Improving Code Readability Models with Textual Features

Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, Rocco Oliveto

Software maintenance accounts for

70%

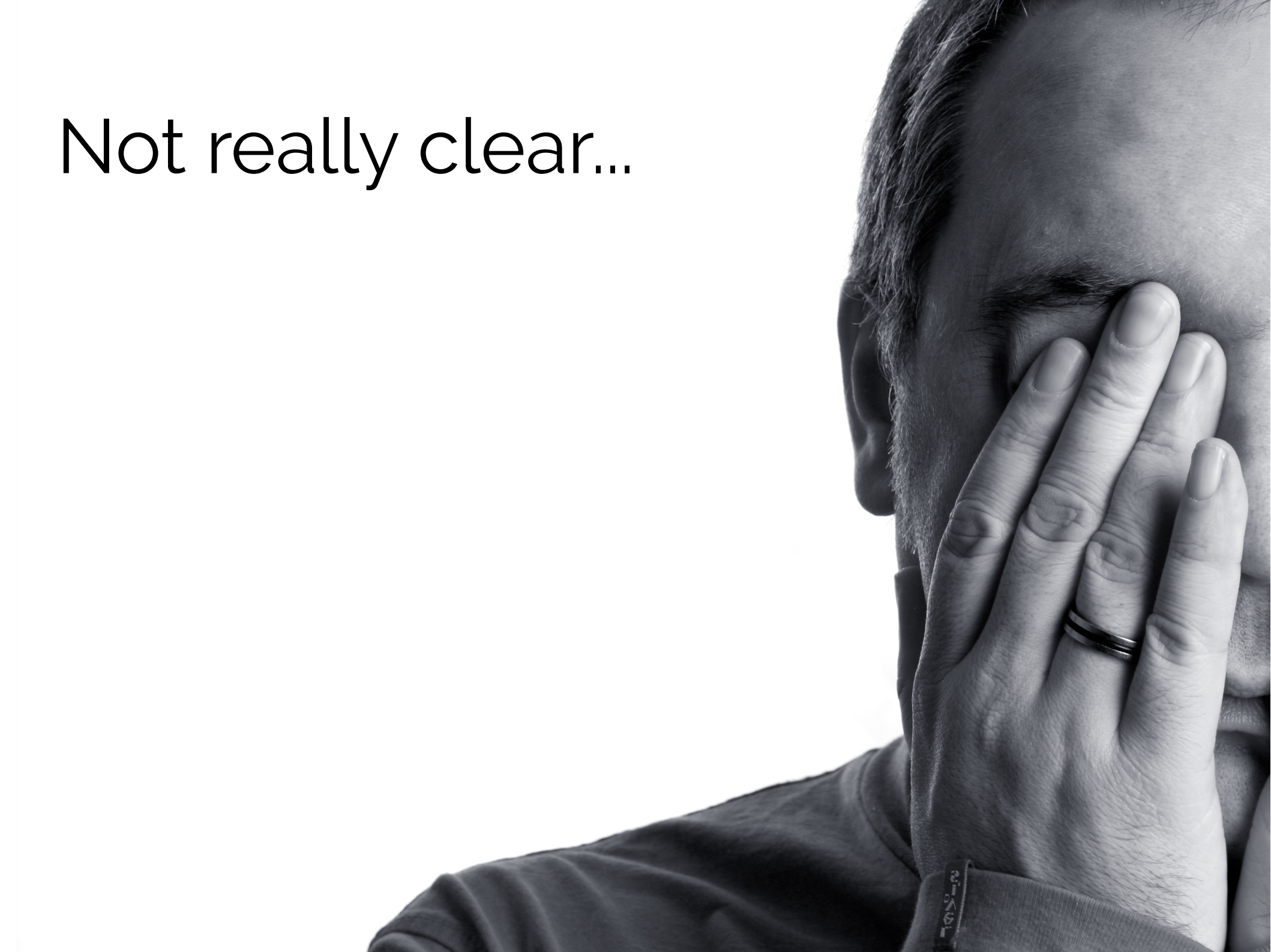
of the costs of a project


LISTEN
UNDERSTAND
ACT

```
for (int i=1;i<=100;i++) {
    int a=((528>>i%15-1)&1)*4;
    int b=((-2128340926>>(i%15)*2)&3)*4;
    System.out.println("FizzBuzz".substring(a,b)+(a==b?i:""));
}
```

What does it do?

Not really clear...

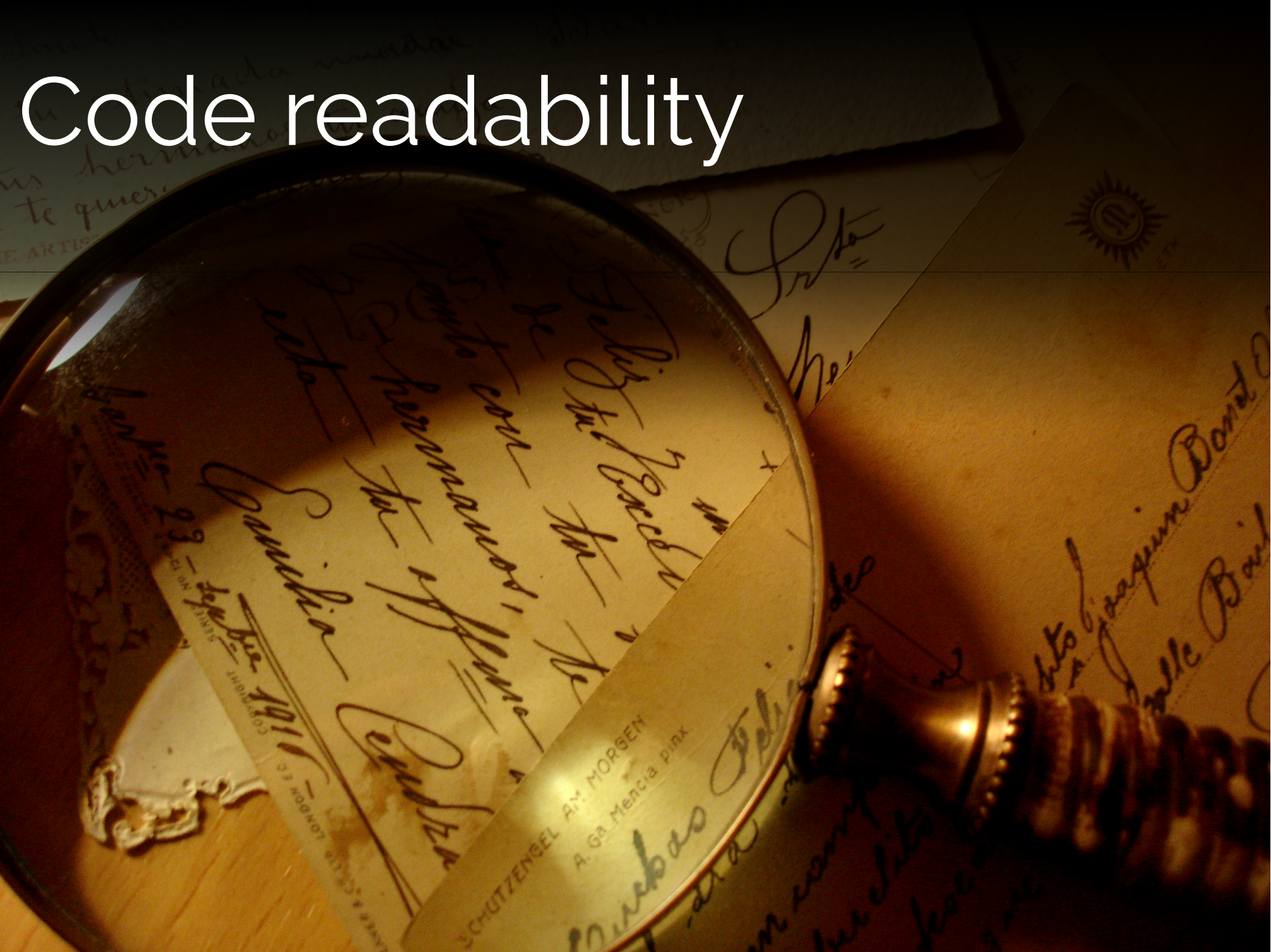


What about this one?

```
for (int i=1;i<=100;i++) {
   String fizzBuzz = “”;
   if (i % 3 == 0)
      fizzBuzz += “Fizz”;

   if (i % 5 == 0)
      fizzBuzz += “Buzz”;

   if (fizzBuzz.isEmpty())
      fizzBuzz += i;

   System.out.println(fizzBuzz);
}
```

Why is it easier to understand?

Code readability



TSE SPECIAL ISSUE ON THE ISSTA 2008 BEST PAPERS 1

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 University of Virginia, Charlottesville, VA 22904.*
E-mail: {jbuse, 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.

A Simpler Model of Software Readability

Daryl Posnett
University of California, Davis
Davis, CA
dposnett@ucdavis.edu

Abram Hindle
University of California, Davis
Davis, CA
ah@softwareprocess.es

Prem Devanbu
University of California, Davis
Davis, CA
devanbu@ucdavis.edu

ABSTRACT

Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into development tool-chains, and thus continually inform developers about the readability level of the code. Unfortunately, readability is a subjective code property, and not amenable to direct automated measurement. In a recently published study, Buse *et al.* asked 100 participants to rate code snippets by readability, yielding arguably reliable mean readability scores of each snippet; they then built a fairly complex predictive model for these mean scores using a large, diverse set of directly measurable source code properties. We build on this work: we present a simple, intuitive theory of readability, based on size and code entropy, and show how this theory leads to a much sparser, yet statistically significant, model of the mean readability scores produced in Buse's studies. Our model uses well-known size metrics and Halstead metrics, which are easily extracted using a variety of tools. We argue that this approach provides a more theoretically well-founded, practically usable, approach to readability measurement.

Categories and Subject Descriptors

H.1 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Human Factors, Theory, Measurement

Keywords

Readability, Halstead, Entropy, Replication

MSR '11, 21-MAY-2011, Waikiki, Honolulu , USA

1. INTRODUCTION

Readability of code is of central concern for developers [1, 18, 19, 24]. Code that is readable is often considered more maintainable; code that is more readable today is presumed to remain easier to read, comprehend, and maintain at a later date.

We conceive of readability as a subjective impression that programmers have of the difficulty of code, as they try to understand it. The relationship between readability and understanding is analogous to syntactic and semantic analysis, readability is the syntactic aspect while understandability is the semantic aspect. In essence, readability is a perceived barrier to understanding that the programmer feels the need to overcome before working with a body of code: the more readable it is, the lower the barrier.

There is much previous work about readability [7, 15, 34, 35, 13, 3]. One major issue with studies of readability is the difficulty of experimentally reifying what is essentially a subjective perception. Measures of subjective perception are both difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, S, accompanied by mean subjective readability scores: $\mathcal{O}(s),\ s \in S$. Buse *et al.* then gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but even so, were able to predict $\mathcal{O}(s)$ to some degree. This was an important contribution since it opens up the possibility of automatic tools that can provide readability scores as feedback to developers for every commit they make; this continuous feedback might potentially improve readability, and thus maintainability, of code over time.

In this paper we improve upon Buse *et al.*'s model, yielding a model that is *simpler, better performing* and *theoretically well-founded* in both classical software engineering and basic information theory. In particular we argue that fundamental aspects of readability have been measured since the 1970s by Halstead's software science metrics [21]. We show that the Halstead metrics can be used to improve upon the

A General Software Readability Model

Jonathan Dorn
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jad5ju@virginia.edu

***Abstract*—We present a generalizable formal model of software readability based on a human study of 5000 participants. Readability is fundamental to maintenance, but remains poorly understood. Previous models focused on symbol counts of small code snippets. By contrast, we approach code as read on screens by humans and propose to analyze visual, spatial and linguistic features, including structural patterns, sizes of code blocks, and verbal identifier content. We construct a readability metric based on these notions and show that it agrees with human judgments as well as they agree with each other and better than previous work. We identify universal features of readability and language- or experience-specific ones. Our metric also correlates with an external notion of defect density. We address multiple programming languages and different length samples, and evaluate using an order of magnitude more participants than previous work, all suggesting our model is more likely to generalize.**

I. Introduction

Modern software developers spend more time maintaining and evolving existing software than writing new code [1], [2], [3]. Software *readability*, a fundamental notion related to the comprehension of text, is critical to software maintenance: reading code is a necessary first step toward maintaining it.

