The Journal of Systems and Software 82 (2009) 1793–1803

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Increasing diversity: Natural language measures for software fault prediction

David Binkley [a], Henry Feild [b], Dawn Lawrie [a,*], Maurizio Pighin [c]

[a] Loyola College Baltimore, MD 21210, USA
[b] University of Massachusetts, Amherst, MA 01003, USA
[c] Universitá degli Studi di Udine, Italy

ARTICLE INFO

Article history:
Available online 26 June 2009

Keywords:
Information retrieval
Code comprehension
Fault prediction
Linear regression models
Empirical software engineering

ABSTRACT

While challenging, the ability to predict faulty modules of a program is valuable to a software project because it can reduce the cost of software development, as well as software maintenance and evolution. Three language-processing based measures are introduced and applied to the problem of fault prediction. The first measure is based on the usage of natural language in a program's identifiers. The second measure concerns the conciseness and consistency of identifiers. The third measure, referred to as the QALP score, makes use of techniques from information retrieval to judge software quality. The QALP score has been shown to correlate with human judgments of software quality.

Two case studies consider the language processing measures applicability to fault prediction using two programs (one open source, one proprietary). Linear mixed-effects regression models are used to identify relationships between defects and the measures. Results, while complex, show that language processing measures improve fault prediction, especially when used in combination. Overall, the models explain one-third and two-thirds of the faults in the two case studies. Consistent with other uses of language processing, the value of the three measures increases with the size of the program module considered.

1. Introduction

This paper studies the application of natural language processing techniques to the problem of fault prediction. Detecting fault prone code early, regardless of software life-cycle phase, allows for the code to be fixed at lower cost; thus, a good fault predictor helps to lower development and maintenance costs. For example, cost savings may come from focusing testing-effort on certain parts of the software, restructuring code, or augmenting documentation. Further motivation comes from Koru and Tian who observe that "software products are getting increasingly large and complex, which makes it infeasible to apply sufficient reviews, inspections, and testing on all product parts given finite resources" (Koru and Tian, 2007), highlighting the need for good fault prediction.

A number of studies have found correlations between structural characteristics of software modules and problems, such as change or defect proneness (Bell et al., 2006; Fenton and Ohlsson, 2000; Gyimóthy et al., 2005; Kokol et al., 2001; Koru and Tian, 2007; Menzies et al., 2007; Munson and Khoshgoftaar, 1992). Example structural measures include lines of code, operator counts, nesting depth, message passing, coupling, information flow-based cohesion, depth of inheritance tree, number of parents, number of previous releases in which the module occurred, and number of faults detected in the module during the previous release (Bell et al., 2006; Ferenc et al., 2002). However, it has been observed that there is need for more sophisticated measures. For example, Nortel Networks and IBM engineers observe that the most troublesome modules are not the ones with the highest structural-measure values (Koru and Tian, 2007); thus, observing the need for more sophisticated techniques.

In addition to greater sophistication, in recent work with structural code measures, Menzies et al. argued that the particular set of measures used in fault prediction is less important than having a sufficient pool to choose from (Menzies et al., 2007). Diversity in this pool is important. For example, many existing measures are strongly correlated with lines of code. One avenue to improve fault predictors is the search for additional measures not correlated with those in the existing pool.

Until recently, the semantic information contained in the natural language of a program (in particular, its identifiers) has gone underutilized in software engineering (perhaps owing to the origin of many analyses in the compiler construction field). The measures considered herein augment those that use structural characteristics by incorporating the semantics of natural language. This complements the structural information used in most measures by providing an orthogonal view of the source code. One view of these

* Corresponding author.
E-mail addresses: binkley@cs.loyola.edu (D. Binkley), hfeild@cs.umass.edu (H. Feild), lawrie@cs.loyola.edu (D. Lawrie), maurizio.pighin@uniud.it (M. Pighin).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.06.036

measures is an attempt to capture the information and intuitive notion of *well written* code. While instructors have long advocated the use of *good* identifier names, it is language processing techniques that can quantify this value. To this end, two case studies show that the language-oriented techniques deserve future study in the challenging domain of software fault prediction.

Three natural language-oriented measures are studied in this paper. The first is the simplest: it measures the percentage of natural language used in a program's identifiers. The intuition is that identifiers composed of natural language words may be easier for an engineer to understand; thus, leading to fewer faults. The second measure is the percent of identifiers violating a variant of Deißenböck and Pizka's rules for *concise and consistent* identifiers (Deißenböck and Pizka, 2005). Ambiguity in the concepts associated with identifiers makes code harder to manipulate without introducing faults (in particular, those associated with concept misunderstanding). The third, referred to as the *QALP score*, is named after a project aimed at providing Quality Assessment using Language Processing (Lawrie et al., 2006). The QALP score measures correlations between the natural language use in a program's source code and its documentation. It thus attempts to quantify the notion of *well-documented code*.

To investigate the value of the three natural language-oriented measures in fault prediction, two case studies are considered – one using the open source program Mozilla and the other a proprietary program written for a business application in a mid-size enterprise. These studies assess the utility of the language-oriented measures in predicting fault-prone modules of source code. Of particular interest is the notion of diverse measures and their importance in fault prediction. Therefore, two models for each case study are presented. In one model, a single language-oriented measure is used in conjunction with other structural measures. In this case, the QALP score is used because it is the most complex of the three. The second model incorporates all three natural language-oriented measures.

The primary contributions of this paper are the following:

1. **Natural language measures.** First and foremost, the paper proposes the use of measures completely unrelated to structural aspects of source code as a means of improving diversity in the pool of fault prediction measures.
2. **Case studies.** It also considers two case studies that explore the usefulness of the proposed measures in fault prediction.

The remainder of the paper includes background information in Section 2. Then a description of the experimental setup of the two case studies is presented in Section 3. The two case studies are presented in Section 4, followed by a discussion of related work, implications of this research, and a summary in Sections 5–7.

2. Background

This section describes each of the proposed measures, providing motivation for their inclusion and describing their computation. All three measures use a common preprocessing step, word extraction, which is described first. Following the description of the measures is a brief overview of the two case-study subjects. Finally, a description of the statistical technique used is given.

2.1. Word extraction

Word extraction has two phases. The first extracts the identifiers from the source and the second splits them into their constituent words (Feild et al., 2006; Lawrie et al., 2006). The first phase is implemented as a simple lex-based scanner (essentially implementing a simple Island Grammar (Moonen, 2001)).

The goal of the second phase is to split the extracted identifiers into 'words'. Each word is a sequence of characters to which some meaning may be associated. The need for splitting comes from identifiers that are made up of multiple "*words*" fused together (*e.g.*, rootcause). Words are often demarcated by word markers (*e.g.*, using CamelCaseing or under_scores). For example, the identifiers spongeBob and sponge_bob both contain the demarcated words sponge and bob. Such words are referred to as *hard words*.

When words are not explicitly demarcated, a splitting algorithm is used to divide each *hard word* into its constituent words. One such algorithm is a greedy algorithm that recursively searches for the longest dictionary prefix and suffix of (the remaining part of) an identifier (Feild et al., 2006). For example, consider the code

```
/* Sponge Bob needs to be given a bath */
bath(spongebob);
```

The greedy algorithm retains the hard word bath, but decomposes the hard word spongebob into sponge and bob. Words that are identified by the splitting algorithm are referred to as *soft words* (*i.e.*, bath, sponge, and bob). Thus, a hard word is made up of one or more soft words. Soft words form the atomic entities used by the three measures.

