selection policy and score range.

### 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 give the annotators the best chance at appreciating 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, the readability of a snippet consisting entirely of boilerplate import statements or entirely of comments is not particularly meaningful.

Consequently, an important tradeoff exists such that snippets must be as short as possible to adequately support analysis, 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 [16], 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. We find that with this set of policies, over 90% of statements in all of the programs we considered (see Figure 8) 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 directly to similar languages like C and C++.

## 2.2 Readability Scoring

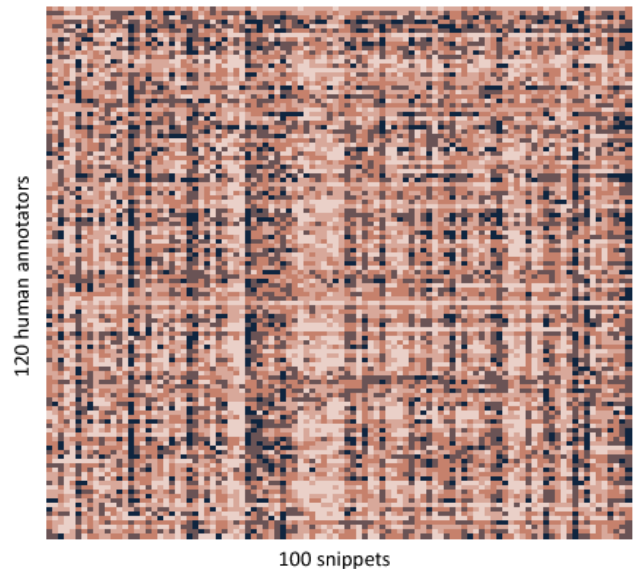
Prior to their participation, our volunteer human annotators were told that they would be asked to rate Java code on its readability, and that their participation would assist in a study of that aspect of software quality. Responses were collected using a web-based annotation tool, *Snippet Sniper*, that users were permitted to access at their leisure. Users were presented with a sequence of snippets and buttons labeled 1–5 [28]. Each user was shown the same set of one hundred snippets in the same order. Users 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 (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. 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.

Our 120 annotators each scored 100 snippets for a total of 12,000 distinct judgments. Figure 1 provides a graphical representation of this publicly-available data.<sup>1</sup> The distribution of scores can be seen in Figure 2. The annotators were all computer science students. They had varying experience reading and writing code as evidenced by their current course enrollment: 17 were taking 100-level courses, 63 were taking 200-level courses, 30 were taking 400-level courses, and 10 were graduate students. In Section 3.2 we will discuss the effect of experience on readability judgments.

The snippets were generated from five open source projects (see Figure 8). 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 7.

Next we consider inter-annotator agreement, and evaluate whether we can extract a single coherent model from this data set. The fact that this judgment data is ordinal

<sup>1</sup>The dataset is available at <http://www.cs.virginia.edu/~weimer/readability/data>

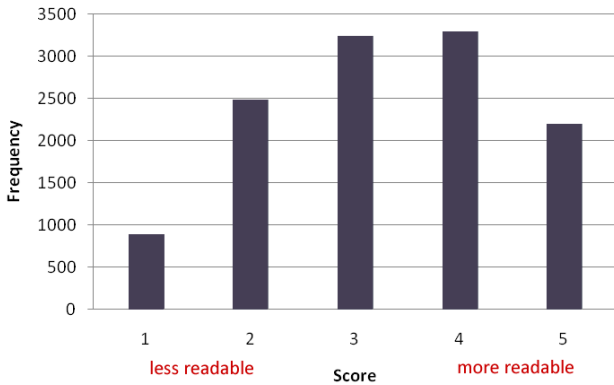


**Figure 1: The complete data set obtained for this study. Each box corresponds to a judgment made by a human annotator. Darker colors correspond to lower readability scores (e.g., 1 and 2) the lighter ones correspond to higher scores. Our metric for readability is derived from these 12,000 judgments. Vertical bands indicate snippets that were judged similarly by many annotators.**

(i.e., ranked), rather than simply nominal, 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. The most popular correlation statistic for this sort of data is the Pearson product-moment correlation coefficient [40]. A Pearson correlation of 1 indicates perfect correlation, and 0 indicates no correlation (i.e., uniformly random scoring with only random instances of agreement). 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 employ the Pearson statistic throughout this study as a measure of agreement.