Much research, both recent and established, has argued that readability plays a large role in software maintenance. A well-known example is Knuth, who viewed readability as essential to his notion of Literate Programming [4]. He argued that a program should be viewed as "a piece of literature, addressed to human beings" and that a readable program is "more robust, more portable, [and] more easily maintained". Haneef argued in favor of a development group dedicated to readability and documentation: "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [5]. Knight and Myers argued that a source-level check for readability improves portability, maintainability and reusability and should thus be a first-class phase of software inspection [6]. Basili *et al.* showed that inspections guided by reading techniques are better at revealing defects [7]. An entire development phase aimed at improving readability was proposed by Elshoff and Marcotty, who observed that many commercial programs were unnecessarily difficult to read [8]. More recently, a 2012 survey of over 100 developers and managers at Microsoft by Buse and Zimmermann found that 90% of responders desire readability as a software analytic feature, placing it among the top three in their survey [9].

Readability metrics are well-established in the domain of non-software natural language. Metrics such as the Automated Readability Index [10] and Flesch-Kincaid Grade Level [11] are commonly used in commercial software and policies. All are based on a few simple measurements, such as the lengths of words and sentences. For example, Flesch-Kincaid is integrated into popular editors such as Microsoft Word and has become a government standard, with the US Department of Defense requiring internal and external documents to have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). In the domain of software, formal metrics for readability are well-established in particular domains such as hypertext [12].

By contrast, general descriptive models of overall software readability are relatively recent, first proposed by Buse *et al.* [13] and refined by Posnett *et al.* [14]. Such models are not coding standards (cf. [15]) but are based on combinations of surface-level syntactic features such as operator counts or line lengths, aim to agree with human judgments, and have been found to correlate with external notions of software quality [16]. Such software readability models do not attempt to describe programmatic complexity (cf. [17]), which derives from system requirements and algorithms, but instead focus on readability as a controllable accidental complexity [18].

Despite the advantages of a formal notion of software readability, previous readability metrics do not adequately *generalize*. They are based on small (typically 7-line) snippets of code from a single programming language, are tied to shallow surface features that do not account for visual presentation or linguistic meaning, and derive from the judgments of a relatively small number of students [13], [14]. We propose a readability model that addresses all of these concerns while remaining lightweight and applicable.

Intuitively, the effectiveness of syntax highlighting suggests that visual or geometric formatting significantly impacts code readability. Similarly, the prevalence of variable naming standards (e.g., underscores, camel case, Hungarian notation) suggests that meaningful linguistic information is captured by identifiers. We thus propose the first incorporation of geometric, pattern-based and linguistic aspects and features into an automated readability metric. For example, code in which the "=" operators in a sequence of assignment statements "line up" vertically on the screen may be viewed as more readable, as may code in which identifiers contain English synonyms or code in which comments form a colored rectangular block. We propose to incorporate such features into our model of readability.

Code readability prediction

TSE SPECIAL ISSUE ON THE ISSTA 2008 BEST PAPERS 1

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 University 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.

A Simpler Model of Software Readability

Daryl Posnett
University of California, Davis
Davis, CA
dpposnett@ucdavis.edu

Abram Hindle
University of California, Davis
Davis, CA
ah@softwareprocess.es

Prem Devanbu
University of California, Davis
Davis, CA
devanbu@ucdavis.edu

ABSTRACT

Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into development tool-chains, and thus continually inform developers about the readability level of the code. Unfortunately, readability is a subjective code property, and not amenable to direct automated measurement. In a recently published study, Buse *et al.* asked 100 participants to rate code snippets by readability, yielding arguably reliable mean readability scores of each snippet; they then built a fairly complex predictive model for these mean scores using a large, diverse set of directly measurable source code properties. We build on this work: we present a simple, intuitive theory of readability, based on size and code entropy, and show how this theory leads to a much sparser, yet statistically significant, model of the mean readability scores produced in Buse's studies. Our model uses well-known size metrics and Halstead metrics, which are easily extracted using a variety of tools. We argue that this approach provides a more theoretically well-founded, practically usable, approach to readability measurement.

Categories and Subject Descriptors

H.1 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Human Factors, Theory, Measurement

Keywords

Readability, Halstead, Entropy, Replication

MSR '11, 21-MAY-2011, Waikiki, Honolulu , USA

1. INTRODUCTION

Readability of code is of central concern for developers [1, 18, 19, 24]. Code that is readable is often considered more maintainable; code that is more readable today is presumed to remain easier to read, comprehend, and maintain at a later date.

We conceive of readability as a subjective impression that programmers have of the difficulty of code, as they try to understand it. The relationship between readability and understanding is analogous to syntactic and semantic analysis, readability is the syntactic aspect while understandability is the semantic aspect. In essence, readability is a perceived barrier to understanding that the programmer feels the need to overcome before working with a body of code: the more readable it is, the lower the barrier.

There is much previous work about readability [7, 15, 34, 35, 13, 3]. One major issue with studies of readability is the difficulty of experimentally reifying what is essentially a subjective perception. Measures of subjective perception are both difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, $\mathcal{S}$, accompanied by mean subjective readability scores: $\mathcal{O}(s)$, $s \in \mathcal{S}$. Buse *et al.* then gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but even so, were able to predict $\mathcal{O}(s)$ to some degree. This was an important contribution since it opens up the possibility of automatic tools that can provide readability scores as feedback to developers for every commit they make; this continuous feedback might potentially improve readability, and thus maintainability, of code over time.

In this paper we improve upon Buse *et al.*'s model, yielding a model that is *simpler, better performing* and *theoretically well-founded* in both classical software engineering and basic information theory. In particular we argue that fundamental aspects of readability have been measured since the 1970s by Halstead's software science metrics [21]. We show that the Halstead metrics can be used to improve upon the

A General Software Readability Model

Jonathan Dorn
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jad5ju@virginia.edu

Abstract—**We present a generalizable formal model of software readability based on a human study of 5000 participants. Readability is fundamental to maintenance, but remains poorly understood. Previous models focused on symbol counts of small code snippets. By contrast, we approach code as read on screens by humans and propose to analyze visual, spatial and linguistic features, including structural patterns, sizes of code blocks, and verbal identifier content. We construct a readability metric based on these notions and show that it agrees with human judgments as well as they agree with each other and better than previous work. We identify universal features of readability and language- or experience-specific ones. Our metric also correlates with an external notion of defect density. We address multiple programming languages and different length samples, and evaluate using an order of magnitude more participants than previous work, all suggesting our model is more likely to generalize.**

I. Introduction

Modern software developers spend more time maintaining and evolving existing software than writing new code [1], [2], [3]. Software *readability*, a fundamental notion related to the comprehension of text, is critical to software maintenance: reading code is a necessary first step toward maintaining it.

Much research, both recent and established, has argued that readability plays a large role in software maintenance. A well-known example is Knuth, who viewed readability as essential to his notion of Literate Programming [4]. He argued that a program should be viewed as "a piece of literature, addressed to human beings" and that a readable program is "more robust, more portable, [and] more easily maintained". Haneef argued in favor of a development group dedicated to readability and documentation: "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [5]. Knight and Myers argued that a source-level check for readability improves portability, maintainability and reusability and should thus be a first-class phase of software inspection [6]. Basili *et al.* showed that inspections guided by reading techniques are better at revealing defects [7]. An entire development phase aimed at improving readability was proposed by Elshoff and Marcotty, who observed that many commercial programs were unnecessarily difficult to read [8]. More recently, a 2012 survey of over 100 developers and managers at Microsoft by Buse and Zimmermann found that 90% of responders desire readability as a software analytic feature, placing it among the top three in their survey [9].

Readability metrics are well-established in the domain of non-software natural language. Metrics such as the Automated Readability Index [10] and Flesch-Kincaid Grade Level [11] are commonly used in commercial software and policies. All are based on a few simple measurements, such as the lengths of words and sentences. For example, Flesch-Kincaid is integrated into popular editors such as Microsoft Word and has become a government standard, with the US Department of Defense requiring internal and external documents to have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). In the domain of software, formal metrics for readability are well-established in particular domains such as hypertext [12].

By contrast, general descriptive models of overall software readability are relatively recent, first proposed by Buse *et al.* [13] and refined by Posnett *et al.* [14]. Such models are not coding standards (cf. [15]) but are based on combinations of surface-level syntactic features such as operator counts or line lengths, aim to agree with human judgments, and have been found to correlate with external notions of software quality [16]. Such software readability models do not attempt to describe programmatic complexity (cf. [17]), which derives from system requirements and algorithms, but instead focus on readability as a controllable accidental complexity [18].

Despite the advantages of a formal notion of software readability, previous readability metrics do not adequately *generalize*. They are based on small (typically 7-line) snippets of code from a single programming language, are tied to shallow surface features that do not account for visual presentation or linguistic meaning, and derive from the judgments of a relatively small number of students [13], [14]. We propose a readability model that addresses all of these concerns while remaining lightweight and applicable.

Intuitively, the effectiveness of syntax highlighting suggests that visual or geometric formatting significantly impacts code readability. Similarly, the prevalence of variable naming standards (e.g., underscores, camel case, Hungarian notation) suggests that meaningful linguistic information is captured by identifiers. We thus propose the first incorporation of geometric, pattern-based and linguistic aspects and features into an automated readability metric. For example, code in which the "=" operators in a sequence of assignment statements "line up" vertically on the screen may be viewed as more readable, as may code in which identifiers contain English synonyms or code in which comments form a colored rectangular block. We propose to incorporate such features into our model of readability.

Code readability prediction

TSE SPECIAL ISSUE ON THE ISSTA 2008 BEST PAPERS 1

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 University 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.

A Simpler Model of Software Readability

Daryl Posnett
University of California, Davis
Davis, CA
dpposnett@ucdavis.edu

Abram Hindle
University of California, Davis
Davis, CA
ah@softwareprocess.es

Prem Devanbu
University of California, Davis
Davis, CA
devanbu@ucdavis.edu

ABSTRACT

Software readability is a property that influences how easily a given piece of code can be read and understood. Since readability can affect maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into development tool-chains, and thus continually inform developers about the readability level of the code. Unfortunately, readability is a subjective code property, and not amenable to direct automated measurement. In a recently published study, Buse *et al.* asked 100 participants to rate code snippets by readability, yielding arguably reliable mean readability scores of each snippet; they then built a fairly complex predictive model for these mean scores using a large, diverse set of directly measurable source code properties. We build on this work: we present a simple, intuitive theory of readability, based on size and code entropy, and show how this theory leads to a much sparser, yet statistically significant, model of the mean readability scores produced in Buse's studies. Our model uses well-known size metrics and Halstead metrics, which are easily extracted using a variety of tools. We argue that this approach provides a more theoretically well-founded, practically usable, approach to readability measurement.

Categories and Subject Descriptors

H.1 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Human Factors, Theory, Measurement

Keywords

Readability, Halstead, Entropy, Replication

MSR '11, 21-MAY-2011, Waikiki, Honolulu , USA

1. INTRODUCTION

Readability of code is of central concern for developers [1, 18, 19, 24]. Code that is readable is often considered more maintainable; code that is more readable today is presumed to remain easier to read, comprehend, and maintain at a later date.

We conceive of readability as a subjective impression that programmers have of the difficulty of code, as they try to understand it. The relationship between readability and understanding is analogous to syntactic and semantic analysis, readability is the syntactic aspect while understandability is the semantic aspect. In essence, readability is a perceived barrier to understanding that the programmer feels the need to overcome before working with a body of code: the more readable it is, the lower the barrier.

There is much previous work about readability [7, 15, 34, 35, 13, 3]. One major issue with studies of readability is the difficulty of experimentally reifying what is essentially a subjective perception. Measures of subjective perception are both difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, S, accompanied by mean subjective readability scores: $\mathcal{O}(s), s \in S$. Buse *et al.* then gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but even so, were able to predict $\mathcal{O}(s)$ to some degree. This was an important contribution since it opens up the possibility of automatic tools that can provide readability scores as feedback to developers for every commit they make; this continuous feedback might potentially improve readability, and thus maintainability, of code over time.