2.2. Use of natural language

The first measure, *percent natural language*, is the number of unique soft words found in the dictionary divided by the total number of unique soft words found in the code. It is hypothesized that the more natural language words used in identifiers, the easier the code will be to understand. This in turn is expected to lead to fewer faults. To determine whether or not each soft word comes from a natural language, it is looked up in a dictionary. When multiple natural languages are found in the source code, multiple dictionaries are used. Finally, the precision of this measure is somewhat dependent on the accuracy of the splitting algorithm. The accuracy of the splitter used in the experiments is about 76% (Feild et al., 2006) when compared to a human oracle. In terms of the percent natural languages found in a program's identifiers, the imperfectness of the splitting algorithm is expected to cause slightly, but uniformly, inflate scores; thus, this inflation is not expected to have a significant effect on the studies' results.

2.3. Conciseness and consistency

The second measure is based on the percentage of identifiers that violate syntactic conciseness and consistency rules (Lawrie et al., 2006). Such violations potentially lead to confusion as to the concepts represented by identifiers and thus may make the code more fault prone. The rules are based on Deißenböck and Pizka's formal model for well-formed identifier naming (Deißenböck and Pizka, 2005).

Deißenböck and Pizka's rules include three requirements: two related to identifier consistency (involving homonyms and synonyms) and one related to identifier conciseness. These three are formalized as follows: an identifier *i* is a homonym if it represents more than one concept from the program (*e.g.*, the identifier file in Fig. 1a). Two identifiers *i1* and *i2* are synonyms if the concepts associated with *i1* have a non-empty overlap with the concepts associated with *i2* (*e.g.*, the identifiers file and file_name share the concept *file name* in Fig. 1b). The presence of homonyms or synonyms indicate inconsistent naming of concepts in a program and thus violate Deißenböck and Pizka's identifier consistency rule. Finally, an identifier *i* for concept *c* is concise if no concept less general than *c* is represented by another identifier. For example, the identifier position most directly corresponds to the concept *posi-*

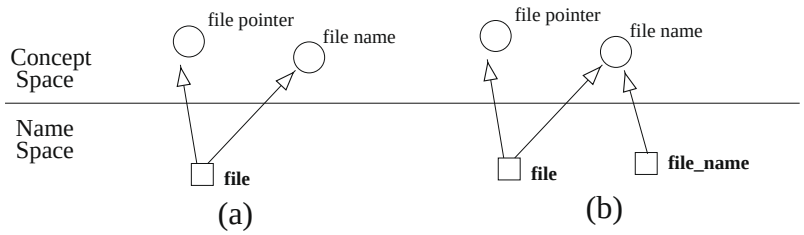

Fig. 1. Illustration of the two types of violations; (a) shows a homonym violation; (b) shows how a synonym violation is also introduced by the function that opens a file.

tion. It would concisely represent the concept *absolute position* provided that the program did not include any other position concepts that were less general then *position* (*e.g.*, *relative position*).

The syntactic variant of these rules does not have access to Deißenböck and Pizka's mapping from identifiers to concepts and is thus an approximation (Lawrie et al., 2006). The approach is based on identifier *containment*: an identifier is *contained* within another if all of its soft words are present, in the same order, in the containing identifier. For example, the identifier position is contained in the identifier relative_position. The key assumption of the syntactic approach is that a maximal identifier – an identifier not contained in any other – is associated with a unique concept. Thus, the set of concepts is approximated using these *maximal* identifiers.

The percentage of violations is calculated as the number of unique identifiers involved in violations divided by the total number of unique identifiers. Violations indicate faults when they cause confusion as to which concept is represented by an identifier. For example, code processing a sequence of absolute positions might include the variables position and index. An update to include relative position might add the variable relative_position, which introduces a violation as position no longer concisely denotes the concept of *absolute position*. In the update, position and relative_position are involved in a violation, thus 66.7% of the programs identifiers are involved in violations. In more realistic software this percentage is normally considerably lower. One might conclude violations occurring between identifiers in different scopes are harmless; however, unlike a compiler, engineers may (need to) remember an identifier even after it goes out of scope.

Similar to the percent natural language, imprecision in the splitter potentially impacts this measure by creating false positives. This is not a problem for dictionary hard words, which are not split. Inspection of the split hard words indicates that false positives are rare.

2.4. QALP score

The third measure, the QALP score, describes the similarity between a module's comments and its code. Borrowing the notion of *cosine similarity* from information retrieval (IR) (Manning et al., 2008), the QALP score considers each word as a separate *dimension* in an *n*-dimensional vector space. The *similarity* between two "documents" is then defined as the cosine of the angle between their two vectors. Given the nature of the vectors generated, the cosine of the angle is always between 0.0 and 1.0. The closer the cosine is to 1.0, the better the match.

When computing the QALP score, several techniques are applied to improve the efficiency of calculating cosine similarity. One employs a stop-list: a collection of words not thought to be relevant to any query. For example, in English, words such as '*the*' and '*an*' are stop-list words. In the context of this project, the comments are assumed to be written in natural language, so they are stopped using an appropriate natural language stop-list (for code with English comments, a standard English-language stop-list is used).

A second technique, *stemming*, reduces the dimensionality of the vector space by eliminating word suffixes; thus, ignoring the particular form of a word. For example, the stem of '*run*', '*running*', and '*runs*' is '*run*'. Since IR uses exact matches of words, stemming improves document matching.

The final technique weights the words using *term frequency- inverse document frequency* (*tf-idf*) (Salton and McGill, 1983), which provides a method for weighting the importance of a given word (called a *term*) to a document relative to the frequency of the term in the entire collection. The weighting takes into account two factors: term frequency in the given document and inverse document frequency of the term in the whole collection. In short, term frequency in a document shows how important the term is in that document. Document frequency of the term shows how common the term is across all documents. A high weight using *tf-idf* is achieved by a term that occurs much more than the average in a document but is rare in the entire collection.

To compute a QALP score, two separate documents are constructed from each module. One contains the comments of the module and the other the code of the module. The comments are stopped and stemmed. The code is stopped using a special stop-list that includes frequently used words not indicative of the concepts present in the code (Lawrie et al., 2006). For example, the stop-list for C includes keywords (*e.g.*, `while`), predefined identifiers (e.g., `NULL`), library function and variable names (e.g., `strcpy` and `errno`), and all identifiers that consist of a single character. Library functions and variable names are included on the stop-list because programmers do not have any control over their names.

After stopping, program identifiers are split using the method described in Section 2.1. The splitting recovers the soft words which are more likely to overlap with the comments. The resulting score is a real number between zero, no similarity, and one, 'perfect' similarity.

Given that there is some erroneous splitting, it is important to consider the impact it has on the QALP score. There are two potential effects of the introduction of an erroneous word by the splitting algorithm. First, the *tf-idf* weights of the other words will be slightly lower because the source code will appear to have more words in it. Overall, this is a minor effect. The other effect is if the split introduces a soft word that also appears in the comments, which would increase other overall QALP score. In practice, this situation is rare.

The QALP score is higher when the comments and the identifiers use the same vocabulary to describe concepts; thus, entirely self-documenting code with no comments would receive a low score of zero. In contrast, this measure is based on the assumption that overlap in the vocabulary between the code and comment help reduce ambiguity as such comments would mention the concepts being captured in well-named identifiers. Thus, less fault prone modules should receive a higher QALP score because they contain easily-comprehensible code, which is more maintainable. In addition, the attention required to comment code well may al-

low for the discovery of errors that would have been over-looked otherwise.