We can combine our large set of judgments into a single model simply by averaging them. Pearson, like other similar 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, and obtain 0.56. Translating this sort of statistic into qualitative terms is difficult, but correlation at this level is typically considered to be moderate to strong. We use this unified model in our subsequent experiments and analyses. Figure 3 shows the range of agreements. We have explored using the median and mode statistics as well, but found that the correlation was essentially the same. We therefore choose the mean because it produces values on a continuum, making them more directly comparable to the classifier probabilities we will discuss later.

This analysis seems to confirm the widely-held belief that



**Figure 2: Distribution of readability scores made by 120 human annotators on code snippets taken from several open source projects (see Figure 8).**

people agree significantly on what readable code looks like, but not to an overwhelming extent, due perhaps to individual preferences. 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.

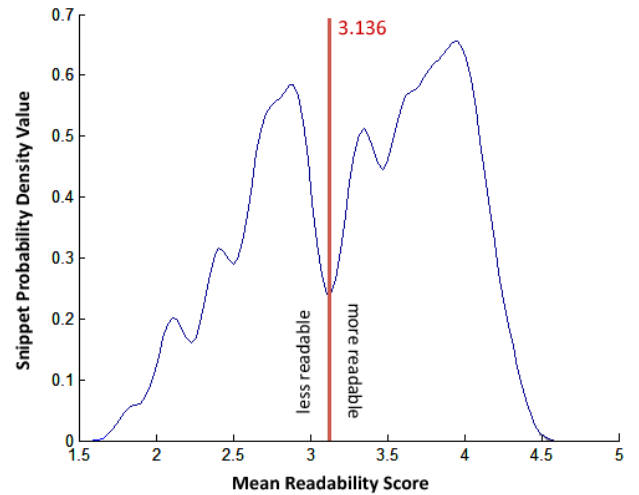
### 3. READABILITY MODEL

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.

#### 3.1 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 4 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 4 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



**Figure 3: Distribution of the average readability scores across all the snippets. The resulting 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.**

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* [33]. 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 3. 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 [21] 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 [27]. This 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.

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

Figure 4: The set of features considered by our metric.

### 3.2 Model Performance

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 [8]. 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 on the set of snippets with randomly generated score labels. 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), 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 rel-

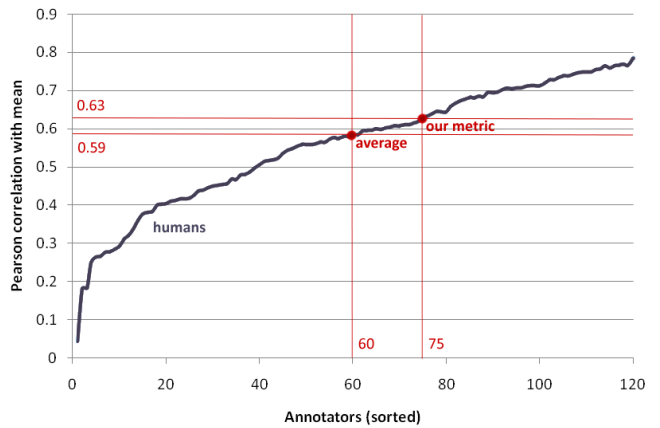


Figure 5: Annotator agreement with a model obtained by averaging the scores of 100 annotators with the addition of our metric.

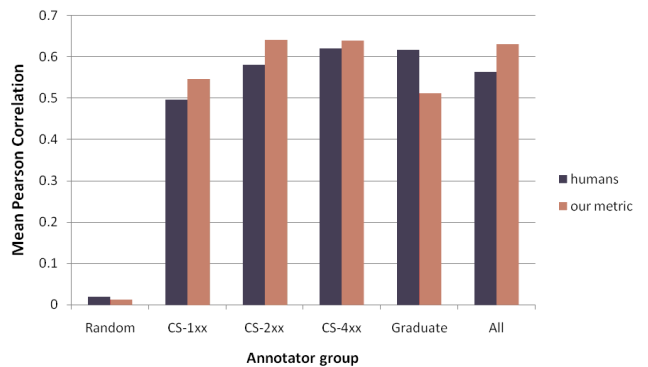


Figure 6: Annotator agreement by experience group.

ative ones. When we compare the output of the Bayesian classifier to the average human score model it was trained against, we obtain a Pearson correlation of 0.63. As shown in Figure 5, that level of agreement is better than what the average human in our study produced. 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 because of its run-time efficiency.