In this paper we improve upon Buse *et al.*'s model, yielding a model that is *simpler, better performing* and *theoretically well-founded* in both classical software engineering and basic information theory. In particular we argue that fundamental aspects of readability have been measured since the 1970s by Halstead's software science metrics [21]. We show that the Halstead metrics can be used to improve upon the Readability Index [10] and Flesch-Kincaid Grade Level [11] are commonly used in commercial software and policies. All are based on a few simple measurements, such as the lengths of words and sentences. For example, Flesch-Kincaid is integrated into popular editors such as Microsoft Word and has become a government standard, with the US Department of Defense requiring internal and external documents to have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). In the domain of software, formal metrics for readability are well-established in particular domains such as hypertext [12].

By contrast, general descriptive models of overall software readability are relatively recent, first proposed by Buse *et al.* [13] and refined by Posnett *et al.* [14]. Such models are not coding standards (cf. [15]) but are based on combinations of surface-level syntactic features such as operator counts or line lengths, aim to agree with human judgments, and have been found to correlate with external notions of software quality [16]. Such software readability models do not attempt to describe programmatic complexity (cf. [17]), which derives from system requirements and algorithms, but instead focus on readability as a controllable accidental complexity [18].

Despite the advantages of a formal notion of software readability, previous readability metrics do not adequately *generalize*. They are based on small (typically 7-line) snippets of code from a single programming language, are tied to shallow surface features that do not account for visual presentation or linguistic meaning, and derive from the judgments of a relatively small number of students [13], [14]. We propose a readability model that addresses all of these concerns while remaining lightweight and applicable.

Intuitively, the effectiveness of syntax highlighting suggests that visual or geometric formatting significantly impacts code readability. Similarly, the prevalence of variable naming standards (e.g., underscores, camel case, Hungarian notation) suggests that meaningful linguistic information is captured by identifiers. We thus propose the first incorporation of geometric, pattern-based and linguistic aspects and features into an automated readability metric. For example, code in which the "=" operators in a sequence of assignment statements "line up" vertically on the screen may be viewed as more readable, as may code in which identifiers contain English synonyms or code in which comments form a colored rectangular block. We propose to incorporate such features into our model of readability.

A General Software Readability Model

Jonathan Dorn
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jad5ju@virginia.edu

***Abstract*—We present a generalizable formal model of software readability based on a human study of 5000 participants. Readability is fundamental to maintenance, but remains poorly understood. Previous models focused on symbol counts of small code snippets. By contrast, we approach code as read on screens by humans and propose to analyze visual, spatial and linguistic features, including structural patterns, sizes of code blocks, and verbal identifier content. We construct a readability metric based on these notions and show that it agrees with human judgments as well as they agree with each other and better than previous work. We identify universal features of readability and language- or experience-specific ones. Our metric also correlates with an external notion of defect density. We address multiple programming languages and different length samples, and evaluate using an order of magnitude more participants than previous work, all suggesting our model is more likely to generalize.**

I. Introduction

Modern software developers spend more time maintaining and evolving existing software than writing new code [1], [2], [3]. Software *readability*, a fundamental notion related to the comprehension of text, is critical to software maintenance: reading code is a necessary first step toward maintaining it.

Much research, both recent and established, has argued that readability plays a large role in software maintenance. A well-known example is Knuth, who viewed readability as essential to his notion of Literate Programming [4]. He argued that a program should be viewed as "a piece of literature, addressed to human beings" and that a readable program is "more robust, more portable, [and] more easily maintained". Haneef argued in favor of a development group dedicated to readability and documentation: "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [5]. Knight and Myers argued that a source-level check for readability improves portability, maintainability and reusability and should thus be a first-class phase of software inspection [6]. Basili *et al.* showed that inspections guided by reading techniques are better at revealing defects [7]. An entire development phase aimed at improving readability was proposed by Elshoff and Marcotty, who observed that many commercial programs were unnecessarily difficult to read [8]. More recently, a 2012 survey of over 100 developers and managers at Microsoft by Buse and Zimmermann found that 90% of responders desire readability as a software analytic feature, placing it among the top three in their survey [9].

Readability metrics are well-established in the domain of non-software natural language. Metrics such as the Automated

Code readability prediction

TSE SPECIAL ISSUE ON THE ISSTA 2008 BEST PAPERS 1

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 University 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.

A Simpler Model of Software Readability

Daryl Posnett
University of California, Davis
Davis, CA
dpposnett@ucdavis.edu

Abram Hindle
University of California, Davis
Davis, CA
ah@softwareprocess.es

Prem Devanbu
University of California, Davis
Davis, CA
devanbu@ucdavis.edu

ABSTRACT

Software readability is a property that influences how easily a given piece of code can be read and understood. Despite its direct impact on [illegible] maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into development tool-chains, and thus encourage writing more readable code. This paper details a study of the factors that seem to influence readability. We build on this work, we present a simple, intuitive theory of readability, based on size and code entropy, and show how this theory leads to a much sparser, yet statistically significant, model of the mean readability scores produced in Buse's studies. Our model uses only factors [illegible] and averages them. We also show that [illegible] correlate with more traditional metrics, which are easily extracted using a variety of tools. We argue that this approach provides a more theoretically well-founded, practically usable, approach to readability measurement.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Human Factors, Theory, Measurement

Keywords

Readability, Halstead, Entropy, Replication

MSR '11, 21-MAY-2011, Waikiki, Honolulu , USA

1. INTRODUCTION

Readability of code is of central concern for [illegible] more maintainable; code that is more readable today is presumed to remain easier to read, comprehend, and maintain at a later date.

We conceive of readability as a subjective impression that [illegible] the difficulty of code as there are for [illegible] subjective impression that [illegible] readability is the power the [illegible]

There is much previous work about readability [7, 15, 34, 35, 13, 3]. One major issue with studies of readability is the difficulty of experimentally verifying what is essentially a subjective perception. Measures of [illegible] are often difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, $\mathcal{S}$, accompanied by mean subjective readability scores: $\mathcal{O}(s)$, $s \in \mathcal{S}$. Buse *et al.* then gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but even so, were able to predict $\mathcal{O}(s)$ to some degree. This was an important contribution since it opens up the possibility of automatic tools that can provide readability scores as feedback to developers for every commit they make: this continuous feedback might potentially improve readability, and thus maintainability, of code over time.

In this paper we improve upon Buse *et al.*'s model, yielding a model that is *simpler, better performing* and *theoretically well-founded* in both classical software engineering and basic information theory. In particular we argue that fundamental aspects of readability have been measured since the 1970s by Halstead's software science metrics [21]. We show that the Halstead metrics can be used to improve upon the

A General Software Readability Model

Jonathan Dorn
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jad5ju@virginia.edu

***Abstract*—We present a generalizable formal model of software readability based on a human study of 5000 participants. Readability is fundamental to maintenance, but remains poorly understood. Previous models focused on symbol counts of small code snippets. By contrast, we approach code as read on screens by humans and propose to analyze visual, spatial and linguistic features, including structural patterns, sizes of code blocks, and verbal identifier content. We construct a readability metric based on these notions and show that it agrees with human judgments as well as they agree with each other and better than previous work. We identify universal features of readability and language-specific or experience-specific ones. Our metric also correlates with an external notion of defect density. We address multiple programming languages and different length samples, and evaluate using an order of magnitude more participants than previous work, all suggesting our model is more likely to generalize.**

I. Introduction

Modern software developers spend more time maintaining and evolving existing software than writing new code [1], [2], [3]. Software *readability*, a fundamental notion related to the comprehension of text, is critical to software maintenance: reading code is a necessary first step toward maintaining it.

Much research, both recent and established, has argued that readability plays a large role in software maintenance. A well-known example is Knuth, who viewed readability as essential to his notion of Literate Programming [4]. He argued that a program should be viewed as "a piece of literature, addressed to human beings" and that a readable program is "more robust, more portable, [and] more easily maintained". Haneef argued in favor of a development group dedicated to readability and documentation: "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [5]. Knight and Myers argued that a source-level check for readability improves portability, maintainability and reusability and should thus be a first-class phase of software inspection [6]. Basili *et al.* showed that inspections guided by reading techniques are better at revealing defects [7]. An entire development phase aimed at improving readability was proposed by Elshoff and Marcotty, who observed that many commercial programs were unnecessarily difficult to read [8]. More recently, a 2012 survey of over 100 developers and managers at Microsoft by Buse and Zimmermann found that 90% of responders desire readability as a software analytic feature, placing it among the top three in their survey [9].

Readability metrics are well-established in the domain of non-software natural language. Metrics such as the Automated Readability Index [10] and Flesch-Kincaid Grade Level [11] are commonly used in commercial software and policies. All are based on a few simple measurements, such as the lengths of words and sentences. For example, Flesch-Kincaid is integrated into popular editors such as Microsoft Word and has become a government standard, with the US Department of Defense requiring internal and external documents to have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). In the domain of software, formal metrics for readability are well-established in particular domains such as hypertext [12].

By contrast, general descriptive models of overall software readability are relatively recent, first proposed by Buse *et al.* [13] and refined by Posnett *et al.* [14]. Such models are not coding standards (cf. [15]) but are based on combinations of surface-level syntactic features such as operator counts or line lengths, aim to agree with human judgments, and have been found to correlate with external notions of software quality [16]. Such software readability models do not attempt to describe programmatic complexity (cf. [17]), which derives from system requirements and algorithms, but instead focus on readability as a controllable accidental complexity [18].

Despite the advantages of a formal notion of software readability, previous readability metrics do not adequately *generalize*. They are based on small (typically 7-line) snippets of code from a single programming language, are tied to shallow surface features that do not account for visual presentation or linguistic meaning, and derive from the judgments of a relatively small number of students [13], [14]. We propose a readability model that addresses all of these concerns while remaining lightweight and applicable.

Intuitively, the effectiveness of syntax highlighting suggests that visual or geometric formatting significantly impacts code readability. Similarly, the prevalence of variable naming standards (e.g., underscores, camel case, Hungarian notation) suggests that meaningful linguistic information is captured by identifiers. We thus propose the first incorporation of geometric, pattern-based and linguistic aspects and features into an automated readability metric. For example, code in which the "=" operators in a sequence of assignment statements "line up" vertically on the screen may be viewed as more readable, as may code in which identifiers contain English synonyms or code in which comments form a colored rectangular block. We propose to incorporate such features into our model of readability.

Structural features

Code readability prediction

TSE SPECIAL ISSUE ON THE ISSTA 2008 BEST PAPERS 1

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].

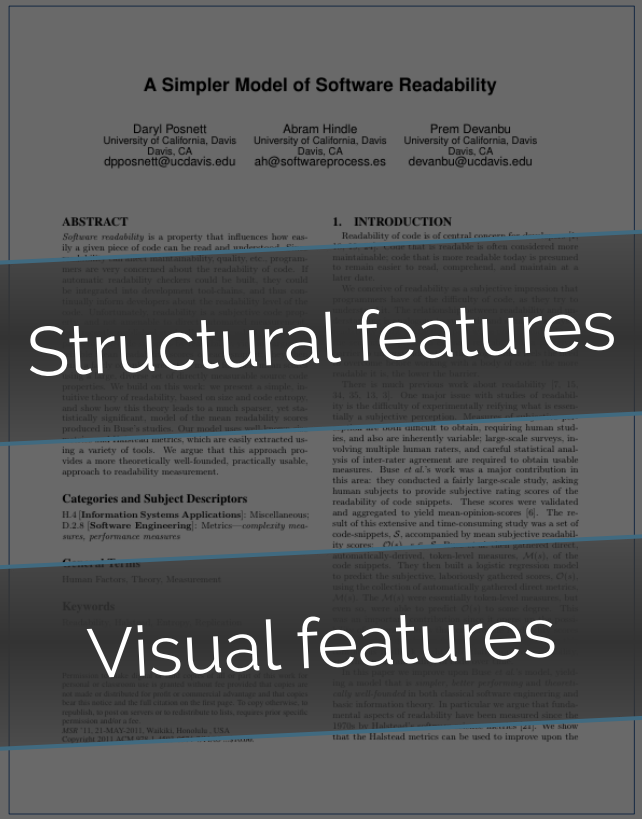
We hypothesize that programmers have some intuitive notion of this concept, and that program features such as indentation (e.g., as in Python [40]), choice of identifier names [34], and comments are likely to play a part. Dijkstra, for example, claimed that the readability of a program depends largely 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], to omap layout [3], and hypertext [17] have been found useful. We describe the first general readability metric for code.

