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.

In addition, we target multiple languages with potentially distinct notions of readability: Java, a verbose object-oriented baseline; Python, which eschews curly braces but enforces indentation; and CUDA, a C-and-assembly derived language for programming GPUs. We consider varying amounts of code, from the small snippets of previous work to samples representing an entire 50-line screen. Finally, we base our model on the judgments of over 5,000 different human annotators, of which 1,800 have industry experience.

The contributions of this paper are as follows:

- We present a formal descriptive model of software readability based on syntactic notions as well as novel features related to the visual and linguistic presentation of code in modern software development. (Section III)
- We perform a human study involving over 5,000 humans, three programming languages, and 360 code samples, each up to 50 lines long (larger than previous work in all cases, an order of magnitude larger in some). (Section IV) Our metric correlates with human judgments of readability 2.3 times better than does previous work. (Section V-A)
- We demonstrate that our readability metric correlates with defect density, an external notion of software quality. Less readable code is more likely to contain analysis-reported defects by a factor of 2.2. (Section V-C)
- Using our large dataset, we analyze the commonalities and differences in readability judgments across languages, human experience levels, and programming backgrounds. For example, we find that student judgments of readability are explained by concerns like the use of operators and numeric literals, while industry experience leads to judgments that place a higher priority on whitespace. In addition, while we find that line length is important to the readability of programs in the aggregate, other factors such as the extent of operator usage often play a larger role within individual languages. (Section V-D)

II. Motivation

In this section we motivate the need for visual, spatial and linguistic features in a readability model, as well as detailing a lack of generality in previous models. Consider the partial Python code in Figure 1. It features well-aligned procedure bodies, a visibly-patterned structure with similarly-shaped blocks of text, and the frequent use of English-language words inside identifier names (e.g., "children" serves as both a field name and as part of a method name). In our human study, two-thirds of participants viewing the full code rated it as readable (i.e., a 4 or 5 on a 5-point scale). Intuitively, the regular visual structure and selection of identifiers seem to contribute to this judgment.

However, previous attempts to model readability are unable to capture that intuition. The readability tool of Buse *et al.* [13] gives the entire code a score of 0% (the lowest possible). This occurs even though the code in question is interspersed with blank lines (the most powerful positive feature in the Buse model) and contains relatively few long lines or identifiers (the two most negative features). Work by Posnett *et al.* [14] has previously demonstrated that the Buse model does not generalize to samples much larger than about seven lines. In addition, this example helps to demonstrate that it fails to capture visual and linguistic regularity.

```

def handleBlockQuote(node):
    result = BlockQuoteDitem(node.nodeName)
    result.children = processChildren(node)
    return result

def handleList(node):
    result = ListDitem(node.nodeName)
    result.children = processChildren(node)
    return result

def handleListItem(node):
    result = ListItemDitem(node.nodeName)
    result.children = processChildren(node)
    return result

def handleTable(node):
    result = TableDitem(node.nodeName)
    # Ignore table contents that are not tr
    result.children = [x
```

Fig. 1: Sample code from the Python Docutils package (lines after 20 omitted for space). Two-thirds of human annotators rated this code as quite readable (4 or 5 on a 5-point scale). By contrast, the Buse readability tool rates it 0% readability, its lowest possible score.

As another example of this line of reasoning, consider the sample CUDA code in Figure 2. In our human study, two-thirds of human annotators rated the code as unreadable (i.e., a 1 or 2 out of 5). A naïve counting suggests that it is heavily commented: 61% of its characters appear in comments. However, a closer inspection indicates the comments follow a less-desired pattern: one comment contains another (line 1) and the majority of the comments serve to remove code rather than to explicate it (it may not be immediately clear if any of the three comments describe the current behavior of the code).

The Buse metric does not match our human annotators' judgments and gives this sample a high 94% readability score (out of 100%). Even though this sample is seven lines long, the desired length for the Buse metric, its results are still unintuitive: it counts the number of comments without considering the spatial structure (i.e., the long, thin and nested comments on line 1 and 2). Similarly, it counts the number of identifiers without regard for their length or content. This code sample features a clear identifier naming convention, but the first non-comment identifier starts with "`et`", which is not obviously an English word.

We desire a formal descriptive metric of software readability that more closely matches human intuitions and generalizes to handle visual and linguistic features, longer code samples, and multiple programming languages. In the next section we describe our model, building on previous purely-syntactic approaches and extending them with richer semantic features.

```
    //float *attenuationIntegralPlaneArray_d;    //stores partial integral on planes parallel to the camera
    //CUDA_SAFE_CALL(cudaMalloc((void **)&attenuationIntegralPlaneArray_d, img->dim[1]*img->dim[3]*sizeof(float)));

    et_line_integral_attenuated_gpu_kernel <<<G1,B1>>> (*d_activity, *d_attenuation, currentCamPointer);

    CUDA_SAFE_CALL(cudaThreadSynchronize());
  }
```

Fig. 2: Sample CUDA code from the NiftyRec project. Two-thirds of human annotators rated this code as unreadable (1 or 2 on a 5-point scale). By contrast, the Buse readability metric gives it a high 94% readability score.

III. Readability Model

While readability is widely regarded as critical to code quality and maintainability [1], [2], [3], [5], [8], [17], [19], [20], [21], it is not clear how such a potentially subjective notion should be modeled formally [22]. Buse *et al.* first proposed a model based on a large number of shallow syntactical features [16], while Posnett *et al.* proposed an improved model based on a much smaller number of features [14].

Following previous work, we define readability as a mapping from a code sample (some number of contiguous lines) to a continuous score domain. We base our mapping on a weighted logistic combination of numerical features derived from the code sample. Unlike previous work, we design and include features that capture structural patterns, visual perception, alignments, and natural language notions, in addition to punctuation and syntax.

Since our model takes into account the spatial structure and natural language content of various tokens, we require a lexer definition for each subject language (i.e., a list of regular expressions and associated tokens). For languages where token types may depend on context (e.g., "identifier" vs. "type name" in C-like languages), we treat all such tokens as identifiers. This approach allows our metric to capture more semantic richness than pure character counting, but without requiring formal parsing (and thus we can still analyze "ill-formed" code fragments). This is in contrast to previous metrics [13], [14], which count punctuation characters without considering language rules. Similarly, our metric requires a dictionary for the natural languages used in the code development (cf. the spelling checker in a modern IDE).