We also repeated the experiment separately with each annotator experience group (e.g., 100-level CS students, 200-level CS students). Figure 6 shows the mean Pearson correlations. The dark blue 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, 400-level CS students agree with each other more often (Pearson correlation over 0.6) than do 100-level CS students (correlation under 0.5). The light red bar on the right indicates the correlation between our metric (trained on the annotator judgments for that group) and the average of all annotators

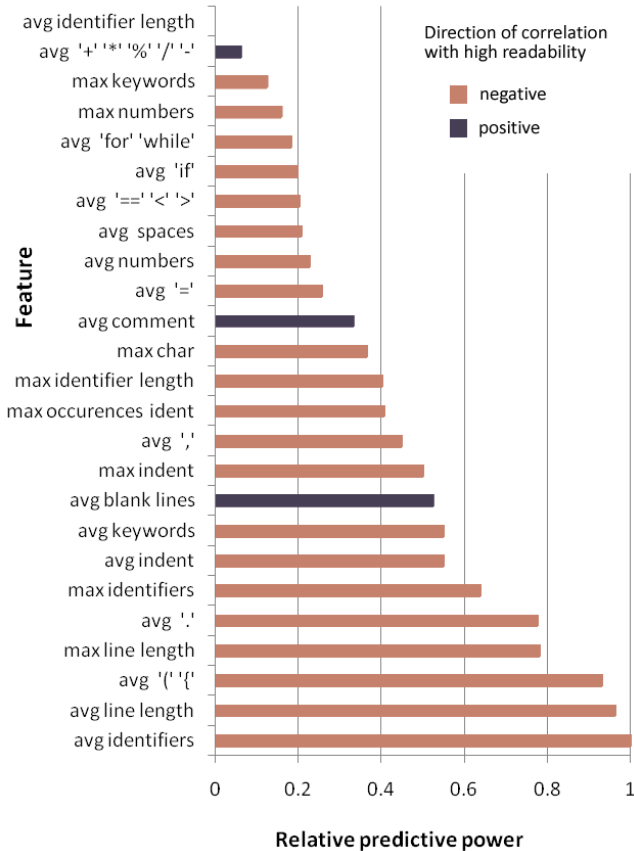


Figure 7: Relative power of features as determined by a singleton (one-feature-at-a-time) analysis. The direction of correlation for each is also indicated.

in the group. Two interesting observations arise. First, for all groups except graduate students, our automatic metric agrees with the human average more closely than the humans agree. We suspect that the difference with respect to graduates may be a reflection of the more diverse background of the graduate student population, their more sophisticated opinions, or some other external factor. Second, we see a gradual trend toward increased agreement with experience.

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 7 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 of itself, a very predictive factor; neither are `if` constructs, loops, or comparison operators. Section 5 includes a discussion of some of the possible implications of this result.

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 phe-

Project Name	KLOC	Maturity	Description
JasperReports 2.04	269	6	Dynamic content
Hibernate* 2.1.8	189	6	Database
jFreeChart* 1.0.9	181	5	Data rep.
FreeCol* 0.7.3	167	3	Game
jEdit* 4.2	140	5	Text editor
Gantt Project 3.0	130	5	Scheduling
soapUI 2.0.1	98	6	Web services
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 dev.
jMencode 0.64	7	3	Video encoding

Figure 8: Benchmark programs used in our experiments. The “Maturity” column indicates a self-reported *SourceForge* project status. \*Used as a snippet source.

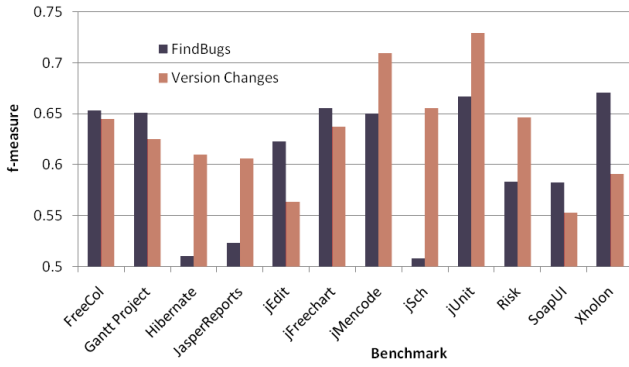
nomena. As a simple example, if there is exactly one space between every two words then a feature that counts words and a feature that counts spaces will capture essentially the same information and leaving one of them out is unlikely to decrease accuracy. A principle component analysis (PCA) indicates that 98% of the total variability can be explained by 6 principle components, thus implying that feature overlap is significant.