2.5. Study subjects

Two programs were used as case studies: Version 1.6 of the open source browser, Mozilla, and a proprietary program, referred to as $\mathscr{MP}$. These programs were chosen based on their size and the availability of defect (fault) data. In the case of Mozilla this data was extracted by examining the bug database maintained by Bugzilla, which assigns each bug to a set of classes (Gyimóthy et al., 2005). For $\mathscr{MP}$ fault data was collected in a similar in-house database in which two kinds of bugs are recorded: those revealed during the integration test phase and those generated by faults "in the field" during operation. For both, the database stores the module containing the fault, its date, a description of the fault, and the customer (if present). Table 1 summarizes fault statistics for both Mozilla and $\mathscr{MP}$.

In more detail, Mozilla is coded primarily in C++, with parts in C and Java. It contains 3,004,824 *LoC* (lines of code as measured by the UNIX utility wc) and 2,383,034 *SLoC* (source lines of code as measured by SLOCCount (Wheeler, 2005)). *SLoC* includes only non-comment, non-blank lines of code. By language, Mozilla contains 1,549,636 *SLoC* of C++, 829,814 *SLoC* of C, and 3584 *SLoC* of Java. The second case study, $\mathscr{MP}$, is written exclusively in C and has 454,609 *LoC* and 282,246 *SLoC*. It is a business application written in a mid-size enterprise.

The defect data used for Mozilla is similar to that used in the fault prediction work of Gyimóthy et al. from whom it was obtained. The data differs in that it is a later version and, thus, a slight refinement. The data includes the number of defects found in Mozilla's 3677 C++ classes. The data was gathered by examining the Bugzilla bug reports. Each of these include a list of lines from one or more files that had been modified to fix the bug. By examining the source code, each bug was assigned to one or more classes; thus, the final outcome was a bug count per class. If the bug fix changed more than one class then each affected class had its count incremented. Of the 3677 C++classes, more than half of the classes (2609 of them) contained no bugs while 1068 contained one or more bugs. The experiments reported herein exclude inner classes and classes generated on-the-fly. This left 3505 classes, of which 1041 contain one or more defects as is summarized in Table 1.

The program $\mathscr{MP}$ is coded primarily in C. It contains 456,931 *LoC* and 282,246 *SLoC*. The defect data is available for each of the 1161 C source files, which were tracked in a database. All defects have an associated date and module. Given that the program was developed by a team of 20 engineers, one programmer from the team was assigned to fix a particular defect. For this reason, many programmers worked on the same module over the 10–12 year life-cycle of the project.

The program runs under UNIX where it interfaces with an INFORMIX Database via an API named ALL-II. This has the effect of enforcing a rather fixed structure on the program where there are slots to call API functions such as add, update, read, delete, and lock. Slots are documented by an "environment team" and specifically commented by the programmer; this internal documentation is mandated by the development process.

Each $\mathscr{MP}$ file is of one of three fundamental types: interactive modules, report modules, and report activation. Interactive modules implement the interactive and transactional aspects of the system. Each breaks down in a similar pattern that includes sections to access the database, produce interactive forms, control data-entry, and perform data-elaboration. These modules are usually the more complex and larger.

Report modules implement batch and reporting functions. They are in general simpler, typically having three sections that access the database, produce the report output and perform data-elaboration. However, a small number are rather complex.

Report activation modules implement the forms that ask for parameters and activate a report. They are built from sections that access the database, and simple sections that query the user for parameters, control the parameters, and activate the appropriate batch or report module.

Table 1
Distribution of faults for the two programs studied.

	$\mathscr{MP}$	Mozilla
Program and fault statistics		
Modules	1161	3505
no Faults	191	2464
1–3	471	926
4–6	224	71
7–10	123	24
11 or more	152	20

2.6. Statistical tests

Linear mixed-effects regression models are used to analyze the data (Verbeke and Molenberghs, 2001). Such models easily accommodate unbalanced data, and, consequently, are ideal for this analysis. These statistical models allow the identification and examination of important explanatory variables associated with a given response variable.

The construction of a linear mixed-effects regression model starts with a collection of explanatory variables and a number of interaction terms. The interaction terms allow the effects of one explanatory variable to change depending upon the value of another explanatory variable. Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Some non-significant variables and interactions are retained to preserve a hierarchical, well-formulated model (Morrell et al., 1997). To provide a measure of model quality, a p-value and the coefficient of determination R^2 are reported with each model. The coefficient of determination can be interpreted as the percentage of the variation in the number of defects that is explained by the model.

In the mixed-effects models, computing a standard t-value for each comparison and then using the standard critical value increases the overall probability of a Type I error, the error of rejecting a "correct" null hypothesis (Ott and Longnecker, 2001). Thus, Bonferroni's correction is made to the p-values to correct for multiple testing. In essence each p-value is multiplied by the number of comparisons, and the adjusted p-value is compared to the standard significance level (0.05) to determine significance. Bonferroni's correction is chosen because it is a rather conservative test.

3. Experimental setup

This section describes the computation of the measures in the study. To focus on the value and viability of the language-oriented measures, only five measures are considered: *Percent Natural Language*, *Percent Violation*, the *QALP score*, and two structural measures: *LoC* (the total lines of code) and *SLoC* (the total source lines of code, that is, non-comment, non-blank lines of code (Wheeler, 2005)). Many existing structure measures are correlated with *LoC* and *SLoC* (Gyimóthy et al., 2005), making their inclusion a pragmatic choice as it factors in the major effects of structural measures while maintaining a focus on the natural-language based measures. Unsurprisingly, *LoC* and *SLoC* are highly correlated. Interestingly, they each bring something unique to the various models (if they did not then the backward illumination would

remove one or the other). In fact when both measures are included in a model, then they provide some insight into the value of non-code lines.

Computation is composed of three steps as depicted in Fig. 2. The first step (from Fig. 2a to b) breaks the source into modules. Natural modules include functions, classes, and files, but none of the computation for the language-oriented measures is tied to any given choice. The method of reporting the defect data determined the type of module used when computing the measures. Often, this step can be omitted as the source is physically laid out by module. $\mathscr{MP}$ source, for example, was organized by file, which matches the defect data. For Mozilla, some files contain multiple classes. In such cases, Step 1 gathers together the code associated with a single class into a single file used to hold the source of the particular module. The resulting modules (Mozilla classes and $\mathscr{MP}$ files) are sufficient to compute the structural measures. The appropriate UNIX utilities were then used to compute *LoC* and *SLoC*.

The second phase separates each model into comments and code as illustrated in Fig. 2c. This is done using *src2srcml* to insert XML tags throughout the code (Collard et al., 2003). Once marked-up, the source is then run through a simple fact extractor to separate the comments and code. The comments include those that occur immediately before a class or function definition and those that appear within the class or function.

The final phase applies language processing techniques to improve the efficiency and the accuracy of the language-oriented measures. In particular, as illustrated in Fig. 2d, the identifiers are split and the comments are stopped and stemmed. The five measures are computed for each module in the program.

Most of these are straight forward; the QALP score is computed by indexing the modules with the search engine Yari (Lavrenko and Croft, 2001), which can output the cosine similarity for each module. These measures and the number of faults in the module (obtained from the fault database) are used to produce the models discussed in the subsequent sections.

4. Results and discussion

The analysis first considers two preliminary models, one for each program. It then considers full models for each program. The preliminary models include only the QALP score and were built to determine if natural language measures had potential application to fault prediction. Their success led to the consideration of other language based measures; thus, the subsequent models include not only the QALP score, but also the Percent Natural Language and Percent Violations. The presentation of both the preliminary and final models concludes with a discussion comparing the two.

4.1. Preliminary Mozilla model

The statistical analysis first considers Mozilla. The initial linear mixed-effects regression model for predicting the defects in Mozilla begins with the explanatory variables QALP score, denoted as *qs*, *LoC*, *SLoC*, and their interactions. From this model statistically insignificant terms (those having a p-value > 0.05) are removed. The resulting model is significant ($p < 0.0001$) and explains just over a third, 34%, of the defects in the Mozilla data set.[1] Its coefficient of determination, $R^2 = 0.340$, is low.