• *Buse and Weimer are with the Department of Computer Science at The University 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.



A Simpler Model of Software Readability

Daryl Posnett
University of California, Davis
Davis, CA
dpposnett@ucdavis.edu

Abram Hindle
University of California, Davis
Davis, CA
ah@softwareprocess.es

Prem Devanbu
University of California, Davis
Davis, CA
devanbu@ucdavis.edu

ABSTRACT

Software readability is a property that influences how easily a given piece of code can be read and understood. [illegible] maintainability, quality, etc., programmers are very concerned about the readability of code. If automatic readability checkers could be built, they could be integrated into software development tool-chains, and thus encourage project managers to pay attention to it. Unfortunately, the readability of a piece of code is ultimately a human judgment, and therefore requires human studies to validate any given model. On the other hand, human judgments are both difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, S, accompanied by mean subjective readability scores. $\mathcal{O}(s)$ [illegible] automatically gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but [illegible] We show that a [illegible] basic information theory. In particular we argue that fundamental aspects of readability have been measured since [illegible] 1970s by Halstead's [illegible] [21]. We show that the Halstead metrics can be used to improve upon the

Human Factors, Theory, Measurement

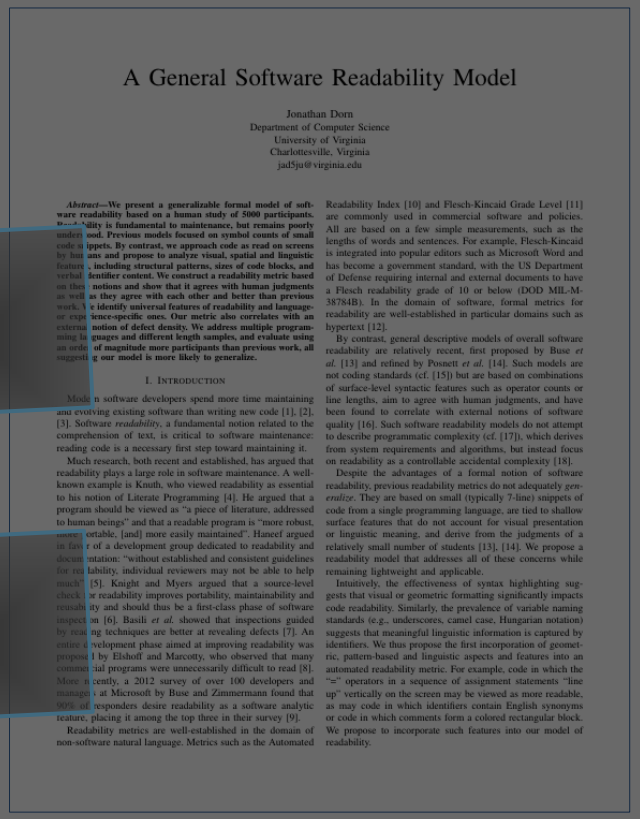
Keywords

1. INTRODUCTION

Readability of code is of central concern for [illegible] code is more readable today is presumed to remain easier to read, comprehend, and maintain at a later date. We conceive of readability as a subjective impression that [illegible] of the difficulty of code as there are to [illegible] readability is the power the human

There is much previous work about readability [7, 15, 34, 35, 13, 3]. One major issue with studies of readability is the difficulty of experimentally verifying what is essentially a subjective perception. Measures of readability [illegible] are both difficult to obtain, requiring human studies, and also are inherently variable; large-scale surveys, involving multiple human raters, and careful statistical analysis of inter-rater agreement are required to obtain usable measures. Buse *et al.*'s work was a major contribution in this area: they conducted a fairly large-scale study, asking human subjects to provide subjective rating scores of the readability of code snippets. These scores were validated and aggregated to yield mean-opinion-scores [6]. The result of this extensive and time-consuming study was a set of code-snippets, S, accompanied by mean subjective readability scores. $\mathcal{O}(s)$ [illegible] automatically gathered direct, automatically-derived, token-level measures, $\mathcal{M}(s)$, of the code snippets. They then built a logistic regression model to predict the subjective, laboriously gathered scores, $\mathcal{O}(s)$, using the collection of automatically gathered direct metrics, $\mathcal{M}(s)$. The $\mathcal{M}(s)$ were essentially token-level measures, but [illegible] We show that a [illegible] basic information theory. In particular we argue that fundamental aspects of readability have been measured since [illegible] 1970s by Halstead's [illegible] [21]. We show that the Halstead metrics can be used to improve upon the

MSR '11, 21-MAY-2011, Waikiki, Honolulu, USA



A General Software Readability Model

Jonathan Dorn
Department of Computer Science
University of Virginia
Charlottesville, Virginia
jad5ju@virginia.edu

***Abstract*—We present a generalizable formal model of software readability based on a human study of 5000 participants. Readability is fundamental to maintenance, but remains poorly understood. Previous models focused on symbol counts of small code snippets. By contrast, we approach code as read on screens by human eyes and propose to analyze visual, spatial and linguistic features, including structural patterns, sizes of code blocks, and verbal identifier content. We construct a readability metric based on these notions and show that it agrees with human judgments as well as they agree with each other and better than previous work. We identify universal features of readability and language-or experience-specific ones. Our metric also correlates with an external notion of defect density. We address multiple programming languages and different length samples, and evaluate using an order of magnitude more participants than previous work, all suggesting our model is more likely to generalize.**

I. Introduction

Modern software developers spend more time maintaining and evolving existing software than writing new code [1], [2], [3]. Software *readability*, a fundamental notion related to the comprehension of text, is critical to software maintenance: reading code is a necessary first step toward maintaining it.

Much research, both recent and established, has argued that readability plays a large role in software maintenance. A well-known example is Knuth, who viewed readability as essential to his notion of Literate Programming [4]. He argued that a program should be viewed as "a piece of literature, addressed to human beings" and that a readable program is "more robust, more portable, [and] more easily maintained". Haneef argued in favor of a development group dedicated to readability and documentation: "without established and consistent guidelines for readability, individual reviewers may not be able to help much" [5]. Knight and Myers argued that a source-level check for readability improves portability, maintainability and reusability and should thus be a first-class phase of software inspection [6]. Basili *et al.* showed that inspections guided by reading techniques are better at revealing defects [7]. An entire development phase aimed at improving readability was proposed by Elshoff and Marcotty, who observed that many commercial programs were unnecessarily difficult to read [8]. More recently, a 2012 survey of over 100 developers and managers at Microsoft by Buse and Zimmermann found that 90% of responders desire readability as a software analytic feature, placing it among the top three in their survey [9].

Readability metrics are well-established in the domain of non-software natural language. Metrics such as the Automated Readability Index [10] and Flesch-Kincaid Grade Level [11] are commonly used in commercial software and policies. All are based on a few simple measurements, such as the lengths of words and sentences. For example, Flesch-Kincaid is integrated into popular editors such as Microsoft Word and has become a government standard, with the US Department of Defense requiring internal and external documents to have a Flesch readability grade of 10 or below (DOD MIL-M-38784B). In the domain of software, formal metrics for readability are well-established in particular domains such as hypertext [12].

By contrast, general descriptive models of overall software readability are relatively recent, first proposed by Buse *et al.* [13] and refined by Posnett *et al.* [14]. Such models are not coding standards (cf. [15]) but are based on combinations of surface-level syntactic features such as operator counts or line lengths, aim to agree with human judgments, and have been found to correlate with external notions of software quality [16]. Such software readability models do not attempt to describe programmatic complexity (cf. [17]), which derives from system requirements and algorithms, but instead focus on readability as a controllable accidental complexity [18].

Despite the advantages of a formal notion of software readability, previous readability metrics do not adequately *generalize*. They are based on small (typically 7-line) snippets of code from a single programming language, are tied to shallow surface features that do not account for visual presentation or linguistic meaning, and derive from the judgments of a relatively small number of students [13], [14]. We propose a readability model that addresses all of these concerns while remaining lightweight and applicable.

Intuitively, the effectiveness of syntax highlighting suggests that visual or geometric formatting significantly impacts code readability. Similarly, the prevalence of variable naming standards (e.g., underscores, camel case, Hungarian notation) suggests that meaningful linguistic information is captured by identifiers. We thus propose the first incorporation of geometric, pattern-based and linguistic aspects and features into an automated readability metric. For example, code in which the "=" operators in a sequence of assignment statements "line up" vertically on the screen may be viewed as more readable, as may code in which identifiers contain English synonyms or code in which comments form a colored rectangular block. We propose to incorporate such features into our model of readability.