#### 4. 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 using a fixed classifier. In this section we use that tool to investigate whether our model of readability compares favorably with external conventional metrics of software quality. Specifically, we first look for a correlation between readability and FindBugs, a popular static bug-finding tool [22]. Second, we look for a similar correlation with changes to code between versions of several large open source projects. We chose FindBugs defects and version changes related to code churn in part because they can be measured objectively. Finally, we look for trends in code readability across those projects.

The set of open source Java programs we have employed as benchmarks can be found in Figure 8. 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 on a machine with a 2GHz processor and disk with a maximum 150 MBytes/sec transfer rate.



**Figure 9: f-measure for using readability to predict functions with FindBugs defect reports and functions which change between releases.**

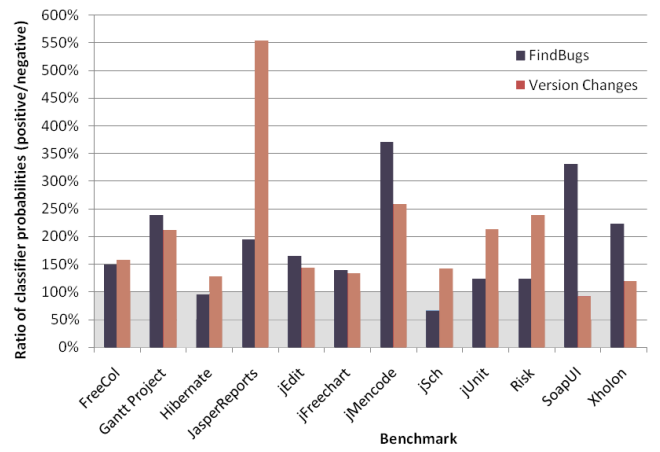
## 4.1 Readability Correlations

Our first experiment attempts to correlate 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. We normalized function set sizes to avoid bias between programs for which more or fewer defects were reported. 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.”

Our second experiment relates future code churn to 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 “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.

Figure 9 summarizes the results of these two experiments. Guessing randomly yields an f-measure of 0.5 and serves as a baseline, while 1.0 represents a perfect upper bound. The average f-measure over 11 of our benchmarks for the FindBugs correlation is 0.61. The average f-measure for version changes over all 12 of our benchmarks is 0.63. 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.

A second important consideration is the *magnitude* of the difference. We claim that classifier probabilities (i.e. continuous output v.s. discrete classifications) is useful in evaluating readability. Figure 10 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., >



**Figure 10: Mean ratio of the classifier probabilities (predicting ‘less readable’) assigned to functions that contained a FindBugs defect or that will change in the next version to those that were not. For example, Risk 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.**

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.

For both of these 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 0.7 in some cases. Again, our goal is not a perfect correlation with version changes and code churn. These moderate correlations do, however, imply a connection between code readability, as described by our model, and defects and upcoming code changes.

## 4.2 Software Lifecycle

To further investigate the relation of our readability metric to external factors, we investigate changes over long periods of time. Figure 11 shows how readability 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.

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 crosscutting features. Consider jUnit, which has recently adopted a “completely different API . . . [that] depends on new features of Java 5.0 (annotations, static import. . .)” [15]. We thus conducted an additional experiment to measure readability against maturity and stability.

Figure 12 plots project readability against project maturity, as self-reported by developers. The data shows a noisy upward trend implying that projects that reach maturity tend to be more readable. The results of these two experi-