[1] This model is different than the one reported in (Binkley et al., 2007) because modules that contained no comments were excluded from the data used to generate the model in that work. They are included here because the other natural language measures are not dependent on comments. The same sets of modules were used to generate all models reported herein.

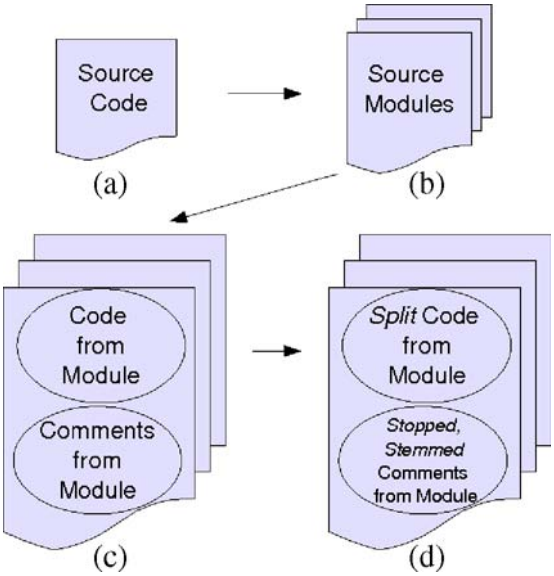

Fig. 2. The preprocessing steps.

$$defects = 0.091 - 0.0018\,LoC + 0.0051\,SLoC + 0.12\,qs - 0.0040\,qs \times SLoC \quad (1)$$

regrouping to gather the terms for QALP score yields

$$defects = 0.091 - 0.0018\,LoC + 0.0051\,SLoC + qs(0.12 - 0.0040\,SLoC). \quad (2)$$

The model shows two effects. First, *defects* increases with *SLoC* at a rate of about 1 defect for every 200 source (non-comment, non-blank) lines of code. However this is tempered by the inverse influence had by *LoC*. Because *LoC* is composed of *SLoC* and blank and comment lines, this indicates the blank lines and comments reduce defects.

From the point of view of this study, the far more interesting part of the model is the retention of the QALP score. This indicates that the QALP score plays a role in the prediction of faults. Of prime interest is the coefficient of the QALP score: $0.12 - 0.0040\,SLoC$. For higher QALP scores to be associated with lower defects, this coefficient has to be negative. In a simple model, free from interactions, this coefficient would be a constant and, if less than zero, the model would exhibit the desired correlation. In the presence of interactions, the value of the coefficient depends on other explanatory variables. Considering the coefficients in Eq. (1), as *SLoC* increases, the coefficient of *qs* grows increasingly negative. Thus, for larger modules, QALP score performs better. In this case the coefficient is negative when *SLoC* is greater than 30 lines. This is true for 2503 of Mozilla's 3505 modules. For the remainder, the coefficients are positive. Modules having less than 30 lines are too short for the QALP score to be a good predictor of faults.

4.2. Preliminary $\mathscr{MP}$ Model

The statistical analysis of $\mathscr{MP}$ begins with the same set of explanatory variables as that of Mozilla. However, the final model (after removal of statistically insignificant terms) has a substantially higher coefficient of determination.

$$defects = -1.94 + qs(-3.5 + 0.53\,LoC - 0.92\,SLoC) + 0.056\,LoC - 0.058\,SLoC. \quad (3)$$

The model's coefficient of determination ($R^2 = 0.612, p < 0.0001$) indicates that it explains just over 61% of the variance in the

number of defects. The influence of *LoC* and *SLoC* is similar to that for the model for Mozilla. The model also indicates that the QALP score plays a role in the prediction of faults, but its effect is more complicated.

Considering the coefficient of *qs* in Eq. (3), as *SLoC* increases, the coefficient of *qs* grows increasingly negative. A simplistic reading of this is that for larger modules, QALP score performs better. However, the coefficient for *LoC* in Eq. (3), has the opposite interpretation: as *LoC* grows the coefficient grows increasingly positive.

Taken together, these two make interpreting the effect difficult. One approach is a graphical interpretation, as shown in Fig. 3, where *LoC* is graphed against *SLoC*. As *LoC* is bounded from below by *SLoC*, no point can occur below the solid 45 degree line. The area between the 45 degree line and the upper line delineates the region in which the coefficient of *qs* $(-3.5 + 0.53\,LoC - 0.92\,SLoC)$ is dominated by $-0.92\,SLoC$. In this region, the coefficient of *qs* is negative (the region has a slight downward shift to account for the constant −3.5). Above the shaded region the term $+0.53\,LoC$ dominates and *qs* has a positive coefficient. The points in the graph represent the actual values for the modules of $\mathcal{MP}$. Of these 248 (21%) lie above the cutoff line; thus, for the majority of the data (79% of the time) the coefficient of *qs* is negative.

A second method for interpreting models with multiple interactions uses a quantitative approach. The approach considers several values for one of the variables. In this case the ratio of *LoC* to *SLoC* is considered. Typical values to use include the mean together with the low and high end of the 95% confidence interval around the mean. For the model in Eq. (3), these yield

$LoC = 1.665\,SLoC$ (lower bound),
$LoC = 1.676\,SLoC$ (mean),
$LoC = 1.687\,SLoC$ (upper bound).

Substituting these values in for *LoC* of Eq. (3) yields the following three models.

Using the lower bound

$$defects = -1.94 + qs(-3.5 - 0.038\,SLoC) + 0.0352\,SLoC.$$

Mean

$$defects = -1.94 + qs(-3.5 - 0.032\,SLoC) + 0.0359\,SLoC.$$

Using the upper bound

$$defects = -1.94 + qs(-3.5 - 0.026\,SLoC) + 0.0364\,SLoC.$$

The positive coefficient of *SLoC* at the end of each equation supports the unsurprising result that an increase in module size (as measured in the non-comment, non-blank lines of code) brings an increase in defects. In this case at the rate of about one defect per 30 *SLoC*. Due to the interaction with *qs*, this number should be taken as a rather rough estimate. The coefficient of *qs* in all three equations is negative; meaning that QALP score has the desired slope. Furthermore, as module size increases, this coefficient grows increasingly negative; thus, QALP scores are better predictors for larger modules.

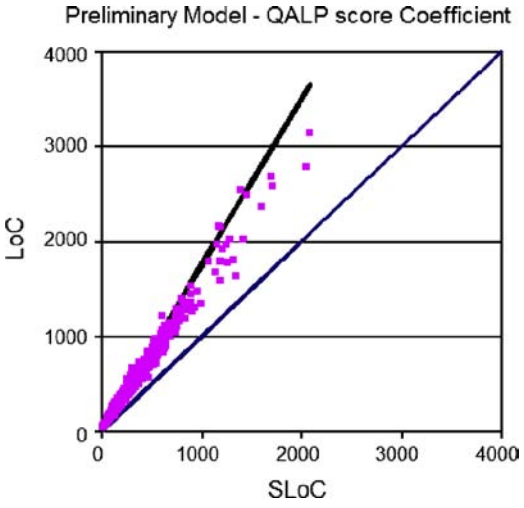

Fig. 3. Break even point for coefficient of QALP score in $\mathcal{MP}$'s preliminary model.