In this section we describe our features in general. In the next section we describe our human study and how we learn the relative weights of these features.

a) Structural Pattern Features: We propose several types of visual features for modeling readability. The first measures the frequency of changes in characteristics such as indentation from line to line in the source code. Intuitively, code that smoothly transitions into and out of indentation blocks and that is more consistent in the size of nested blocks or commented regions may provide a comfortable visual framework for the programmer. In some cases, such a regular framework would foster a more rapid understanding of the structure and semantics of the code. Consider Figure 1: the regular shapes of the function bodies may help the reader to understand that the functions have similar meanings.

```
        const int tid=blockIdx.x*blockDim.x+threadIdx.x;
        if(tid < c_VoxelNumber){
            int3 imageSize = c_ImageDim;

            short z=(int)(tid/((imageSize.x)*(imageSize.y))

            int radius = (windowSize-1)/2;
            int index = tid - imageSize.x*imageSize.y*radiu
            z -= radius;

            float4 finalValue = make_float4(0.0f, 0.0f, 0.0

            for(int i=0; i<windowSize; i++){
                if(-1<z && z<imageSize.z){
                    float4 gradientValue = tex1Dfetch(gradi
                    float windowValue = tex1Dfetch(convolut
                    finalValue.x += gradientValue.x * windo
                    finalValue.y += gradientValue.y * windo
                    finalValue.z += gradientValue.z * windo
                }
                index += imageSize.x*imageSize.y;
                z++;
            }

            smoothedImage[tid] = finalValue;
        }
        return;
    }
```

Fig. 3: Sample CUDA code from the NiftyRec project (long lines have been truncated for space). The indentation pattern in this sample may be reasonably reconstructed using just the first 5 components of the discrete Fourier transform.

We extend our intuition that certain aspects of readable code should change in a regular pattern from one line to the next to other line-based metrics. For example, line length, the nesting of function calls, and the use of whitespace are also relevant. We codify this intuition mathematically: we measure the line length frequency by measuring the values for each line of code and taking the discrete Fourier transform (DFT) of that signal [23]. The DFT computes the coefficients to reconstruct a signal using a Fourier series, and can be thought of as capturing the frequency distribution of the signal. It is commonly used in image processing (e.g., [24]) but has not, to the best of our knowledge, been previously used to help tease apart software readability. For each line-based metric, we compute both the bandwidth of the signal and the amplitudes of the primary frequencies; these indicate the significance of rapid changes in the feature as well the broader pattern.

To make this discussion concrete, consider Figure 6a, which shows the measured indentation of the code sample in Figure 1. The amplitudes of the coefficients of the corresponding DFT are shown in the solid curve in Figure 6b. In determining

```
class class_attribute(PythonStructural, Element): pass
class expression_value(PythonStructural, Element): pass
class attribute(PythonStructural, Element): pass

# Structural Support Elements
# ---------------------------

class parameter_list(PythonStructural, Element): pass
class parameter_tuple(PythonStructural, Element): pass
class parameter_default(PythonStructural, TextElement):
class import_group(PythonStructural, TextElement): pass
class import_from(PythonStructural, TextElement): pass
class import_name(PythonStructural, TextElement): pass
class import_alias(PythonStructural, TextElement): pass
class docstring(PythonStructural, Element): pass

# =================
#  Inline Elements
# =================

# These elements cannot become references until the sec
# pass.  Initially, we'll use "reference" or "name".

class object_name(PythonStructural, TextElement): pass
class parameter_list(PythonStructural, TextElement): pa
class parameter(PythonStructural, TextElement): pass
class parameter_default(PythonStructural, TextElement):
class class_attribute(PythonStructural, TextElement): p
class attribute_tuple(PythonStructural, TextElement): p
```

Fig. 4: Sample Python code from the Docutils project (long lines have been truncated for space). The three relatively wide comment blocks in this snippet can be adequately represented with low frequencies, resulting in a smaller bandwidth in the X direction.

```
                deltaW += vd[threadIdx.x] * hd[threadI
        }

        // update weights
        if (i < I && j < J) {
                deltaW /= samples;

                int w = j * I + i;

                cudafloat learningRate = UpdateLearnin
                UpdateWeight(learningRate, momentum, d
        }

        if(i < I && threadIdx.y == 0) {
                errors[i] = error;

                // Update a
                if (j == 0) {
                        deltaA /= samples;

                        cudafloat learningRate = Updat
                        UpdateWeight(learningRate, mom
                }
        }

        // Update b
        if (i == 0 && j < J) {
                deltaB /= samples;
```

Fig. 5: Sample CUDA code from the GPUMLib project (long lines have been truncated for space). The three comments are small relative to the rest of the code; the longest line is 135 characters long. Adequately representing these short comments in the DFT requires a significant number of high-frequency adjustments, resulting in a larger bandwidth in the X direction.

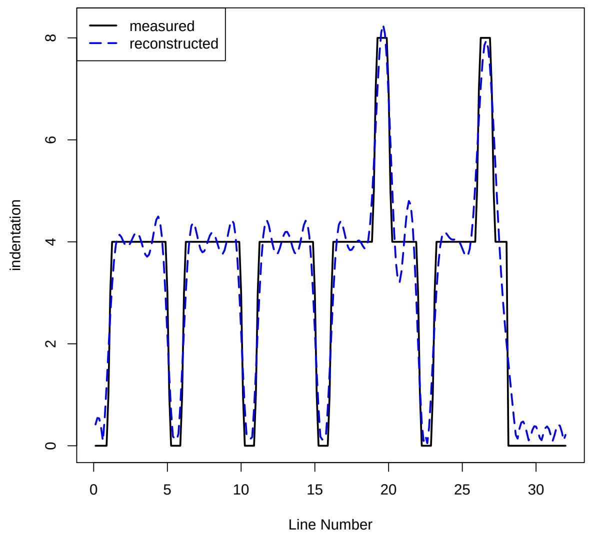
the bandwidth, we are interested in how much fine-tuning of high-frequency components is needed to describe the original signal. That is, how wide is the range of frequencies containing most of the information in the signal? We would like to select a bandwidth such that the amplitudes for these frequencies are sufficient to reconstruct a good approximation of the original signal, as shown in the dashed reconstruction in Figure 6a.

Since the DFT is even-symmetric and periodic, we can define the range $[-f, f]$ to be this range of frequencies; the bandwidth is therefore $2f + 1$. Finding the bandwidth reduces to finding an appropriate value for f. In many signal processing applications, the signal bandwidth is often taken to be the range of frequencies over which the amplitude is above -3 dB relative to the maximum amplitude [25]. However, due to the extreme amplitude of the first DFT coefficient for the code features we are analyzing, this rule typically results in a bandwidth of 1, corresponding to the range $[0, 0]$. This single coefficient permits the reconstruction of a sine wave; however, a sine wave typically does not fit the measured code features very well. Instead, we compute the bandwidth by finding the highest frequency f for which the amplitude is greater than standard deviation of the amplitudes. Empirically, we have found that this tends to provide good approximations of the original signal, as shown in Figure 6a and Figure 6c.