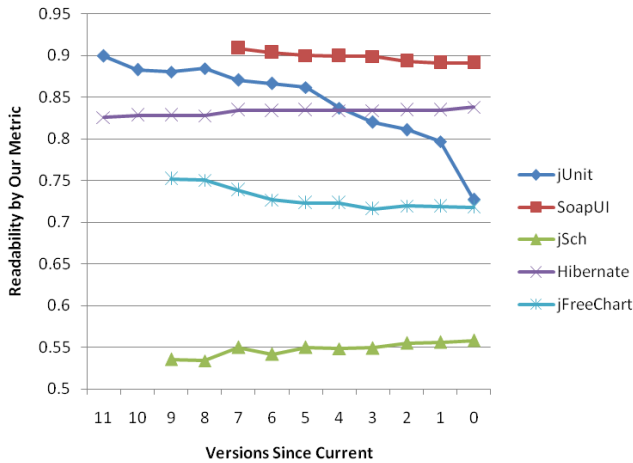


Figure 11: Average readability metric of all functions in a project as a function of project lifetime. Note that over time, the readability of some projects tends to decrease, while it gradually increases in others.

ments would seem *not* to support the Fred Brooks argument that, “all repairs tend to destroy the structure, to increase the entropy and disorder of the system . . . as time passes, the system becomes less and less well ordered” [14] for the readability component of “order”. While Brooks was not speaking specifically of readability, a lack readability can be a strong manifestation of disorder.

## 5. 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.

To start, we found that the length of identifier names constitutes almost no influence on readability (0% relative predictive power). Recently there has been 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, may not be as advisable as previously thought [39]. While descriptive identifiers certainly *can* improve readability, perhaps some additional attention should be paid to the fact that they may also reduce it; our study indicates the net gain may be near zero.

For example, forcing readers to process long names, where shorter ones would suffice, may negatively impact readability. Furthermore, identifier names are not always an appropriate place to encode documentation. There are many cases where it would be more appropriate to use comments, possibly associated with variable or field declarations, to explain program behavior. Long identifiers may be confusing, or even misleading. We believe that in many cases sophisti-

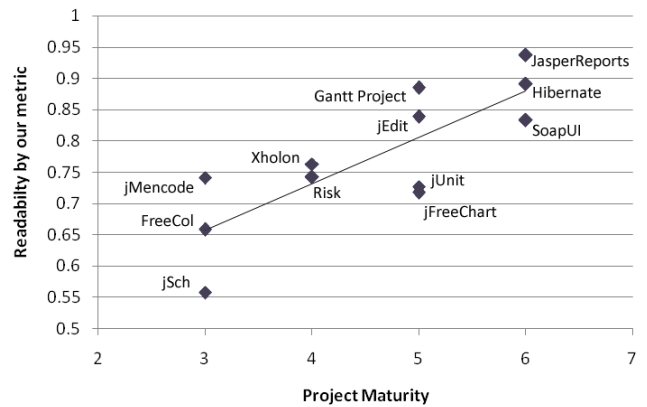


Figure 12: Average readability metric of all functions in a project as a function of self-reported project maturity with best fit linear trend line. Note that projects of greater maturity tend to exhibit greater readability.

cated integrated development environments (IDEs) and specialized static analysis tools designed to aid in software inspections (e.g., [3]), may constitute a better approach to the goal of enhancing program understanding.

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 only moderately well-correlated with 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 [43, 32]. 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 conclusion, that readability does not appear to be negatively impacted by repeated characters or words, runs counter to the common perception that operator overloading is necessarily confusing. Finally, our data suggests that languages should add additional keywords if it means that programs can be written with fewer new identifiers.

As we 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 means for conducting such experiments.

## 6. RELATED WORK

Previously, we identified several of the many automated readability metrics that are in use today for natural language [12, 18, 24, 31]. While we have not found analogous metrics targeted at source code (as presented in this paper), some metrics do exist outside the realm of traditional language. For example, utility has been claimed for a readability metric for computer generated math [30], for the layout of treemaps [4], and for hypertext [20].

Perhaps the bulk of work in the area of source code readability today is based on coding standards (e.g., [2, 6, 41]). These conventions are primarily intended to facilitate collaboration by maintaining uniformity between code written by different developers. Style checkers such as *PMD* [9] and *The Java Coding Standard Checker* are employed as a means to automatically enforce these standards.

We also note that machine learning has, in the past, been used for defect prediction, typically by training on data from source code repositories (e.g., [7, 17, 23, 25]). We believe that machine learning has substantial advantages over traditional statistics and that much room yet exists for the exploitation of such techniques in the domains of Software Engineering and Programming Languages.

## 7. 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 [19]. 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 3.2 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.

## 8. CONCLUSION

It is important to note that the metric described in this paper is not intended as the final or universal model of readability. Rather, we have shown how to produce *a* metric for software readability from the judgments of human annotators, relevant specifically to those annotators. In fact, 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.