4.3. Complete Mozilla model

The complete statistical analysis of Mozilla starts with the explanatory variables *Percent Natural Language*, abbreviated *pnl*, *Violations*, abbreviated *v*, *qs*, *LoC*, *SLoC*, and their interactions. The final linear mixed-effects model, which is statistically significant ($p < 0.0001$) includes the three natural-language measures. It explains just over 37% of the variation in the number of defects (the coefficient of determination, R^2, is 0.372), which is an improvement from the 34% of the preliminary model. The model also includes a number of interactions involving *LoC* and *SLoC*, making interpretation challenging. Regrouped to focus on the natural language measures, the final model is

$$\begin{aligned} defects = {} & -0.057 - 0.001\,LoC + 0.007\,SLoC \\ & + qs(-0.12 - 0.0023\,SLoC) \\ & + pnl(0.34 - 0.00061\,SLoC) \\ & + v(-1.7 - 0.0075\,LoC + 0.024\,SLoC). \end{aligned} \quad (4)$$

In the first line of the equation, it is perhaps easier to understand the involvement of *LoC* and *SLoC* if *LoC* is replaced by "*SLoC+bc*" where *bc* measures the number of blank and comment lines. The second and third terms can then be replaced by "$0.006\,SLoC - 0.001\,bc$." Thus, larger modules include more defects while white space and internal documentation (presumably up to some limit) improve code quality.

Using the regrouped model, each natural-language measure is now considered in turn. As with the preliminary models, the desired sign of the coefficient for the *qs* is negative because a higher QALP score is expected to be associated with fewer faults. In the model, the coefficient is always negative and grows increasingly negative as size, *SLoC*, increases. Thus the value of *qs* increases with larger modules.

The desired sign of the coefficient for *pnl* is also negative because increased use of natural language in identifier names leads to fewer faults. In the model the coefficients are negative when the term involving *SLoC* dominates, which starts when *SLoC* is greater than 55 and increases as *SLoC* grows. In the source for Mozilla, this is true for 2012 (57.4%) of the 3505 modules.

Finally, violations of the conciseness and consistency rules are expected to accompany naming confusion and thus increase the likelihood of faults. Therefore, the coefficient of *v* $(-1.7 - 0.0075\,LoC + 0.024\,SLoC)$ is expected to be positive. Unlike *qs* and *pnl*, this coefficient includes competing influences from *LoC* and *SLoC*. As in the preliminary $\mathcal{MP}$ model, one approach for simplifying this coefficient is to consider the mean and bounds of the 95% confidence interval. In doing so, the coefficient of *v* is positive when *SLoC* is greater than 148, 149, and 150 lines for the lower bound, mean, and upper bound, respectively. This is true for 1100 (31.4%) of Mozilla's modules. In summary, the use of all three natural language measures suggests that the three capture different aspects of the code in their fault predicting ability.

4.4. Complete $\mathcal{MP}$ model

The complete statistical analysis of $\mathcal{MP}$ starts with the explanatory variables *pnl*, *v*, *qs*, *LoC*, *SLoC*, and their interactions. The final

linear mixed-effects model, which is statistically significant ($p < 0.0001$), includes the three natural-language measures. It explains just over 63.3% of the variations in the number of defects, which is only a mild improvement over the 61.4% of the preliminary model. The model again includes a number of interactions involving *LoC* and *SLoC*, making interpretation challenging. Regrouped to focus on the natural language variables, the final model is

$$\begin{aligned} defects = {} & -2.2 - 0.11\,LoC + 0.13\,SLoC + qs(-76 + 0.77\,LoC \\ & - 1.2\,SLoC + 255\,v) + pnl(1 - 0.14\,LoC + 0.22\,SLoC) \\ & + v(-0.22 - 0.038\,LoC). \end{aligned} \quad (5)$$

The direct influences of *LoC* and *SLoC* are similar to the other models and, as in the proceedings cases, their coefficients should serve only as rough estimates because of the interactions that *LoC* and *SLoC* have with the other explanatory variables. In addition, Each of the three natural language measures is included in the final model.

For the QALP score, the coefficient includes an interaction with one of the other natural language measures. Initially ignoring this term (involving v), the coefficient is very similar to that from the preliminary model. The constant increases from −3.5 to −76.0, which, similar to the Mozilla model, indicates improved predictive ability of the QALP score with larger modules. Repeating the two analyses from Section 4.2, Fig. 4 shows the range in which QALP score has the desired negative coefficient. The complete model does a better job of predicting the faults. In this case, only 26 points are above the upper cutoff line; thus 98% of the time the coefficient of qs is negative as desired. Finally, the ignored term involving v indicates that violations of the conciseness and consistency rules have a negative effect on the influence of the QALP score. In other words, naming confusion not only is expected to impeded programmers, but it also impedes the prediction ability of QALP score.

For $\mathscr{MP}$, *pnl* has the desired negative coefficient when $1 - 0.14\,LoC + 0.22\,SLoC$ is less than zero. As the structure of this coefficient is similar to that of the QALP score, a similar analysis is appropriate. One key difference is in the sign of the two program size measures. Thus in the graphical analysis, the desired outcome is to have all the points lie *above* the line. As can be seen in Fig. 5, a considerable number of points (just over 37%) lie *below* the cutoff line. Numerically, using the lower bound, mean, and upper bound of the ratio of LoC to SLoC, the coefficient is negative when SLoC is greater than 130, 109, and 94 lines, respectively. Using the mean, 828 (71%) of $\mathscr{MP}$'s modules are larger than 109 SLoC.

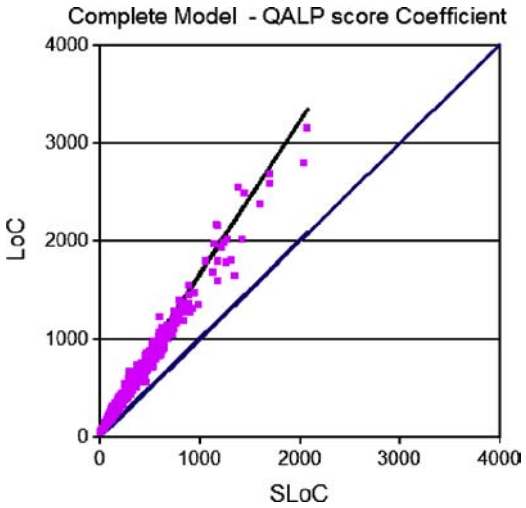

Fig. 4. Break even point for coefficient of QALP score using the $\mathscr{MP}$'s complete model.

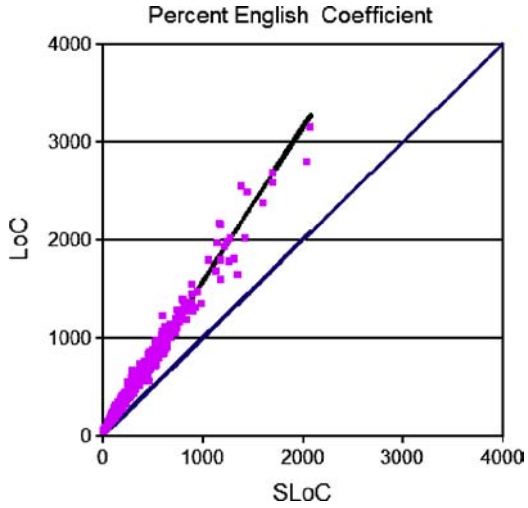

Fig. 5. Break even point for Percent Natural Language's coefficient in $\mathscr{MP}$'s complete model.

Finally, violations of the conciseness and consistency rules are expected to lead to confusion and thus more faults. Therefore, the coefficient for v is expected to be positive. The coefficient shown in Eq. (5) is always negative. However, the model places the interaction between qs and v with the coefficient of qs. Rewriting it in terms of v yields the coefficient $-0.22 - 0.038\,LoC + 255\,qs$. For sufficiently large QALP scores this coefficient is positive. Using the mean QALP score in the data set of 0.023, rule violations have the expected results only for modules less than 130 *SLoC* which is true for 461 modules or about 40%.