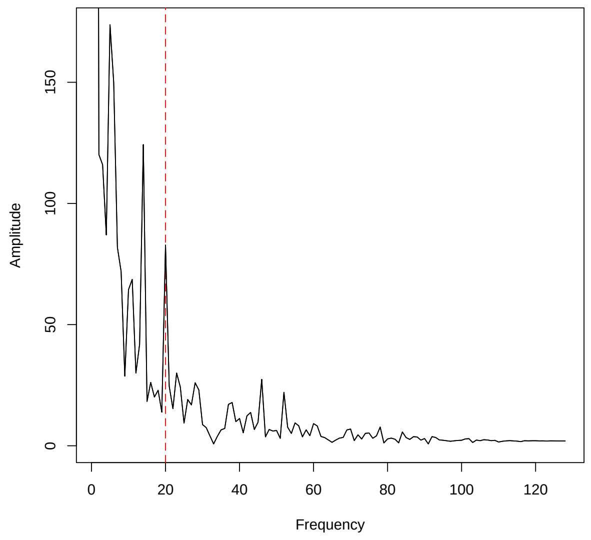
b) Visual Perception Features: The intuition for this set of features is that source code is often rendered on-screen for a programmer to view, that is, as a matrix of colored characters and whitespace. We propose to model the effects of syntax highlighting, which is commonly used but not universally standardized, by assigning different symbolic colors to characters that make up different sorts of tokens. To the extent that syntax highlighting positively impacts code readability, we would expect that visually large regions of same-color text will be less readable than text containing changes in color, since such same-color text is essentially not highlighted (i.e., the extra perceptual channel provided by color is not being used to make the code more easily understandable).

We extract two main types of features from this symbolically colored text. The first groups the characters in the code snippet according to their assigned color and computes the total character count for each such group. This effectively computes the total area highlighted with each color in the syntax highlighted text. Our metric measures the ratios between the areas occupied by each pair of colors. For example, consider Figure 2. The top two lines consist of commented code, to which we assign one symbolic color. Lines 4 and 6 consist primarily of identifiers, which we assign another color. Our metric computes the ratio of the number of characters in the comments to the number of characters involved in identifiers names; in this case, there are almost twice as many characters in the comments as in the identifiers. We also compute the fraction of the total text that is highlighted with each color, for example, in Figure 2, 61% of the text is in the comments.

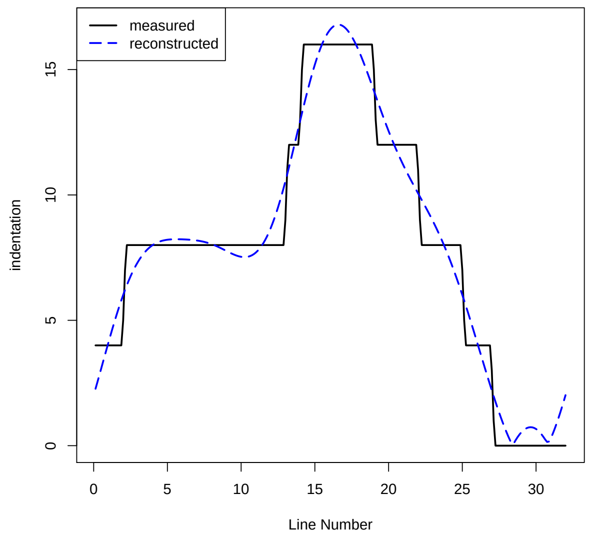
Our second visual feature takes the two-dimensional discrete Fourier transform of the image formed by the colored char-



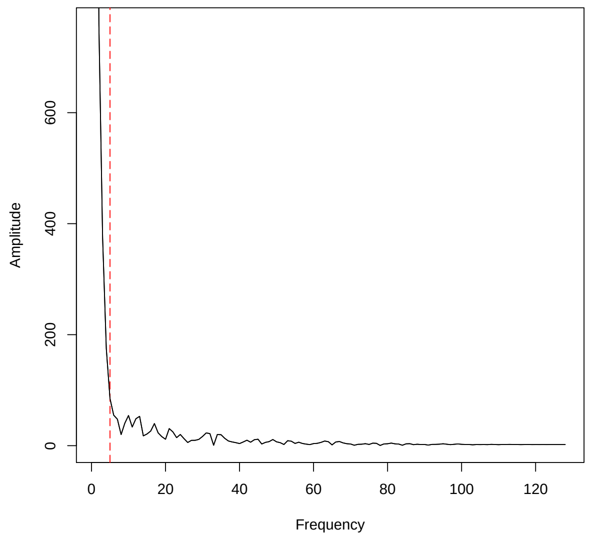
(a) Indentation for code in Figure 1 (solid) and DFT-based reconstruction (dashed).



(b) DFT for indentation in Figure 1.



(c) Indentation for code in Figure 3 (solid) and DFT-based reconstruction (dashed).



(d) DFT for indentation in Figure 3.

Fig. 6: Indentation and corresponding DFT of selected code samples. The dashed curves in (a) and (c) describe the signal reconstructed by setting the DFT amplitudes outside the central band to zero and taking the inverse DFT of the result. To show more of the detail of the transform, (b) and (d) show only the lower half of the frequency domain, since the Fourier transform is even-symmetric and periodic. The vertical dashed lines indicates the bandwidth of the signal; most of the information needed to reconstruct the signal lies between this line and its reflection around 0.

acters. Conceptually, this is akin to "standing back" from the monitor or otherwise zooming out until the individual letters are not visible but the blocks of color are, and then measuring the relative noise or regularity of the resulting view. In this case, we measure the average bandwidth of the transformed signal in the horizontal and vertical directions.

Consider the code samples in Figure 4 and Figure 5. Both samples contain three comments, but the comments in the first are long relative to line length in the sample while the comments in the second are relatively very short. In terms of the DFT, we expect the shorter comments in the second sample to require more high frequency adjustment to represent than the longer comments, resulting in a larger bandwidth. We represent this mathematically by constructing matrices in which each cell corresponding to a comment character contains a 1 while all other cells contain a 0, then taking the two-dimensional DFT of the matrices. The results of this computation are shown in Figure 7. The central peak of Figure 7a is much narrower in the X direction than the central peak of Figure 7b, providing a visual intuition for the bandwidths of the respective signals. In the case of these two code samples, the computed bandwidth in the X direction is four times larger for Figure 5 than for Figure 4.

c) Alignment Features: We observe that humans often "line up" the syntactic elements of source code to make similar structure clear. For example, in a sequence of assignment statements the "=" operators and right-hand-side expressions are often placed in the same respective columns. To capture this intuition we again consider the source code as a matrix of characters to find regions in which tokens of the same type occurring on consecutive lines of code are aligned to the same column. Specifically, we look for a pattern consisting of more than two lines of source code in which a transition between whitespace and a particular token type or between two particular token types occurs at the same column in every such line. Note that it is possible for a single line to be part of more than one such pattern. For example, in the case described above where "=" operators and the right-hand-side expressions of consecutive assignments are aligned, our metric recognizes two alignment patterns, one for the operators and another for the expressions. We count the number and extent of these regions and include both as features in our model.