Structural features

Visual features

Code readability prediction

Learning a Metric for Code Readability

Raymond P.L. Buse, Westley Weimer

A Simpler Model of Software Readability

Daryl Posnett, Abram Hindle, Prem Devanbu

A General Software Readability Model

Jonathan Dorn

Structural features

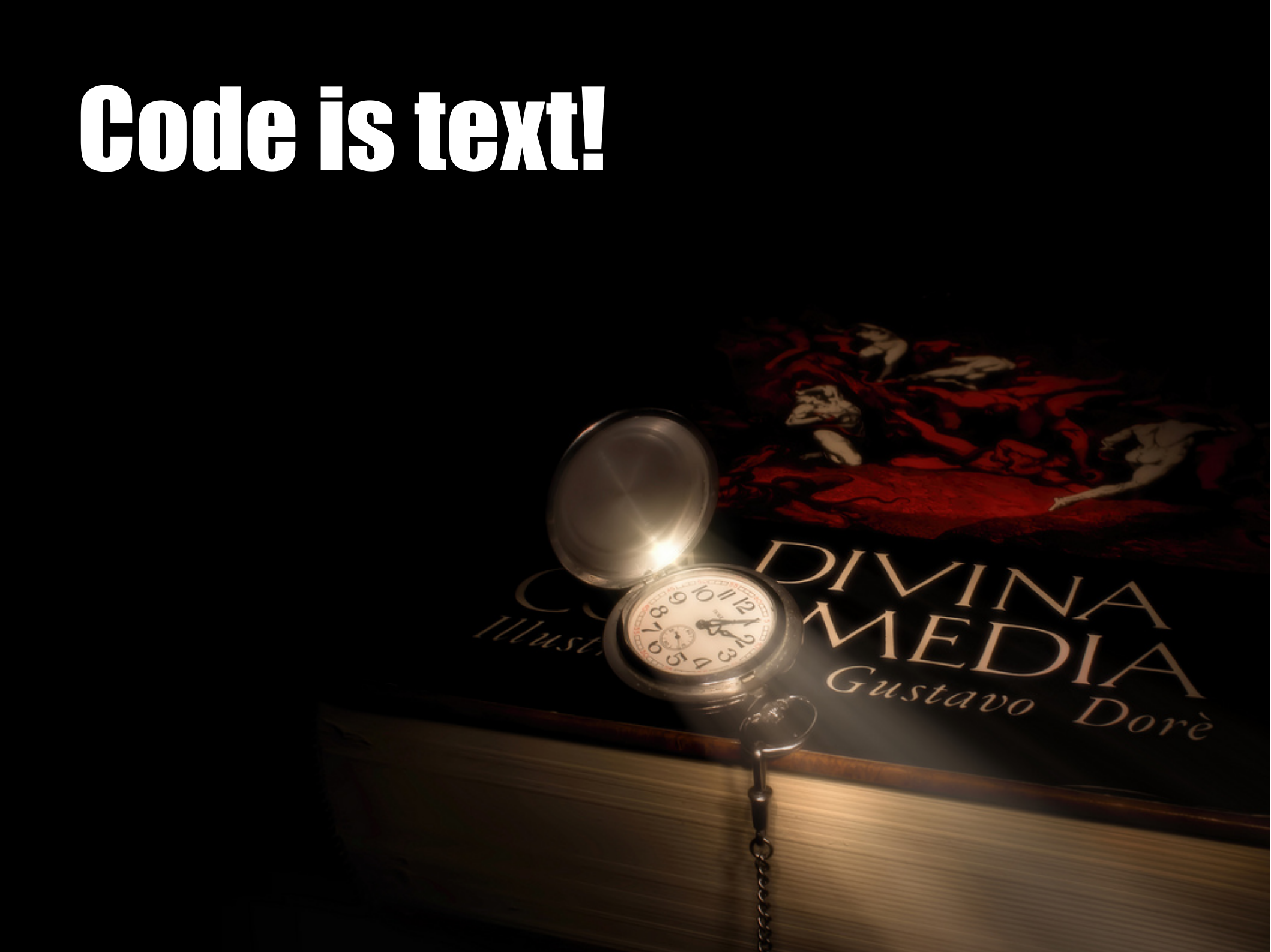
Visual features

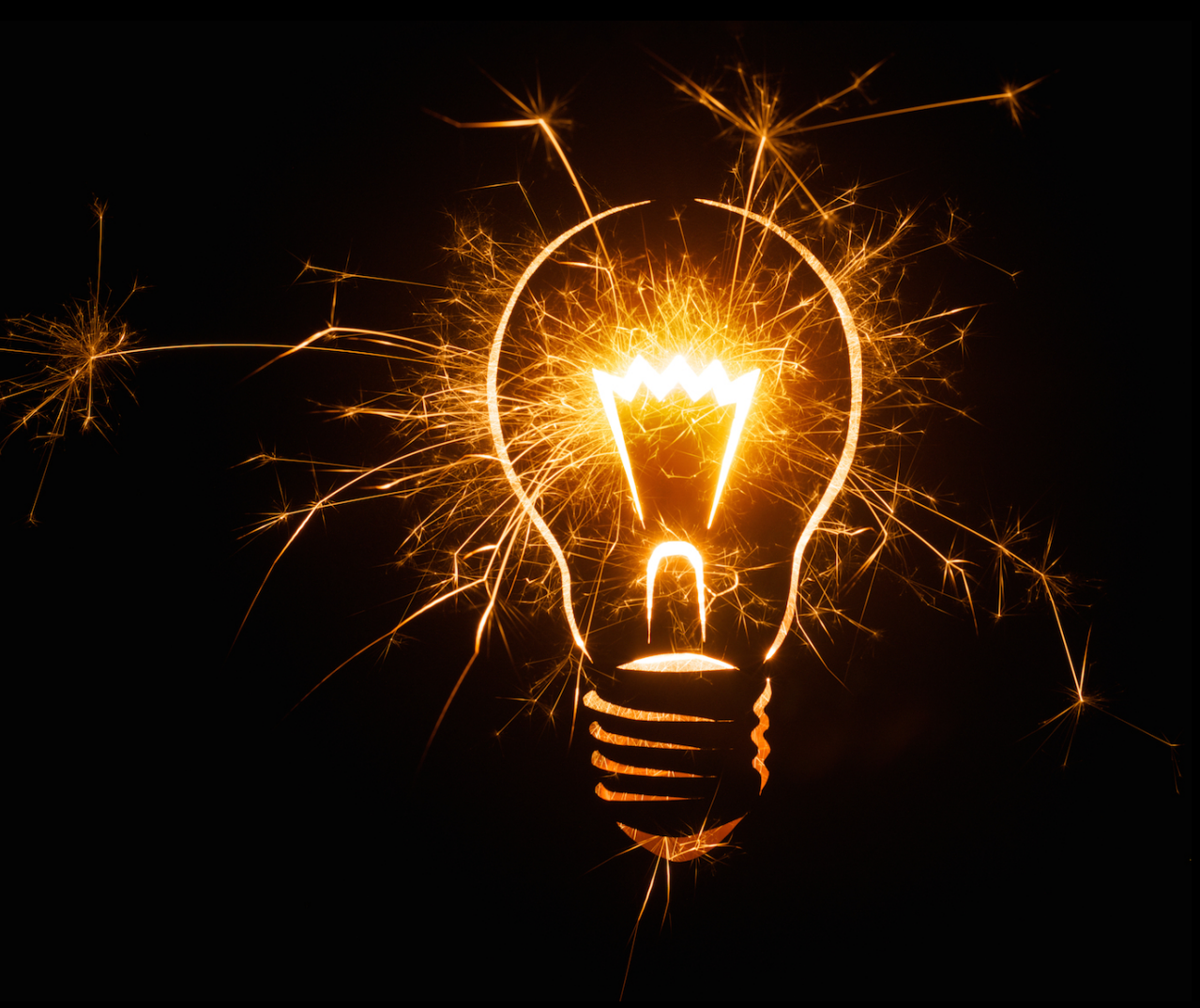
Two datasets

Code readability prediction


Something is missing...

Code is text!