4.5. Discussion

The discussion first considers the models for Mozilla and $\mathscr{MP}$, which focuses on each program independently before relating the two. The discussion concludes with a statistical comparison of the different measures used in the models. To begin with, an inspection of the code for Mozilla generated three relevant insights as to why the natural language measures proved less effective at predicting faults. First, many of the classes had few, if any, reported defects and few comments. Both of these tend to obscure the prediction ability of any measure. For example, the complete absence of comments produces a QALP score of zero. When these zeros occur in classes with a wide range of faults, they produce statistical 'noise', which interferes with the ability of the statistical techniques to find correlations.

Second, many of the comments that did exist were either used to make up for a lack of *self documenting code* or were *outward looking*. In the first case, the code uses identifiers that are not easily understood out of context, so comments are required to explain the code. In the second case, comments are intended for users of the code and thus ignore the internal functionality of the code. In both cases, the code and comments have few words in common, which leads to a low QALP score. For example, the code snippet in Fig. 6 shows an example of both types of comments. This snippet determines whether there is anything other than whitespace contained in the variable mText. Unfortunately, it is not clear from the function name, IsDataInBuffer, that it is simply a whitespace test. The first comment informs the reader of this; thus, the comment is compensating for a lack of self-documentation. The second comment is an outward looking comment, reflecting on the

```
// Don't bother if there's nothing but whitespace.
// XXX This could cause problems...
if (! IsDataInBuffer(mText, mTextLength))
break;
```

Fig. 6. Mozilla-1.6 Example code snippet.

implications that the local code segment may have on the system as a whole.

The third insight follows from Mozilla being open source, which means that it includes many different coding practices and styles (Koru and Tian, 2007). This is evident when inspecting samples of the code where identifier naming and general commenting are not done in a systematic fashion.

Next inspecting the source for $\mathcal{MP}$ uncovered one notable difference to that of Mozilla. This is the modularization of the $\mathcal{MP}$ source. Each $\mathcal{MP}$ module (*i.e.*, each file) contained many short, well-commented functions. In addition, the comments include those that are *inward-looking* (*i.e.*, they refer to the functionality of the code). These characteristics arose from very strict programming rules adopted by the software group. Each module, independent of its functionality, was built starting from a fixed skeleton into which the engineer inserted code and comments. A team manager directed the project, and a team of three to four senior engineers operated on the framework by specifying the set of libraries at the disposal of the programmers, the skeleton of the modules, and the operating environment. This gave rise to a strict structure for each module in terms of comments, variable declaration (often done with predefined macros), and having the same kind of functions in modules of the same type (*e.g.*, those dealing with the GUI all had the same structure independent of the particular data-set they managed). In addition, the structure of general comments were specified and were mandatory for principal functions.

Comparing the models for each program, three points are highlighted: first, an observation made when comparing the models for Mozilla and $\mathcal{MP}$, second, a comparison of the source code for the two programs, and finally, the two separate 'kinds' of comments that appear in the code. To begin with, the observation deals with cumulative defects.

Previous empirical studies have validated the 80:20 principle, which states that a large majority (around 80%) of problems (*i.e.*, changes or defects) are actually rooted in a small proportion (around 20%) of the code (Fenton and Ohlsson, 2000; Porter and Selby, 1990; Tian and Troster, 1998). Furthermore, Boehm and Basili back this up noting that "Studies from different environments over many years have shown, with amazing consistency, that between 60% and 90% of the defects arise from 20% of the modules, with a median of about 80%" (Boehm and Basili, 2001).

Using this as a guideline, Bell et al. assess their fault prediction model by ranking files based on the model's prediction and then selecting the top 20% of the ranked files (Bell et al., 2006). This is a way in which to utilize the information provided by a fault predictor. They report that the files in the top 20% contained 71.1% of the faults. However, this is not a complete picture of the approaches' performance as it does not consider the percentage of faults in the top 20% of files when ranked by *actual* faults. This percentage represents the best performance that a fault predictor can hope to achieve.

Following Bell et al. the top of Fig. 7 shows the cumulative percent of faults when modules are ranked using the *predicted* number of faults (gray line) from the fitted models in Section 4 and the *actual* number of faults (black line). This shows the quality of the models. The gap between these two curves (shown graphically in the lower part of Fig. 7) captures the room for improvement in the fitting. Comparing the charts illustrates the superiority of the model for $\mathcal{MP}$.

Looking at the 20% point in the upper graph for Mozilla, the top 20% of fault-containing modules include 83% of the faults (thus, Mozilla is almost right in-line with the 80:20 rule). Graphically, when sorted on predicted faults based on the fitting, the top 20% of modules include 61% of the faults. Thus, there is significant room for improvement in the fitting.

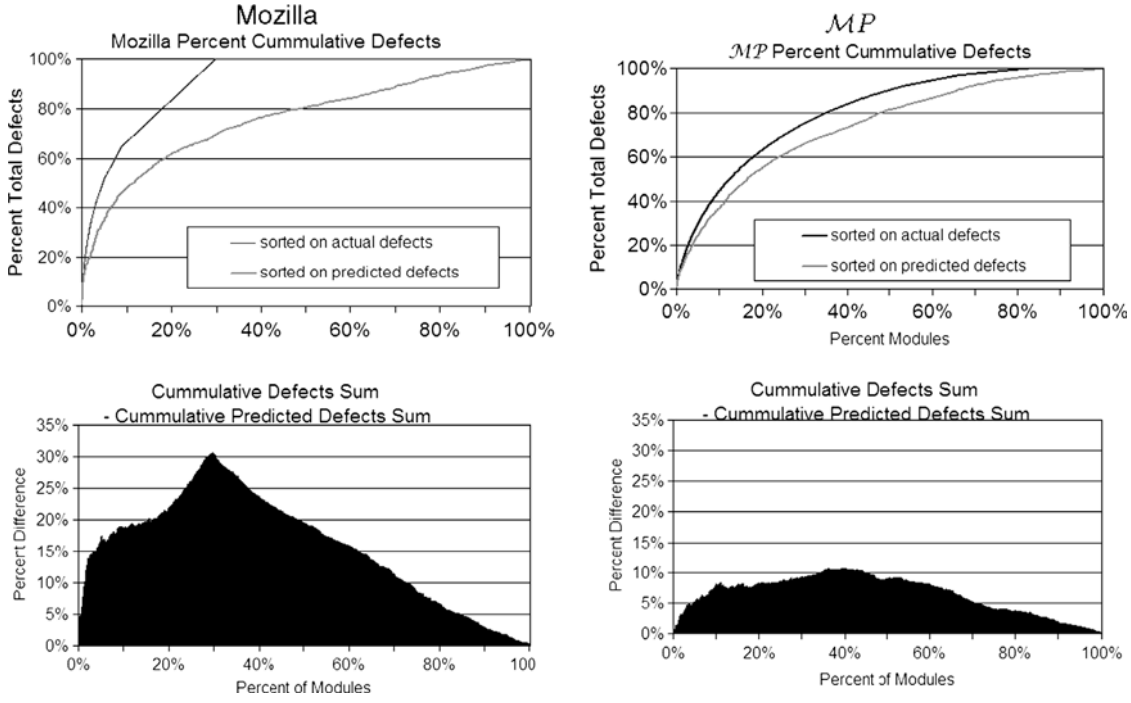

Fig. 7. Cumulative faults.