d) Natural Language Features: In addition to these visual features, we investigate the effect of the content of identifiers on readability. Using and producing natural language is viewed as increasingly important for understanding or documenting code (e.g., [26]). For readability, we look for identifiers in which the prefix or suffix is a recognizable English word. We also look for identifiers with underscore-separated terms and identifiers written in camel case. In both cases, we record the fraction of identifiers in which the terms are English words.

e) Shallow Syntax Features: Following previous work [13], we incorporate some syntactic features, such as counts of various punctuation characters, into our model. Beside their demonstrated explanatory power, these features provide a baseline against which to evaluate the relative explanatory power of our new features. That is, if we find that simple syntactic features predict most of a human's readability judgment, this would provide evidence against the utility of our proposed features. The syntactic features we use include line length, the count of distinct identifiers, and the frequency of keywords, numeric literals, and operators in the code sample.

f) Formal Descriptive Model: Since we have many interacting features that may be positively or negatively associated with readability, we must select a subset of features on which to base the model. We then construct our metric using logistic regression to learn the relative weights of the features. Note that unlike linear least-squares regression, which minimizes error, logistic regression finds parameters that maximize the likelihood of observing the sample values. We use the results of a large human study, detailed in the next section, to provide suitable sample values.

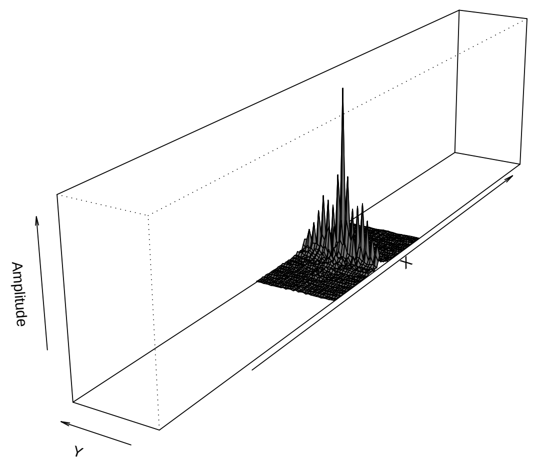
We use the wrapper selection approach [27] to identify a small subset of features that effectively describes the sample values. The wrapper approach trains a classifier on each subset of features it evaluates, ranking the subsets according to the accuracy of the trained classifier. For our overall readability metric, this approach identified 7 key features. This contrasts with the 25 features used by Buse *et al.* [13]. In some experiments we learn metrics for particular languages or subsets of users, in which case as few as four key features may be identified (see Section V-D). In all cases we use slightly more than the three features identified in Posnett *et al.*'s final metric [14].

IV. Human Study

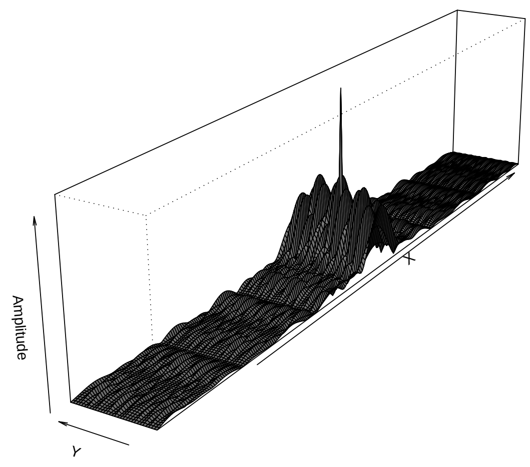
We conducted an IRB-approved web-based survey of over 5000 humans to determine ground-truth readability ratings for 360 examples of the languages in our study. Study participants were each shown twenty samples and asked to rate them on a 1–5 Likert scale from very unreadable (1) to very readable (5). The examples shown to an individual participant were randomized so that each participant saw distinct samples of varying lengths taken from Java, Python, and CUDA. We applied basic syntax highlighting to each example, assigning colors to token types as described in Section III. Participants could go back to review and change their answers to previous samples at any time before the end of the survey. After rating the code samples, participants were asked questions about their experience with programming in general and specifically with the programming languages used.

A. Sample Selection

All snippets used in our survey were drawn from open-source projects in the `SourceForge` repository. We used that website's search function to identify projects tagged as primarily using each of the languages in the study. We then selected the first 10 projects from the list of recently updated projects for each language (as of March 15, 2012). In cases where a project was incorrectly tagged and contained no source code in



(a) 2D DFT for comments in Figure 4.



(b) 2D DFT for comments in Figure 5.

Fig. 7: Amplitudes of the two-dimensional DFTs of the comments in two code samples. The DFT in (a) has fewer data points in the X direction because of the shorter lines in the code sample it represents. Note that the X-direction width of the central mound is greater in (b) than in (a). The corresponding bandwidths reflect this; the average bandwidth in (b) is over 4x larger than the bandwidth in (a).

the desired language, we skipped the project and selected the next in the list of search results. The projects used as sources of code samples are found in Table I. Overall, thirty projects spanning three languages and totaling over seven million lines of code were selected as sources for sample code.

For each language, we extracted short, medium, and long samples, corresponding to roughly 10, 30, and 50 lines each. For each language and length bound, we chose 40 samples uniformly at random from all available lines in source files anywhere in the project ending in the correct extension. We did not require the samples to contain grammatically well-formed definitions; instead, each sample could begin or end in the middle of a block or multi-line statement, so as to simulate the user experience of viewing a window of an arbitrary portion of code. The three languages, three sample lengths, and 40 samples of each yielded a total of 360 samples for our survey.

B. Survey Participants

We initially announced the survey to the members of a massively on-line class at Udacity, an on-line education provider similar to edX or Coursera.[1] The students were informed of the purpose of the survey and that no identifying information would be collected. The survey request was also posted by participants to the programming forum of `reddit`, a popular social news website,[2] and thus attracted participants from throughout a much wider, and more professional, community.

Ultimately, 5468 participants attempted the survey and provided at least one readability judgment. Of those, 2870 completed all 20 samples and 2681 provided answers to the final experience questions. Table II summarizes the participants' experience levels. The extensive breadth and depth of participation in our survey helps to ensure the generality of our model. For example, even if attention is restricted solely to participants with at least five years of industrial experience, our survey contains almost an order of magnitude more participants (1091) than the entirety of the participants used in previous work (i.e., 110 undergraduate and 10 graduate students [13]).