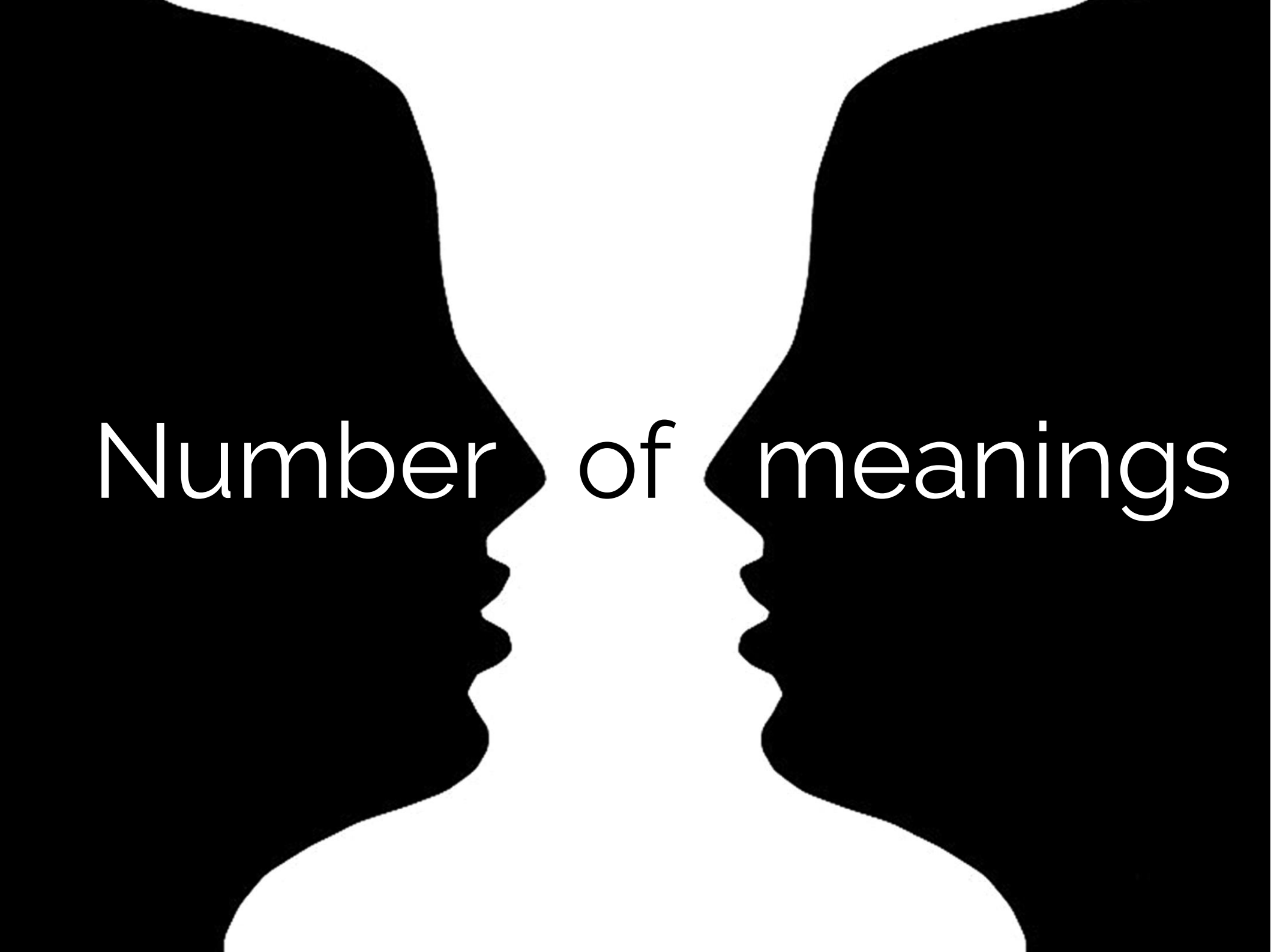
New features

Comments readability


Comments and
identifiers consistency

Identifier terms in dictionary

Narrow meaning identifiers


Number of meanings

Textual coherence

Case study

200
Java snippets

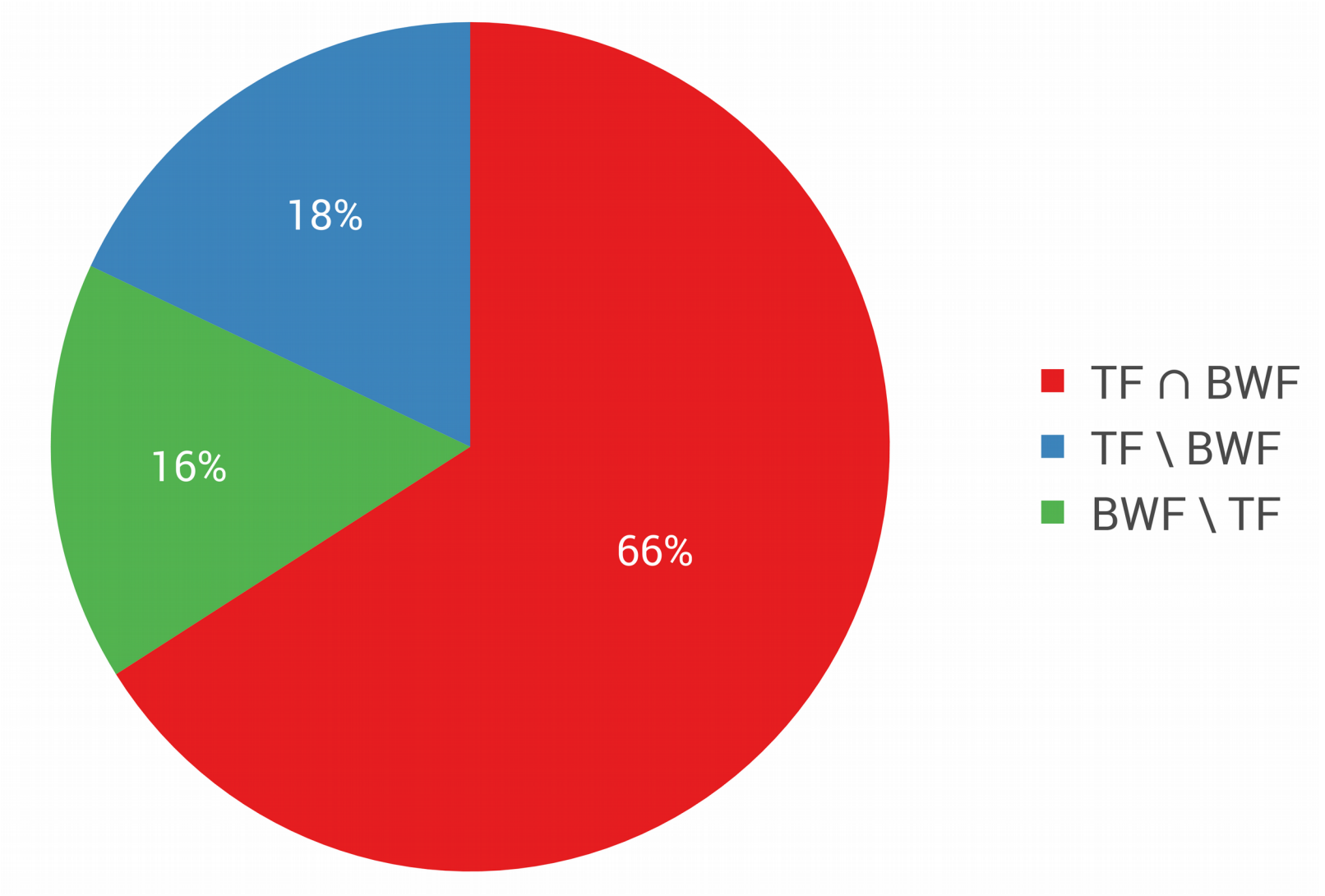
9
annotators

New dataset

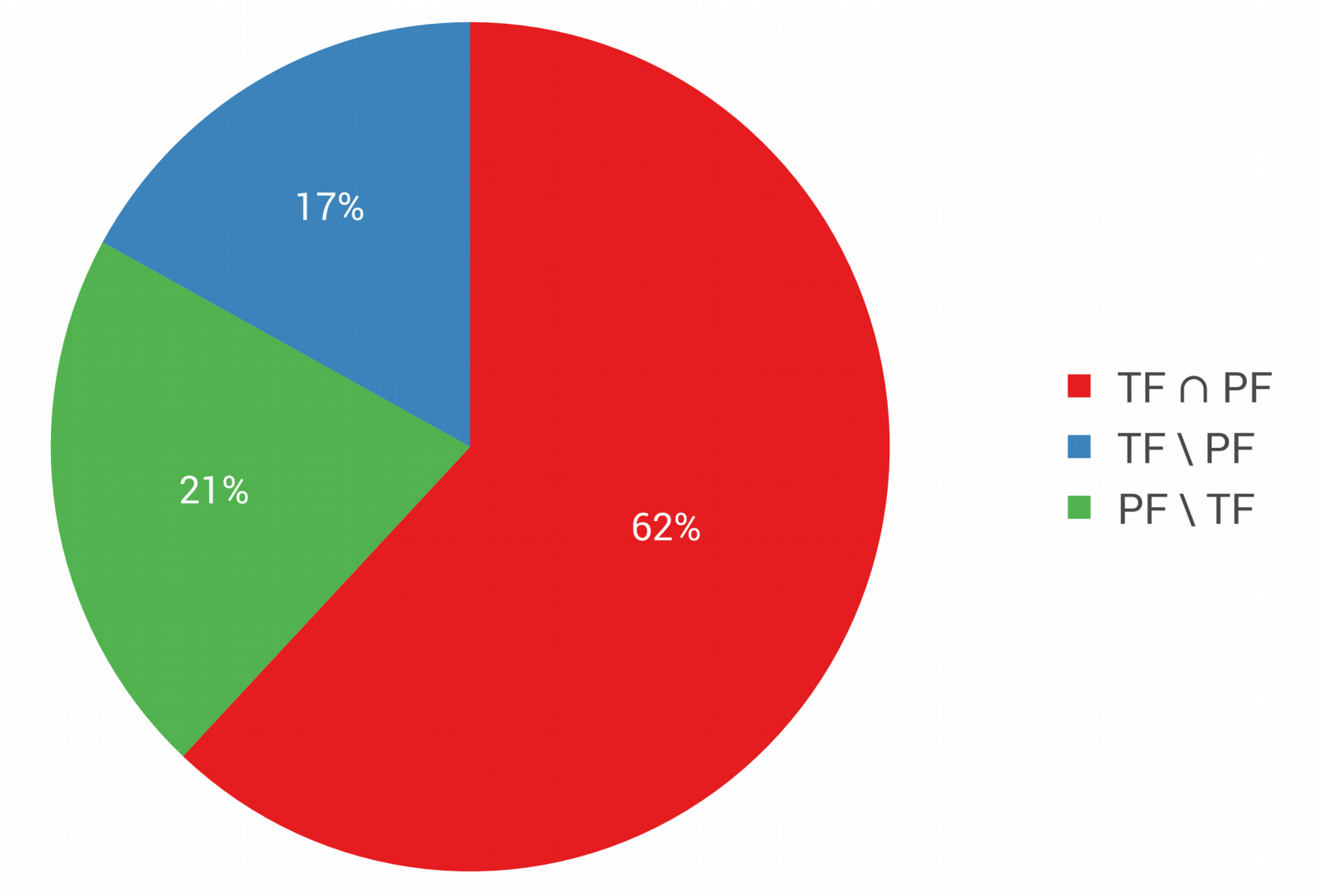
Do textual features complement the others proposend in the literature?