In comparison, the upper graph for $\mathscr{MP}$ shows the top 20% of fault-containing modules include 63% of the faults (thus the distribution of faults is more uniform for $\mathscr{MP}$). When sorted on predicted faults based on the fitting, the top 20% of the modules include only 55% of the faults. While this is significantly less that the 80:20 rules would predict, it is quite good considering that 63% is the best that could be attained (the upper black line). In other words, the model generated for $\mathscr{MP}$ only 'misses' 8% of the possible faults it could find at the 20% point.

A similar analysis was performed on the preliminary models. At the 20% cut-off 55% of Mozilla's faults were found and 54% of $\mathscr{MP}$'s faults. Given that there is a 6% improvement for Mozilla while only a 1% improvement for $\mathscr{MP}$, the variables *Percent Natural Language* and *Percent Violation* are more informative for Mozilla. This may indicate that variable names are more indicative of quality in an open-source environment; however, a wider study would be needed to identify such a trend.

However, given that Mozilla's model is a comparatively poor predictor of faults, a comparison of the source code for $\mathscr{MP}$ and Mozilla was performed. This comparison revealed that the environment in which $\mathscr{MP}$ was written includes strict style guidelines that encourage a kind of uniformity across all of the source code, giving the engineers less freedom in writing code, in writing comments, and in organizing modules. In contrast, the development of open-source software includes "a spectrum of processes from undefined and flexible processes to some extent defined and controlled processes among open-source projects" (Koru and Tian, 2007). Consistent with this, Mozilla showed evidence of a diversity of programmers and programming styles.

Finally, source inspection revealed that comments can (roughly) be divided into two types: *inward-looking* and *outward-looking*. In the models for $\mathscr{MP}$, the QALP score benefits from the presence of *inward-looking* comments. In contrast, the absence of such comments in Mozilla leads, in part, to an inferior model for Mozilla. Together these observations suggest two things: first, a need for increased inward-looking documentation (something that is advocated by Literate Programming (Bentley and Knuth, 1986) and Intentional Programming (Cordes and Brown, 1991)), and second, the QALP score would benefit from the incorporation of techniques that better assess the value of outward-looking comments.

The models discussed above are necessarily limited by the attributes they consider. Given that the purpose of this work is to explore the possibility of diversifying the pool of available measures, in this case focusing on natural language measures, it is interesting to consider the correlation between different pairs of measures. The correlation results for Mozilla appear in Table 2 and those for $\mathscr{MP}$ appear in Table 3.

These two tables indicate similar information, which given the differences in the two programs, seems to indicate a wider trend. First, unsurprisingly *LoC* is highly correlated with *SLoC*. Second, each of the three natural language measures have a low correlation with all of the other measures. This means that not only are they providing different information than structural measures, they are also providing different information from each other. This indicates their diverse nature as fault prediction measures. Third, the natural language measures have a much lower correlation with faults than the structural measures. This indicates that they would not be good measures in isolation. However, given their statistical significance in the models, they do provide different and meaningful information not currently captured by the structural measures, such as *LoC*.

Finally, it should be noted that some of the differences observed between the two case studies could have resulted from other factors not directly considered. One such example is programming paradigm. Another is program development history. In addition the reliability of the models is dependent on the completeness and consistency of the defect data (Koru and Tian, 2007).

Table 2
Correlation matrix for Mozilla.

Variables	*LoC*	*SLoC*	QALP score	% Natural language	Violations	Faults
LoC	1.000	0.994	0.218	0.058	0.010	0.551
SLoC	0.994	1.000	0.203	0.054	0.005	0.563
QALP score	0.218	0.203	1.000	0.219	0.049	0.069
% Natural language	0.058	0.054	0.219	1.000	0.090	−0.006
Violations	0.010	0.005	0.049	0.090	1.000	0.016
Faults	0.551	0.563	0.069	−0.006	0.016	1.000

Table 3
Correlation matrix for $\mathscr{MP}$.

Variables	*LoC*	*SLoC*	QALP score	% Natural language	Violations	Faults
LoC	1.000	0.991	0.169	−0.237	0.272	0.700
SLoC	0.991	1.000	0.168	−0.217	0.285	0.650
QALP score	0.169	0.168	1.000	−0.115	0.022	0.074
% Natural language	−0.237	−0.217	−0.115	1.000	−0.070	−0.216
Violations	0.272	0.285	0.022	−0.070	1.000	0.092
Faults	0.700	0.650	0.074	−0.216	0.092	1.000

5. Related work

This section considers five recent projects in the area of fault prediction and then describes a study of faults in general. First, Gyimóthy et al. describe the calculation and validation of a collection of object-oriented metrics for fault-proneness detection (Gyimóthy et al., 2005). Many of these metrics were originally proposed by Chidamber and Kemerer Chidamber and Kemerer (1994). They evaluate the metrics by comparing fault predictions against the defects extracted from the Bugzilla database using four assessment methods (*e.g.*, one method used was based on machine learning). The methods all yield similar results. They do note the need for measures not correlated with *LoC*. As shown in Tables 2 and 3, the language-oriented measures are not correlated with *LoC*; thus, it would be interesting to include them in the predictors generated by Gyimóthy et al.

Second, Koru and Tian show that the top modules in change-count rankings and those with the highest measurement values are different (Koru and Tian, 2007). The authors use change count in preference to defect count as they reported several issues when collecting defect data, such as completeness, consistency in data collection, and problems with mapping defects onto modules. They observe that, at the significance level of $\alpha = 0.01$, the top-change modules are neither the top-measurement modules when identified by ranking nor the top measurement modules when identified by a voting mechanism. Furthermore, using clustering to partition modules into those with similar number of changes, they again observe that the high-change modules are not the modules with the highest measurement values. (The high-change models do have fairly high measurement values.) That structural measures alone do not detect the top-change modules, suggests the need for non-structural measures, such as natural language measures.

Third, Bell et al. build a fault predictor based on file characteristics that can be objectively assessed: LoC, whether this was the first release in which the file appeared, how many previous releases the file occurred in, how many faults were detected in the file during the previous release, etc. (Bell et al., 2006). Based on these characteristics, they build negative binomial regression mod-

els to predict files that are likely to contain faults in the next release. Such models are an extension of linear regression designed to handle outcomes for non-negative integers. The advantage of using such a technique is that it allows for some degree of additional variability in fault counts that is not explained by any of the available predictor variables. The use of more sophisticated statistical modeling, as done by Bell et al., partially motivated the use of linear mixed-effects regression models in this study of language-oriented measures.

Fourth, Menzies et al. report that *how* the attributes are used to build a fault predictor is much more important than *which* particular attributes are used (Menzies et al., 2007). They build several predictors using a variety of techniques all starting from the same set of measures. Many of the measures bring similar information to a model and thus different techniques often choose different subsets of the measures, while achieving similar results. The authors note the value that diversity brings to the set of measures. Again, the natural language measures, not being correlated with the structural measures, would make interesting additions.

Finally, Marcus et al. introduce the non-structural measure *Conceptual Cohesion of Classes* for fault prediction (Marcus et al., 2008). This measure is calculated as the average of the cosine similarities between all the pairs of methods in a class. The cosine similarity is computed from the latent semantic indexing space generated from comments and identifiers of the methods. Identifiers are split into hard words. Three case studies are performed to show that the inclusion of this new measure is a better predictor of faults, which further supports the need for a more diverse pool of measures for fault prediction. In conclusion Boehm and Basili report on where and when bugs are found and how many resources are consumed to fix them. "Factors affecting the percentage of defects caught include the number and type of peer reviews performed, the size and complexity of the system, and the frequency of defects better caught by execution, such as concurrency and algorithm defects. Our studies have provided evidence that peer reviews, analysis tools, and testing catch different classes of defects at different points in the development cycle. We need further empirical research to help choose the best mixed strategy for defect-reduction investments." (Boehm and Basili, 2001). In the future it would be interesting to investigate whether natural language measures correspond to a particular kind of error, for example, those caught by code walk throughs.