For a survey this large, small threats to validity can have a magnified effect. One potential concern was ballot stuffing (i.e., multiple submissions from the same IP address). Our logs and random samples indicate at most 2.9% of judgments may have resulted from such mischief. Similarly, a UI bug prevented some users of the Opera browser from submitting their experience answers after submitting their readability ratings. We estimate that at most 2.3% of all users were affected.

To put the size of this survey in perspective, a recent paper examined 3000 papers over the last ten years of six major software engineering conferences and found that most user evaluations involve 6–30 participants and that "over 45 participants" is "very large" [28, Fig. 11]. In fact, that work observed 0 user evaluations involving 46+ participants that included

[1] http://www.udacity.com/overview/Course/cs262

[2] http://www.reddit.com/r/programming/comments/tr5da/please_participate_in_the_university_of_virginias/

TABLE I: Sources for code samples used in the on-line survey.

Project	Language	LOC
BarraCUDA Fast Short Read Aligner	CUDA	4671
Cryptohaze	CUDA	9656
CUDASW++	CUDA	12255
GPU Autocorrelator	CUDA	1891
GPUMLib	CUDA	6385
libCudaOptimize	CUDA	2163
NiftyRec	CUDA	4262
Nifty Reg	CUDA	5677
Vocale	CUDA	5797
VolTK	CUDA	16
CUDA total		52773
Angry IP Scanner	Java	20289
Azuerus/Vuze	Java	896621
FreeMind	Java	108332
Liferay Portal	Java	5340077
PDF Split and Merge	Java	18963
SAP NetWeaver Server	Java	41385
SQuirreL SQL Client	Java	659723
Sweet Home 3D	Java	110597
TightVNC	Java	12418
Webmin	Java	9401
Java total		7217806
Docutils	Python	178200
Firebird	Python	36471
GNS3	Python	214256
Inkscape	Python	19878
MySQL for Python	Python	5098
Pidgin	Python	1087
qBittorrent	Python	4299
Scribus	Python	38382
Task Coach	Python	157750
Ubuntuzilla	Python	1651
Python total		657072
total		7927651

TABLE II: Survey participant self-reported experience.

Category	Median (yrs)	> 1yr	> 5yr	> 10yr
Overall	8	2598	1972	1242
Java	2	1896	646	247
CUDA	0	181	8	2
Python	1	1655	253	59
School	3	2118	522	28
Industry	3	1808	1091	655

both students and practitioners, making this evaluation novel in that regard.

No users were excluded from participating in the survey. All votes from all users are included in the analyses below, except where otherwise noted (e.g., when comparing industrial to academic participants, the relevant slices of the dataset are used). Our raw, anonymous study data is publicly available.[3]

V. Study and Experimental Results

In this section we empirically evaluate our readability metric with respect to the human judgments from our survey and with respect to external indicators of software quality. In particular, we address four research questions:

1) Does our metric predict human judgments? That is, does our metric agree with humans at least as well as they agree with each other?
2) How much do our novel features (visual, structural, linguistic, etc.) contribute to the success of our metric?
3) Does our metric agree with tool-reported defect density, an external notion of software quality?
4) Can our model illuminate readability differences between various groups (e.g., industrial vs. academic) or various languages (e.g., Python vs. Java)?

We now consider each research question in turn.

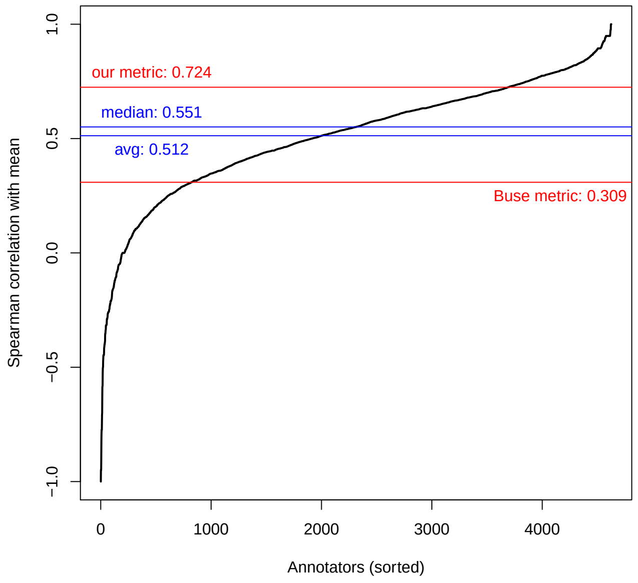


Fig. 8: Agreement with the average score for each code sample. Our metric agrees with the human ground truth at least as well as the human median and human mean agree with it (0.724 vs. 0.551 and 0.512). Previous work does less well. This figure excludes annotators who only rated one or two samples, since the Spearman correlation for those users would not be meaningful.

A. Agreement with Humans

In this subsection, we evaluate the ability of our metric to predict human judgments of readability. Because readability is subjective, humans are not in perfect agreement with each other. Using our large number of human judgments, we consider the average judgment for each sample to be its ground-truth readability. We can then compute inter-annotator agreement. We measure our metric's agreement with the ground truth and compare that to the average and median human agreement with the ground truth.

[3]http://dijkstra.cs.virginia.edu/projects/readability/results.zip

Figure 8 shows the Spearman correlation between the ratings given by each survey participant and the ground-truth ratings for all samples in the survey. Spearman correlation recognizes the similarity between the ordering of elements. Note that our metric has a significantly higher correlation with the ground truth than do the average or median user: that is, our metric agrees with humans at least as well as they agree with each other. Our metric also correlates with humans 2.3 times better than does the Buse metric as published [13], which produces ratings that have a much lower correlation with the ground-truth score. We conclude that our metric agrees well with user perceptions for large and small samples from a variety of projects in three languages.

One potential threat to the validity of such an experiment is overfitting (i.e., learning a metric that is too complex with respect to the data). The concern is essentially that the metric may learn the training data rather than learning generalizable features of the problem. One common technique used to measure the danger of overfitting is N-fold cross validation [29], in which the training set is divided into N groups and each group is held out as testing data for a metric learned on the $N - 1$ remaining groups. If the metric performance under cross-validation were different, that would suggest that the metric failed to learn general rules that apply to the held-out groups. We carried out 10-fold cross validation and found the performance change was approximately 1.2%. This, coupled with our use of at most twelve features in a logistic model, gives us confidence that the metric is learning general readability characteristics that allow it to effectively predict unseen data.

B. Visual and Linguistic Features

In this subsection we explain a portion of our metric's success in terms of the features introduced in Section III. We fold part of this evaluation into a direct comparison with previous work by retraining the Buse metric (which includes only shallow syntactic features) using our dataset.