## 9. REFERENCES

- [1] K. Aggarwal, Y. Singh, and J. K. Chhabra. An integrated measure of software maintainability. *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pages 235–241, September 2002.
- [2] S. Ambler. Java coding standards. *Softw. Dev.*, 5(8):67–71, 1997.
- [3] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. *WISE '01: Proceeding of the first workshop on inspection in software engineering*, July 2001.
- [4] B. B. Bederson, B. Shneiderman, and M. Wattenberg. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Trans. Graph.*, 21(4):833–854, 2002.
- [5] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.
- [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*. Specialized Systems Consultants, Inc., Seattle, Washington, June 1990.
- [7] T. J. Cheatham, J. P. Yoo, and N. J. Wahl. Software testing: a machine learning experiment. In *CSC '95: Proceedings of the 1995 ACM 23rd annual conference on Computer science*, pages 135–141, 1995.
- [8] T. Y. Chen, F.-C. Kuo, and R. Merkel. On the statistical properties of the f-measure. In *QSIC'04: Fourth International Conference on Quality Software*, pages 146–153, 2004.
- [9] T. Copeland. *PMD Applied*. Centennial Books, Alexandria, VA, USA, 2005.

- [10] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976.
- [11] J. L. Elshoff and M. Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25(8):512–521, 1982.
- [12] R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, 1948.
- [13] J. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.
- [14] J. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.
- [15] A. Goncalves. Get acquainted with the new advanced features of junit 4. *DevX*, <http://www.devx.com/Java/Article/31983>, 2006.
- [16] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.
- [17] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.
- [18] R. Gunning. *The Technique of Clear Writing*. McGraw-Hill International Book Co, New York, 1952.
- [19] N. J. Haneef. Software documentation and readability: a proposed process improvement. *SIGSOFT Softw. Eng. Notes*, 23(3):75–77, 1998.
- [20] A. E. Hatzimanikatis, C. T. Tsalidis, and D. Christodoulakis. Measuring the readability and maintainability of hyperdocuments. *Journal of Software Maintenance*, 7(2):77–90, 1995.
- [21] G. Holmes, A. Donkin, and I. Witten. Weka: A machine learning workbench. *Proceedings of the Second Australia and New Zealand Conference on Intelligent Information Systems*, 1994.
- [22] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [23] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, page 364, Washington, DC, USA, 1996. IEEE Computer Society.
- [24] J. P. Kinciad and E. A. Smith. Derivation and validation of the automated readability index for use with technical materials. *Human Factors*, 12:457–464, 1970.
- [25] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125, 2006.
- [26] J. C. Knight and E. A. Myers. Phased inspections and their implementation. *SIGSOFT Softw. Eng. Notes*, 16(3):29–35, 1991.
- [27] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.
- [28] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:44–53, 1932.
- [29] J. Lionel E. Deimel. The uses of program reading. *SIGCSE Bull.*, 17(2):5–14, 1985.
- [30] S. MacHaffie, R. McLeod, B. Roberts, P. Todd, and L. Anderson. A readability metric for computer-generated mathematics. Technical report, Saltire Software, <http://www.saltire.com/equation.html>, retrieved 2007.
- [31] G. H. McLaughlin. Smog grading – a new readability. *Journal of Reading*, May 1969.
- [32] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867, 1983.
- [33] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [34] 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*, pages 284–292, 2005.
- [35] C. V. Ramamoorthy and W.-T. Tsai. Advances in software engineering. *Computer*, 29(10):47–58, 1996.
- [36] D. R. Raymond. Reading source code. In *CASCON '91: Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 3–16. IBM Press, 1991.
- [37] P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. *Empirical Software Engineering, 2005. 2005 International Symposium on*, November 2005.
- [38] S. Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.
- [39] C. Simonyi. Hungarian notation. *MSDN Library*, November 1999.
- [40] S. E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9(4), 2004.
- [41] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley Professional, 2004.
- [42] T. Tenny. Program readability: Procedures versus comments. *IEEE Trans. Softw. Eng.*, 14(9):1271–1279, 1988.
- [43] A. Watters, G. van Rossum, and J. C. Ahlstrom. *Internet Programming with Python*. MIS Press/Henry Holt publishers, New York, 1996.
- [44] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988.