Overlap metrics

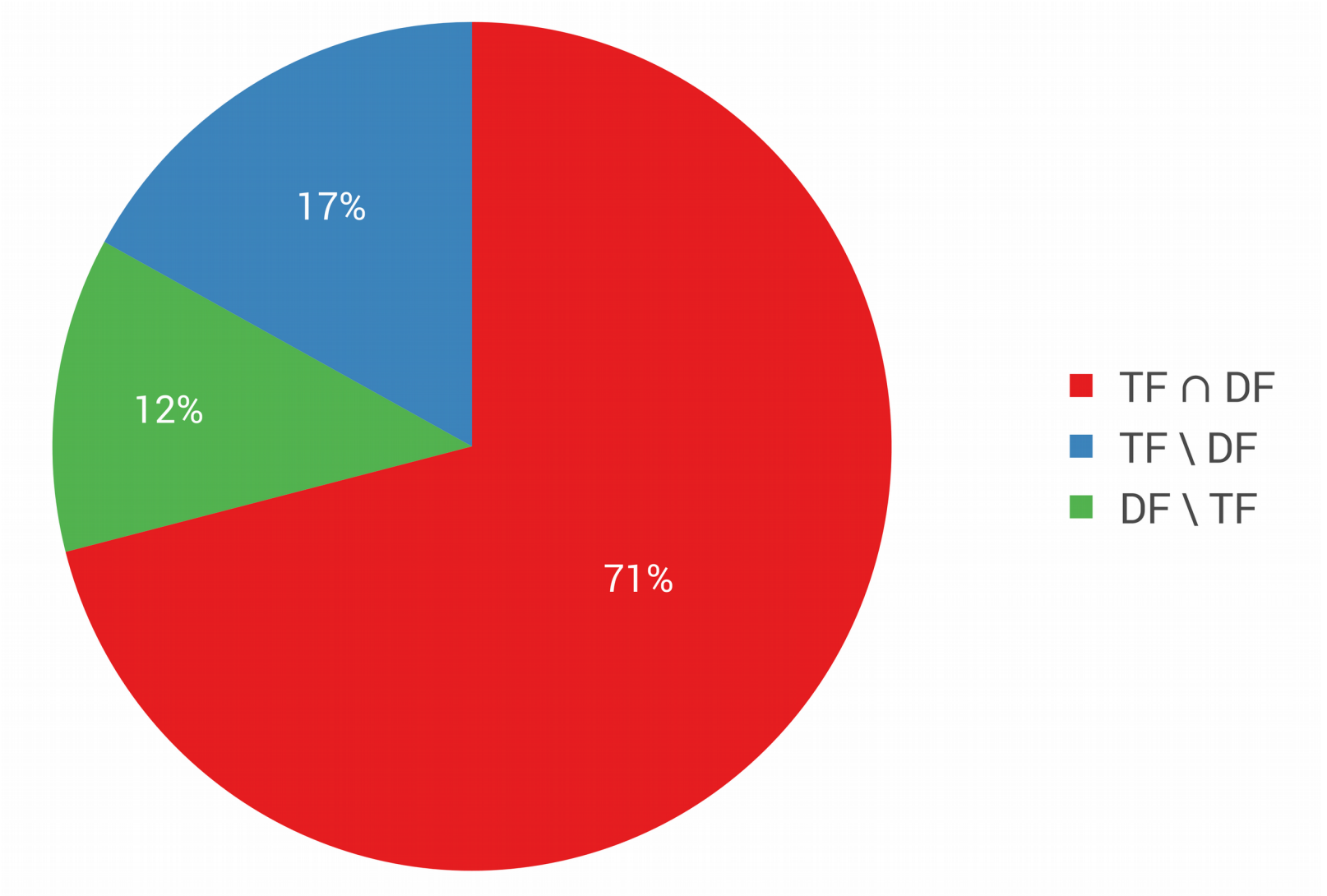
Textual Features vs Buse's



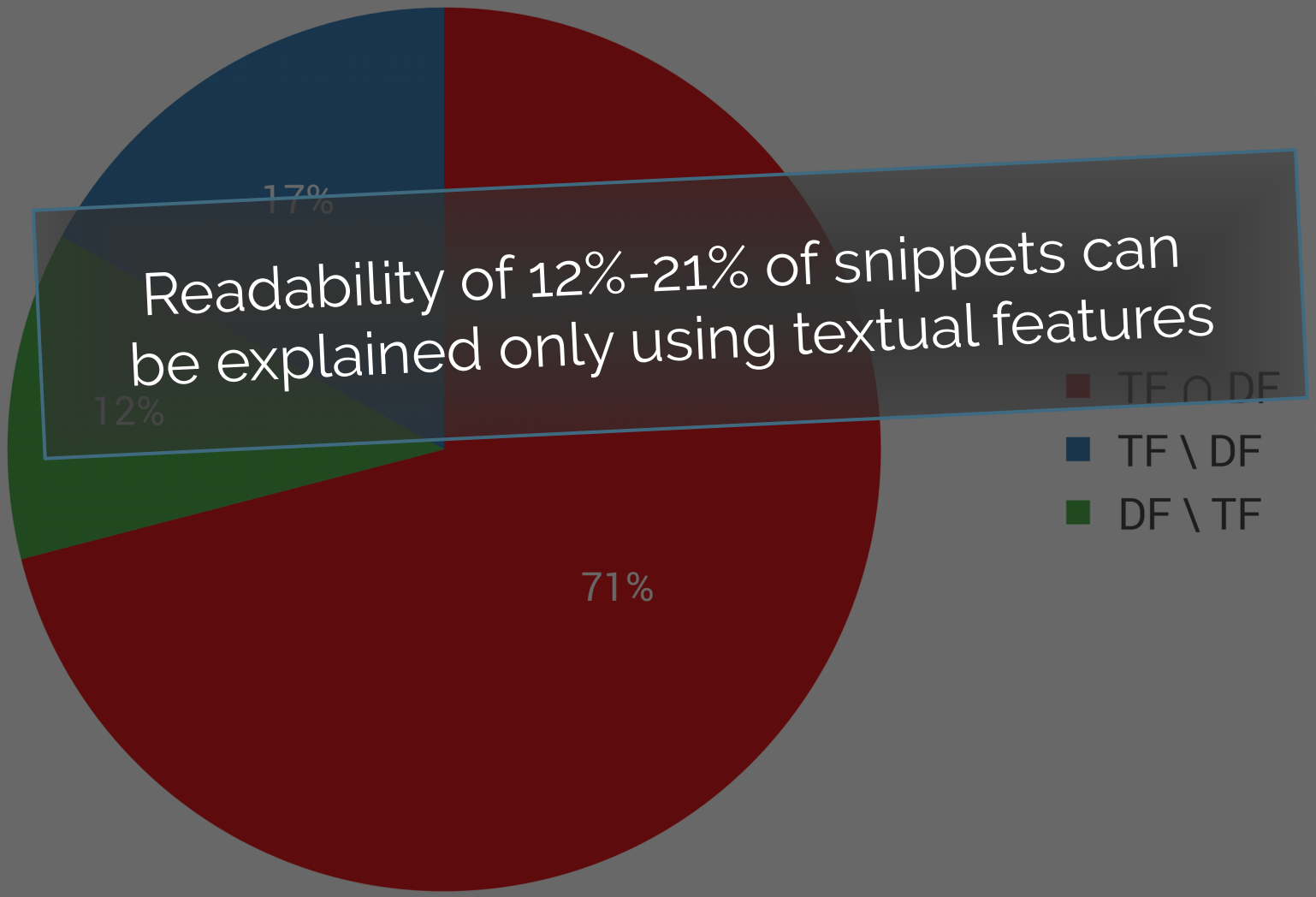
Textual Features vs Posnett's



Textual Features vs Dorn's



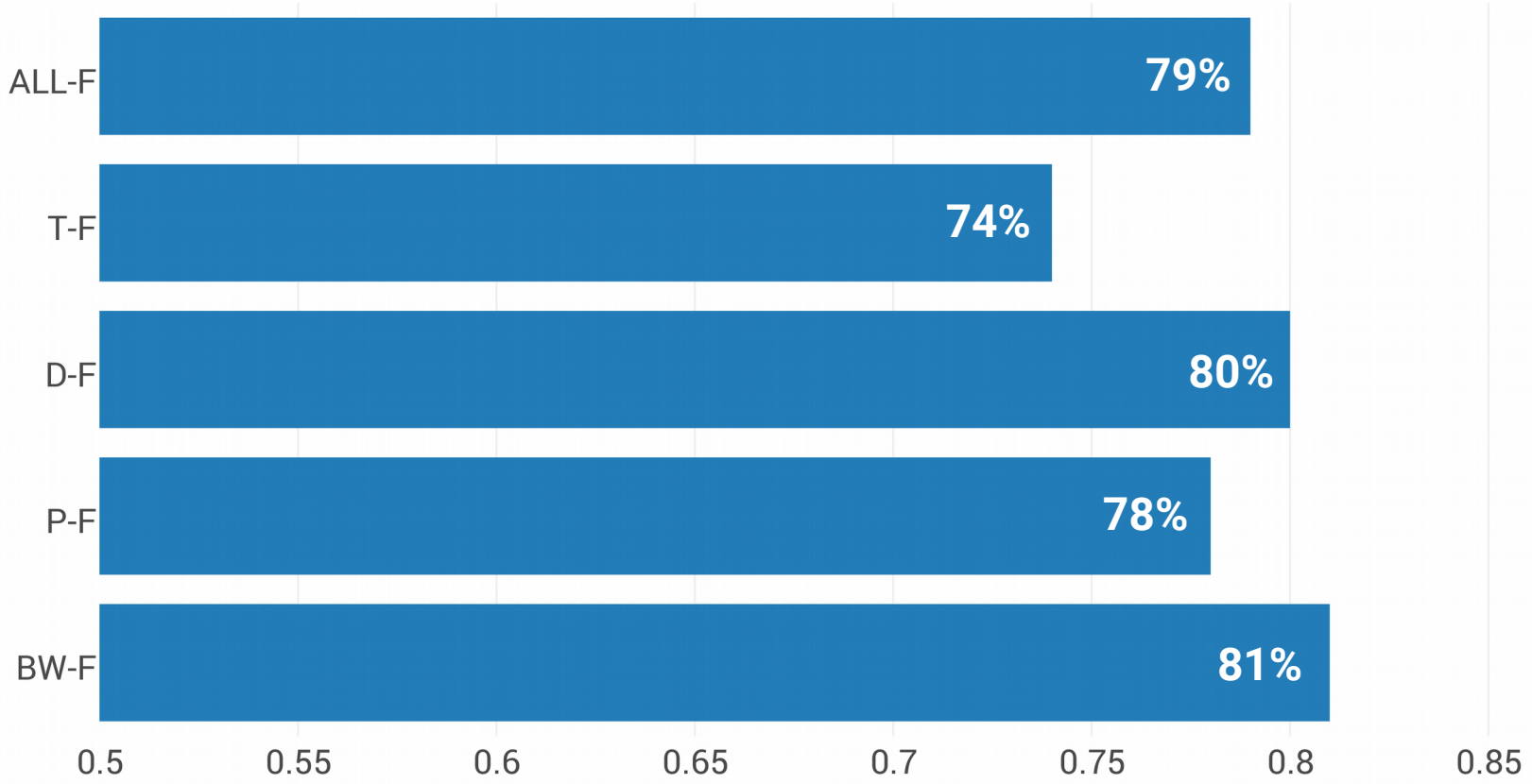
Textual Features vs Dorn's



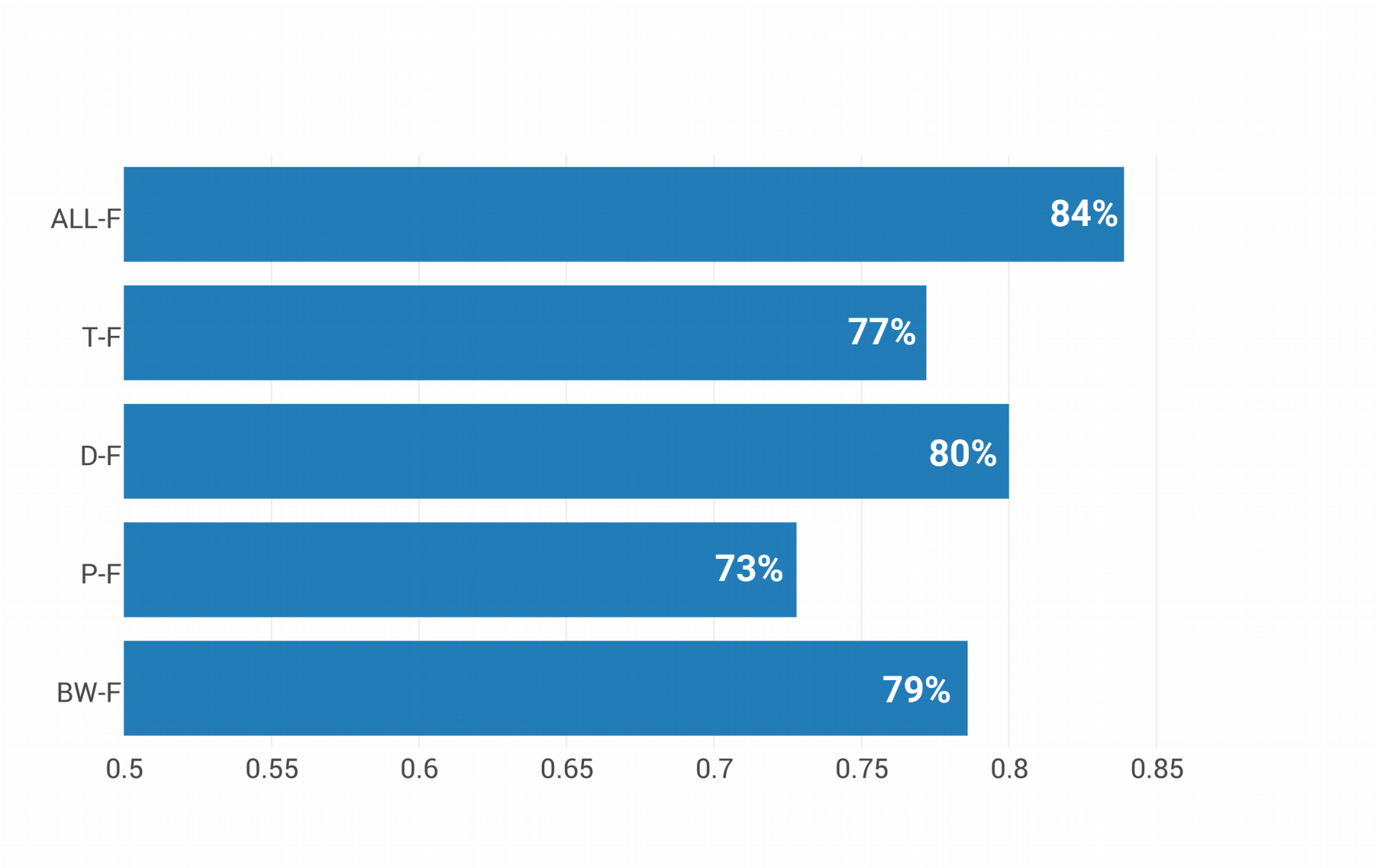
Readability of 12%-21% of snippets can be explained only using textual features

What is the accuracy of a readability model based on structural and textual features?