Figure 9 depicts the performance of our metric and the retrained Buse metric for several interesting subsets of the data in our survey. Performance is calculated using the F-measure, the harmonic mean of precision and recall, a common metric in the domain of information retrieval [30]. Here precision is the ratio of the number of snippets that both humans and our metric rate as "more readable" (i.e., greater than the median ground-truth score) over the total number our metric rates more readable. Recall is the ratio of the snippets that both rate more readable over the total humans rate more readable. Our use of the F-measure to evaluate the performance of a readability metric follows similar uses in previous work (e.g., [13, Fig.9]).

In all cases, a metric using our visual and linguistic features outperforms a metric trained on the same data but not using such features. Over all available samples, the difference in performance was 5%. Given the established power of syntactic features for modeling readability, we thus conclude that our new features contribute significantly to the success of our metric. The gap in performance is greatest when particular languages are considered, where our approach outperforms others by 16–26%. This suggests that while the syntactic features included in the Buse metric are sufficient for an aggregate description of Java readability, they lack the expressiveness to describe the readability requirements of targeted domains.

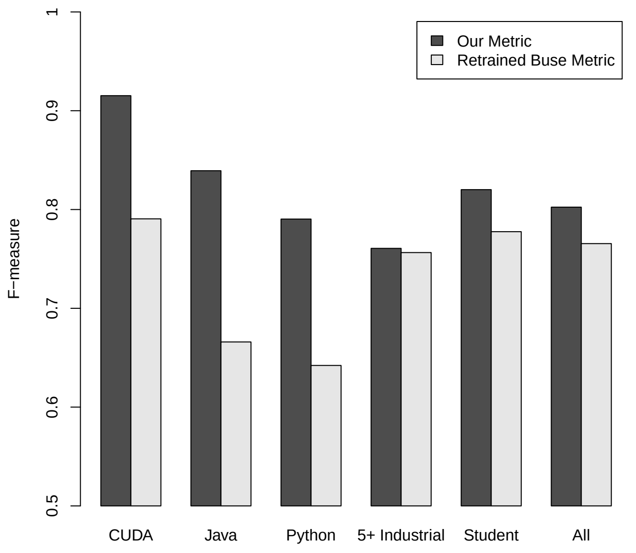


Fig. 9: Comparison of the performance of our metric against a metric based on the syntactic features proposed by Buse *et al.* but retrained on our larger datasets. Our metric outperforms the retrained Buse metric by 5% overall and by 16–26% on particular languages.

The results of cross validation highlight another notable difference between the features in our model and those in the Buse model. The performance difference between testing and training on the same data and using 10-fold cross-validation was 2× to 14× larger for the Buse metric than for our metric, with the largest differences observed on the language-specific slices. This suggests that the features in the Buse metric, while good for predicting student evaluations of Java snippets, do not capture the complexity of this more general situation, and that regression shows more signs of overfitting using the large number of available syntactic features.

C. Agreement with Defect Density

In this subsection we demonstrate that the established link between readability metrics and software quality [16] extends to our readability metric. Readability is a popular metric for use in software analytics. While our goal is to make a metric predictive of human judgments of readability, not to make a metric predictive of software quality, many have observed that similar metrics can be intertwined [31], [32].

We focus on one aspect of defect density. Even for

TABLE III: Percentage of methods implicated by FindBugs.

	Readability		
Benchmark	Low	High	Ratio
Angry IP Scanner	8.2%	1.8%	4.64
Azureus/Vuze	29.8%	14.1%	2.12
FreeMind	26.0%	18.2%	1.43
Liferay Portal	3.7%	2.5%	1.48
PDF Split and Merge	10.8%	3.8%	2.84
SAP NetWeaver Server	8.6%	19.8%	0.43
SQuirreL SQL Client	12.7%	4.1%	3.11
Sweet Home 3D	8.2%	3.8%	2.19
Tight VNC	10.4%	12.8%	0.82
Webmin	12.7%	3.8%	3.37
average	13.1%	8.5%	2.24

open source software with issue-tracking and version control software, it can be difficult to link defect reports to code changes [33], [34] or properly handle reassignments [35] or non-essential changes [36]. We thus employ the popular **`FindBugs`** [37] static analysis software as a proxy for defects. Intuitively, we seek to test the hypothesis that code rated as less readable is more likely to be flagged by **`FindBugs`** as containing a potential defect. Since **`FindBugs`** takes compiled Java bytecode as input, this experiment does not apply to CUDA and Python benchmarks. We consider every method containing at least one line that was mentioned in at least one **`FindBugs`** report to contain a reported defect. We then compute the readability for each method, and compare the readability of methods implicated by bug reports to the readability of those methods involved in no bug reports.

Table III separates the methods of each benchmark according to whether our tool assigned them high or low readability scores. For each category, the table lists the percentage of methods in that category that **`FindBugs`** implicated as potentially containing a defect. On average we find that methods with low readability scores are 2.24 times more likely to be flagged by **`FindBugs`** than methods with high readability scores. Note that **`FindBugs`** operates on compiled bytecode and does not look at syntactic features of source code during its analysis. In other words, **`FindBugs`** and our metric are based on potentially-independent signals in the software (compiled program semantics vs. visual and spatial source code features) but tend to agree on the same result. We thus gain confidence that the increased rate of defect reports among less readable methods reflects a true association between certain bugs and low readability.

D. Universal and Particular Aspects of Readability

Having established that our metric and features are highly predictive of human judgments of readability, we perform a series of comparative analyses on slices of our dataset. These controlled experiments allow us to tease apart which features best explain readability judgments made by various groups of humans or made on various sorts of software. We use the survey data to focus on judgments made by individuals with a particular level of experience in the language at hand.

Intuitively, we expect some of the features that most effectively characterize readable code to be different for different programming languages. Table IV shows our results, which support this intuition when we restrict attention to each language or subset of humans. Each feature's description is prefixed with its category, as described in Section III. Each feature's relative power is calculated using the ReliefF [38] method, which does not assume conditional independence. Higher values indicate a more predictive feature. Finally, each feature's direction of correlation with modeled readability is given. For example, in our full model, long lines decrease readability while our perceptual feature related to the visual shape of comments increases readability.

Not all features are universal across languages. For example, we find that line length is the most significant characteristic of source code affecting readability across all languages, as well as for Java specifically. However, when we consider only Python examples, line length does not play a role in the metric we learned. Instead, the most significant feature is the average number of identifiers per line, which is not a significant feature for either of the other two languages.

Table IV also highlights the differences in perceptions of readability among people with at least 5 years of experience in industry and people with school background, but no more than one year of experience in industry. We consider user ratings provided for all languages. As with the data set from all respondents in the survey, we find that line length is the most significant factor in predicting readability ratings. However, we find that whitespace also plays a significant role for the group with at least 5 years industry experience, whereas it does not among those with primarily academic experience.