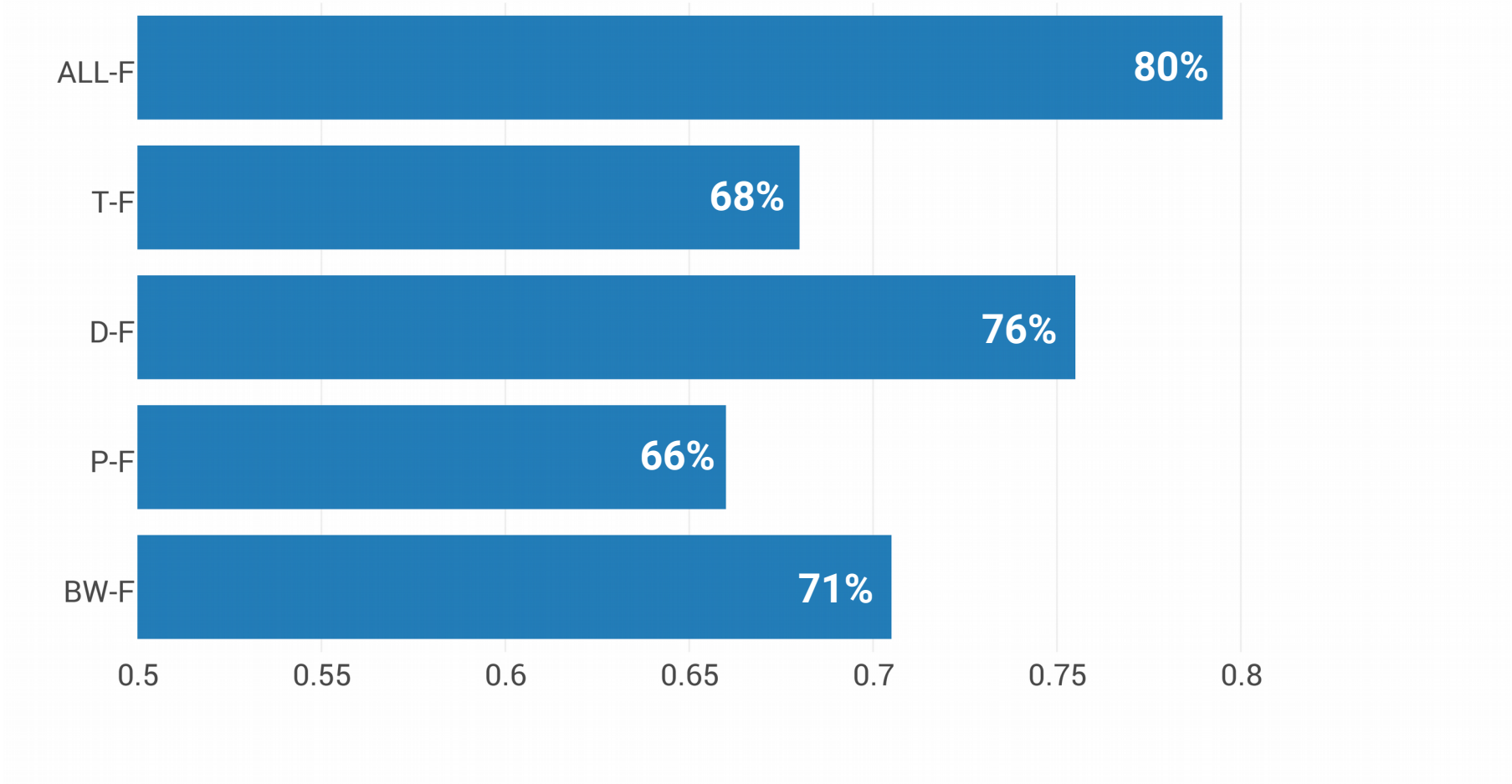
Dataset by Buse and Weimer



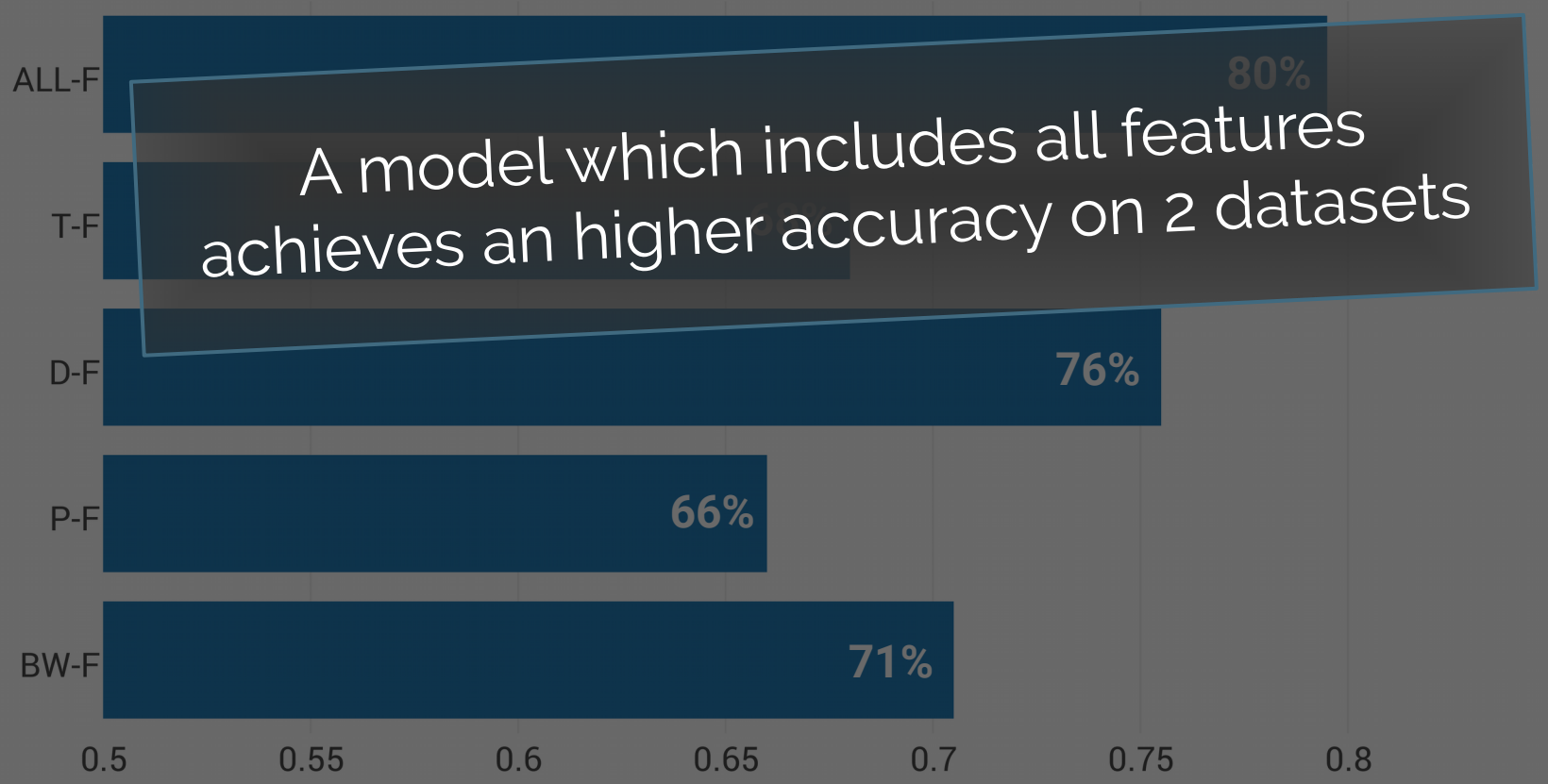
Dataset by Dorn



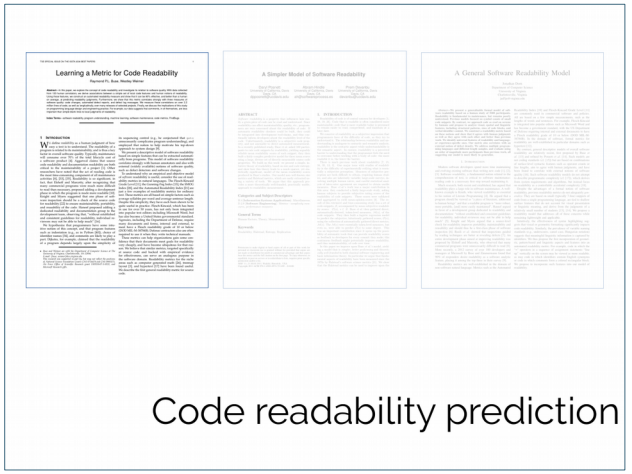
New dataset

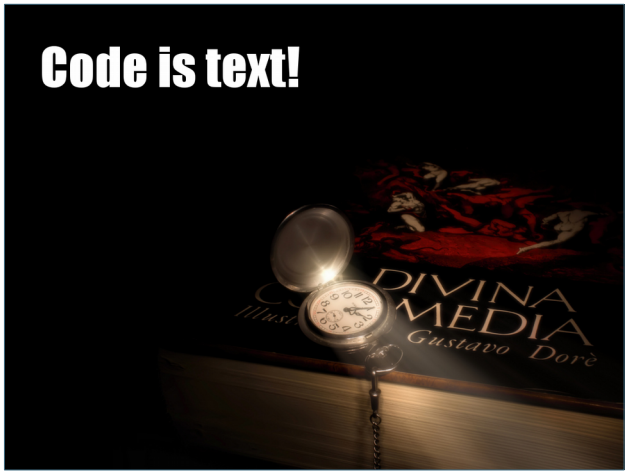


New dataset


ALL-F
T-F
D-F
P-F
BW-F
80%
76%
66%
71%
0.5
0.55
0.6
0.65
0.7
0.75
0.8
A model which includes all features
achieves an higher accuracy on 2 datasets

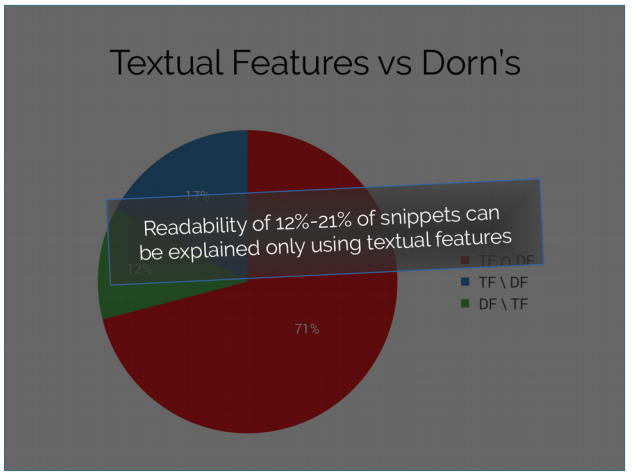
In summary...

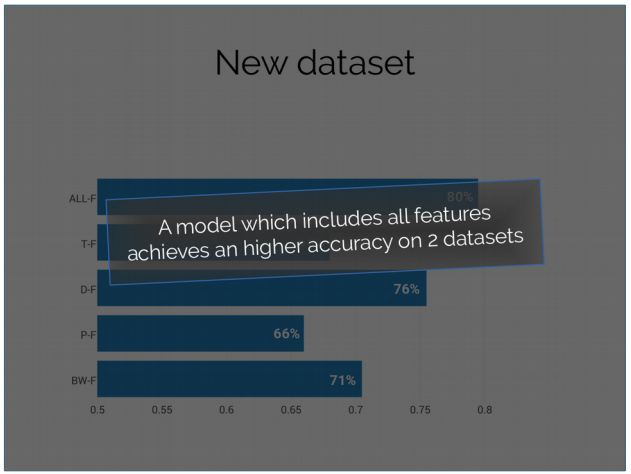

Learning a Metric for Code Readability
A Simpler Model of Software Readability
A General Software Readability Model
Code readability prediction

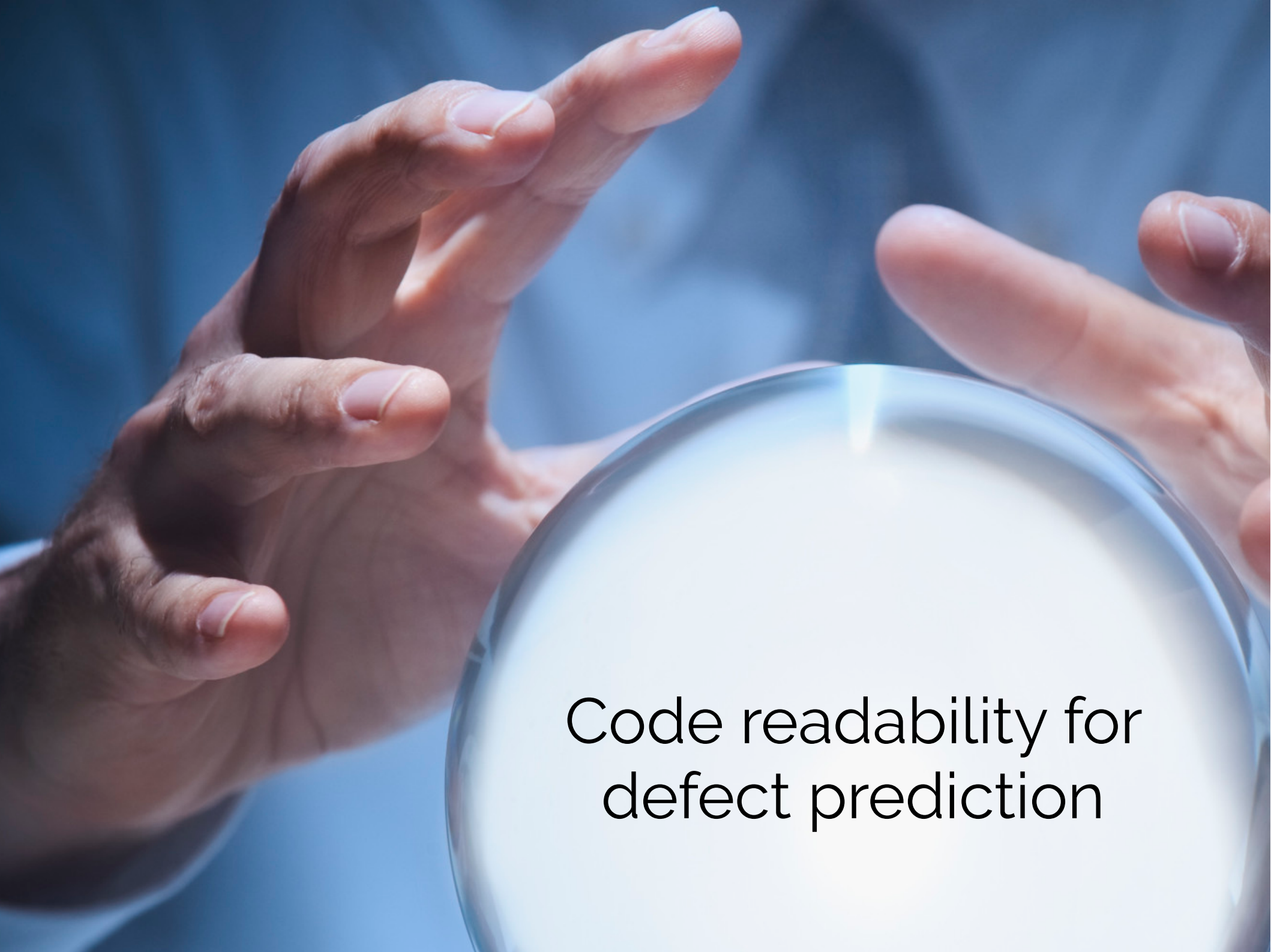

Code is text!
DIVINA
MEDIA
Gustavo Dorè


Case study


200
Java snippets
9
annotators
New dataset


Textual Features vs Dorn's
Readability of 12%-21% of snippets can be explained only using textual features
TF ∩ DF
TF \ DF
DF \ TF
71%


New dataset
A model which includes all features achieves an higher accuracy on 2 datasets
ALL-F
T-F
D-F
P-F
BW-F
80%
76%
66%
71%
0.5
0.55
0.6
0.65
0.7
0.75
0.8


Code readability for
defect prediction

Thanks.