We also investigated the effect of experience on perceptions of readability within single languages. In this experiment, we compare the responses of all participants on code samples from a single language against those of participants with experience in that language. (We do not conduct this experiment for CUDA because the general inexperience with CUDA prevented us from calculating a reliable ground-truth for people with CUDA experience.)

Our results, shown in the table, indicate a marked degree of consistency in the factors that describe readability. For example, the features that predict the Java readability ratings from the entire survey population by and large continue to do so for the subset with Java experience. Only one feature in the former model — the number of lines between instances of the same identifier — is missing from the latter model. The model learned from the experienced Java users data does include several structural features that are not recognized in the other model. We speculate that these patterns may be learned with increased exposure to typical examples in the language.

Similarly, for Python the syntactic use of identifiers and their linguistic content plays a significant role in our models for all responses to Python code examples, and to those from individuals with Python experience. The remainder of the models show a less pronounced consistency than in the Java experiment. However, we do observe that the visual impact of

TABLE IV: Selected features in our readability metric, ranked according to ReliefF values for each. The direction is of correlation with increased readability.

Category	Feature Description	Power	Direction
Full Model, All Code Samples (5468 users, 76741 ratings)			
syntax	line length	0.0319	-
syntax	long lines	0.0210	-
visual	operator area and total area	0.0174	-
structural	1D DFT of syntax	0.0056	-
visual	2D DFT of comments	0.0039	+
visual	string area to keyword area	0.0024	+
alignment	minimum alignment length	-0.0001	+
Java Samples (4956 users, 25684 ratings)			
structural	1D DFT of whitespace	0.0388	-
syntax	long lines	0.0349	-
syntax	lines between identifiers	0.0114	-
syntax	keywords	0.0040	+
structural	1D DFT of syntax	-0.0065	-
Java Samples, 1+ Years Experience (1881 users, 12565 ratings)			
structural	1D DFT of whitespace	0.0433	-
syntax	long lines	0.0273	-
syntax	keywords	0.0212	-
structural	operator to keyword tokens	0.0158	-
structural	1D DFT of identifiers	0.0034	+
structural	1D DFT of keywords	0.0025	+
structural	1D DFT of syntax	0.0008	+
Python Samples (4966 users, 25352 ratings)			
syntax	identifiers	0.0442	-
linguistic	identifier components	0.0141	-
visual	operator area to keyword area	0.0083	-
structural	operator to identifier tokens	0.0025	+
structural	1D DFT of syntax	-0.0021	-
Python Samples, 1+ Years Experience (1654 users, 10901 ratings)			
syntax	identifiers	0.0325	-
linguistic	identifier components	0.0257	-
structural	1D DFT of numbers	0.0099	-
visual	string area to keyword area	0.0053	+
5 Years Industry Experience (1091 users, 21738 ratings)			
syntax	long lines	0.0203	-
syntax	whitespace	0.0979	-
visual	comment area to total area	0.0077	+
structural	1D DFT of whitespace	0.0054	-
Student, Less Than 1 Year Industry (786 users, 15707 ratings)			
syntax	long lines	0.0296	-
structural	1D DFT of syntax	0.0095	+
structural	keyword to comment tokens	0.0085	+
structural	1D DFT of keywords	0.0077	-
visual	keyword area to comment area	0.0072	+
syntax	arithmetic	0.0048	-
syntax	numbers	0.0035	+
visual	operator area to string area	0.0011	+
visual	string area to keyword area	0.0000	+
structural	number to comment tokens	-0.0029	-

keywords is important to both models.

Of the features we considered, our results show that the only factor for readability that could be called universal is line length, and even that is not relevant to Python-only judgments. The readability of most subsets — different languages, different realms of experience — are best explained through features almost unique to those subsets.

However, while specific features are not universal, broad concepts are. For example, each of the feature sets we describe includes features related to the regularity of structural patterns within source code (here measured by one-dimensional Fourier transforms). Similarly, most of the feature sets include a component describing the visual area occupied by various colors on the screen. This supports our claim that our proposed visual and structural techniques help to generalize our notion of readability and provide a strong complement to simple surface level features.

E. Threats to Validity

Although our experiments suggest that our metric agrees with humans on a broad range of readability judgments, our results may not generalize. In this subsection we address a number of possible threats to validity.

a) Human study sample size: The accuracy of our metric depends on the accuracy of the ground-truth data provided by the thousands of volunteer participants in our on-line survey. This represents more than an order of magnitude increase over the number of participants used in previous models [13], [14]. Such a large number of respondents strongly mitigates the danger that any single perspective might happen to dominate our ground-truth readability scores.

b) Human study sample makeup: Another possible threat to validity is that the respondents do not represent the attitudes and preferences of the larger programming community. However, our survey included a large number of practitioners from both academia and industry, with a wide range of experience in both. The variety of experiences and backgrounds shown among the survey participants diminishes the danger of collecting a narrow sample of opinions. In particular, our study contains 1091 participants with more than five years of industrial experience, reducing the risk that our results would reflect the tastes and needs of only the academic community.

c) Code sample selection: In addition to the representativeness of the survey participants, our results depend on the representativeness of the code samples. An unrepresentative set of samples would restrict the range of feature values seen during the training phase and would limit our model's ability to generalize to other environments. To address this concern, we selected code examples from a large number of projects, in a variety of languages and sizes. To avoid human bias in selecting examples that match a particular individual's perspective, we automated the selection process, randomly choosing code examples from projects that had been selected systematically based on external activity rankings.

d) Overfitting: As discussed in Section V-A, the danger of overfitting the metric to the data exists in any application of machine learning. We performed 10-fold cross validation for every metric that we learned as a safeguard to detect cases of overfitting. The average difference revealed was less than 2%, suggesting that overfitting is not a significant threat and that our metric may generalize.

VI. Related Work

The two most closely-related areas of related work include other software readability metrics and software quality metrics.

To the best of our knowledge, the first automatic metric of software readability that was based on the results of a human study was developed by Buse *et al.* [13]. The Buse work demonstrated that it is possible to capture mechanically some notions of readability, a traditionally subjective judgment. However, their study included 120 student participants evaluating 100 short, Java-only snippets. In contrast, our survey features almost 50× more participants, who rated 3× as many long, medium, and short examples of code from three different languages. Their model measured characteristics of the character content in the source code, but did not consider visual or spacial structure in the on-screen representation.

Posnett *et al.* developed an improved readability model [14], based on Buse's survey data, which demonstrated significant agreement with human ratings using a simpler set of features than its predecessor. In addition, they were the first to identify a lack of generality in the Buse model: in particular, they noted its focus on 7-line snippets and difficulties with longer code samples. Their features focus on the textual complexity or entropy of the source code rather than its spacial or visual characteristics.

McCabe introduced the Cyclomatic complexity metric, which aims to quantify the decision logic in a piece of software [39]. Cyclomatic complexity is based on simple control flow graph features. While it is used in practice, previous work has suggested that Cyclomatic complexity largely measures the length of a function and is unlikely to correlate with coding defects (e.g., [17], [40]). Chidamber and Kemerer proposed an improved set of metrics for object-oriented programs [41]. Their metrics are theoretically grounded and capture notions such as the depth of the inheritance tree and the cohesion in methods. By contrast, our approach aims to model readability and human understanding — accidental complexities in the source code text — rather than inherent architectural complexity or control-flow decision logic.

Finally, code clone detection tools often make use of textual features to locate code that has been duplicated from one location to another (e.g., [42], [43], [44]). It is also possible to use searches for similar code to aid in development, and Lee *et al.* demonstrate a tool that aids development with efficient searches of a large corpus [45]. While our structural pattern and alignment features (see Section III) do capture some notions of self-similarity or repetition, our goal is very different from code clone detection or code search.

VII. Conclusion

Readability is one of the three most desired software analytics metrics by managers and developers, but of those three it is the least available by a factor of two [9, Fig. 4]. In this paper we seek to remedy that gap by proposing a general model for software readability. Although readability is fundamental to many aspects of software maintenance, existing models are based on shallow syntax and the views of relatively few students evaluating 7-line Java samples [13], [14], [16], which limits their generality to other situations.

We acknowledge that code is typically read on screens by humans and thus introduce visual, spatial and linguistic features into the domain of software readability metrics. We propose structural pattern features (intuitively capturing regular shapes using a 1D DFT), visual perception features (intuitively capturing the colored regions seen when one "stands back" from an IDE, using a 2D DFT), alignment features (intuitively capturing elements that have been "lined up" by humans to make structure clear, using token information), and natural language features (intuitively capturing words inside identifiers using dictionaries). We performed a large scale human study involving over 5000 participants — an order of magnitude larger than previous readability studies, and containing over 1000 people with at least five years of industrial experience. We targeted multiple languages (Java, Python and CUDA) and presented code samples of multiple, more indicative sizes.

Using our features and our human study data, we derived an automatic metric of software readability that is highly correlated with human values: it agrees with human judgments at least as well as they agree with each other. Our metric shows 2.3 times better agreement than a previously-published metric [13], [16], which we demonstrate does not generalize to multiple languages or code sample sizes. We find that a non-trivial 5% of the performance of our metric can be attributed to our visual, spatial and linguistic features. We demonstrate that our metric, in addition to agreeing with humans, also correlates with analysis-reported defects: less readable code is 2.2 times more likely to be flagged as buggy. Finally, our large survey allows us to tease apart aspects of readability that are universal and aspects that are language specific. For example, long lines generally negatively affect readability over all languages and user backgrounds that we investigated, while we found that users perceive whitespace and the natural language content of identifiers to be specifically important to the readability of Java and Python, respectively.

We have presented a model of readability that is significantly more likely to generalize than previous work. We hope that our large-scale human study and the availability of our dataset may encourage other researchers to consider more general user evaluations.

Acknowledgments

We are grateful to Raymond Buse for providing his readability metric tool and instructions for its use. We also thank David Stavens, CEO of Udacity, of allowing us to advertise this human study during an on-line class. We also warmly thank the thousands of anonymous people who participated for no direct reward.

References

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

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

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

[4] D. E. Knuth, "Literate programming," *Comput. J.*, vol. 27, no. 2, pp. 97–111, 1984.

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

[6] J. C. Knight and E. A. Myers, "An improved inspection technique," *Commun. ACM*, vol. 36, no. 11, pp. 51–61, Nov. 1993.

[7] F. Shull, I. Rus, and V. Basili, "Improving software inspections by using reading techniques," in *International Conference on Software Engineering*, 2001, pp. 726–727.

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

[9] R. P. L. Buse and T. Zimmermann, "Information needs for software development analytics," in *International Conference on Software Engineering*, 2012, pp. 987–996.

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

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

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

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

[14] D. Posnett, A. Hindle, and P. T. Devanbu, "A simpler model of software readability," in *Mining Software Repositories*, 2011, pp. 73–82.

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

[16] R. P. Buse and W. Weimer, "Learning a metric for code readability," *IEEE Trans. Software Eng.*, Nov. 2009.

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

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

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

[20] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *International conference on Software engineering*, 2005, pp. 284–292.

[21] P. A. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," *Empirical Software Engineering*, Nov 2005.

[22] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.

[23] G. D. Bergland, "A guided tour of the fast fourier transform," *Spectrum, IEEE*, vol. 6, no. 7, pp. 41 –52, july 1969.

[24] J. S. Lim, *Two-dimensional signal and image processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[25] J. Bendat and A. Piersol, *Random data: analysis and measurement procedures*. Wiley-Interscience, 1971.

[26] S. Rastkar, G. C. Murphy, and A. W. J. Bradley, "Generating natural language summaries for crosscutting source code concerns," in *International Conference on Software Maintenance*, 2011, pp. 103–112.

[27] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 12, pp. 273–324, 1997.

[28] R. P. L. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," in *Object-Oriented Programming, Systems, Languages and Applications*, 2011, pp. 643–656.

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

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

[31] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *International Conference on Software Engineering*, 2006, pp. 452–461.

[32] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *Empirical Software Engineering and Measurement*, 2007, pp. 364–373.

[33] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced? Bias in bug-fix datasets." in *Foundations of Software Engineering*, 2009, pp. 121–130.

[34] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *Foundations of Software Engineering*, 2011, pp. 15–25.

[35] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ""not my bug!" and other reasons for software bug report reassignments," in *Conference on Computer Supported Cooperative Work*, 2011, pp. 395–404.

[36] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *International Conference on Software Engineering*, 2011, pp. 351–360.

[37] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the conference on Object-oriented programming systems, languages, and applications*, 2004, pp. 132–136.

[38] M. Robnik-Šikonja and I. Kononenko, "Theoretical and empirical analysis of ReliefF and RReliefF," *Mach. Learn.*, vol. 53, pp. 23–69, 2003.

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

[40] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Softw. Eng.*, vol. 26, no. 8, pp. 797–814, 2000.

[41] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[42] E. Duala-Ekoko and M. P. Robillard, "Clone region descriptors: Representing and tracking duplication in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 1, 2010.

[43] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim, "Instant code clone search," in *Foundations of Software Engineering*, 2010, pp. 167–176.

[44] F. Rahman, C. Bird, and P. T. Devanbu, "Clones: what is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[45] M.-W. Lee, S. won Hwang, and S. Kim, "Integrating code search into the development session," in *International Conference on Data Engineering*, 2011, pp. 1336–1339.