6. Implications

Two questions that any measure of an artifact raises are "what changes to the artifact improve the metric?" and "are such changes reasonable?" Simpler metrics typically make this obvious. Consider *LoC*: if large values for lines of code are correlated with more faults, then a module can be 'too long'. For example, it might surpass the human brain's ability to tract essential detail. In this case refactoring the module to make use of submodules removes the 'too long' module. Thus, it is a reasonable change that improves the measure. A second example considers a class with low coupling caused by two essentially separate functionalities. Again refactoring the two functionalities improves the code. In both of these examples, most engineers will see the benefit that accompanies the refactored code. However, with more complex measures the answer to these two questions is less obvious.

A fixation on metric values, however, can be detrimental. Consider an entry level programmer told to reduce the lines of code in her modules. One potential response is to join all the code onto a single line. This has the effect of improving the metric; however, in answer to the second question, the change is not reasonable.

Consider the two questions with respect to QALP score. An engineer with a fixation on metric values could simply replicate the code in the comments causing a rather high cosine similarity. As with the joining of lines, this approach in not reasonable. However, as anyone who has tried to write a formal description of a function can attest, the process forces one to think about the underlying algorithm and potential issues. Thus, to produce a high quality description of a module (in order to document it appropriately) an engineer must give the module serious consideration. While superficial documentation of code carries little value, careful consideration of the code clearly does. Although not the specific purpose, this activity may also lead to a reduction in the number of faults in the module.

In a similar vain, improving the percent natural language and reducing violations may also reduce defects. Choosing names that use natural language words should make the code easier to comprehend, which in turn will reduce faults caused by comprehension errors. Reducing the number of violations requires one to carefully consider the data stored in an identifier. This process could lead to the detection of defects.

7. Summary and future work

A number of studies have validated the relationship between structural measures and some external attributes associated with problems, such as defectiveness, change-proneness, maintenance difficulty, etc. (Bell et al., 2006; Fenton and Ohlsson, 2000; Gyimóthy et al., 2005; Koru and Tian, 2007; Menzies et al., 2007; Munson and Khoshgoftaar, 1992). One theme noted in many of these studies is the need for more complex, non-structural measures.

This paper presents two case studies of three language-oriented measures, which are non-structural measures derived from the comments and identifiers in source code. One advantage of these language-oriented measures is that they are applicable during both initial development, as well as maintenance and evolution. Natural language measures are significant in all of the statistical models for both case studies, and different measures explain different types of faults as is evident from the low correlations among the language-oriented measures.

In terms of the QALP score, two promising areas of future work include investigating scoring techniques for functions with outward-looking comments and incorporating some measure of 'concept capturing' in the score. In the first area, quality of outward-looking comments might be measured by considering their similarity with the external documentation. The second area is based on an observation of Deißenböck and Pizka that "a reader of a program tries to map the identifiers read to the concepts they may refer to" (Deißenböck and Pizka, 2005). Future improvements to the QALP score will attempt to use ideas from machine learning (Mitchell, 1997) to identify the concepts captured in program identifiers.

More general future work relates to the observation that different kinds of defects are caught by different techniques. Natural language measures being different from past structure measures, may be able to automate the ability to catch of defects currently identified using other techniques. If, for example, natural language measures could catch defects now primarily caught through inspection, then one could automate the identification of a type of defect that requires significant manual effort.

The goal of this work is to demonstrate that natural language based metrics have a place in fault prediction and that they are particularly important because they increase the diversity of the pool of measures available to fault prediction. Given that the models presented in Section 4 demonstrate the usefulness of natural

language measures, the next step is to integrate them with other structural measures to determine the value of adopting more complex models over simple predictors based on one or two variables.

Acknowledgments

This work is supported by National Science Foundation Grant CCR0305330. The authors wish to thank Tibor Gyimóthy's research group for providing the Mozilla fault data and the anonymous referees for providing such extremely helpful comments.

References

Bell, R., Ostrand, T., Weyuker. E., 2006. Looking for bugs in all the right places. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA), Portland, MA, July 2006.

Bentley, Jon, Knuth, Don, 1986. Programming pearls: literate programming. Commun. ACM 29 (5).

Binkley, D., Feild, H., Lawrie, D., Pighin. M., 2007. Software fault prediction using language processing. In: Proceedings of 2nd Testing: Academic and Industrial Conference (TAIC-PART), Windor, UK, September 2007.

Boehm, B., Basili, V., 2001. Software defect reduction top 10 list. IEEE Computer 34 (1).

Chidamber, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6).

Collard, M.L., Kagdi, H.H., Maletic, J.I., 2003. An XML-based lightweight C++ fact extractor. In: 11th IEEE International Workshop on Program Comprehension, 2003, pp. 134–143.

Cordes, David, Brown, Marcus, 1991. The literate-programming paradigm. Computer 24 (6).

Deißenböck, F., Pizka, M., 2005. Concise and consistent naming. In: Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005), St. Louis, MO, USA, May 2005.

Feild, H., Binkley, D., Lawrie. D., 2006. An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proceedings of IASTED International Conference on Software Engineering and Applications (SEA 2006), Dallas, TX, November 2006.

Fenton, N.E., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering 26 (8).

Ferenc, R., Beszédes, A., Tarkiainen, M., Gyimóthy. T., 2002. Columbus – reverse engineering tool and schema for C++. In: IEEE International Conference on Software Maintenance (ICSM 2002), Montreal, Canada, October 2002.

Gyimóthy, T., Ferenc, R., Siket, I., 2005. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software Engineering 31 (10).

Kokol, P., Podgorelec, V., Pighin, M., Zorman, M., Sprogar, M., 2001. An analysis of software correctness prediction methods. In: Proceedings of the Second Asia-Pacific Conference on Quality Software, Hong Kong, December 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

Koru, G., Tian, J., 2007. Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products. IEEE Transactions on Software Engineering 33 (8).

Lavrenko V., Croft. W.B., 2001. Relevance-based language models. In: Croft, W.B., Harper, D.J., Kraft, D.H., Zobel, J. (Eds.), Proceedings on the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.

Lawrie, D., Binkley, D., Feild. H., 2006. Syntactic identifier conciseness and consistency. In: Proceedings of 2006 IEEE Workshop on Source Code Analysis and Manipulation (SCAM'06), Phidelphia, USA, September 2006.

Lawrie, D., Feild, H., Binkley. D., 2006. Leveraged quality assessment using information retrieval techniques. In: 14th International Conference on Program Comprehension, 2006.

Manning, C., Raghaven, P., Schutze, H., 2008. Introduction to Information Retrieval. Cambridge University Press.

Marcus, Andrian, Poshyvanyk, Denys, Ferenc, Rudolf, 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions on Software Engineering 34 (2).

Menzies, T., Greenwald, J., Fransk, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 33 (1).

Mitchell, T., 1997. Machine Learning. WCB McGraw-Hill.

Moonen, L., 2001. Generating robust parsers using island grammars. In: Working Conference on Reverse Engineering.

Morrell, C., Pearson, J., Brant, L., 1997. Linear transformation of linear mixed effects models. The American Statistician 51.

Munson, J.C., Khoshgoftaar, T.M., 1992. The detection of fault-prone programs. IEEE Transactions on Software Engineering 18 (5).

Ott, R.L., Longnecker, M., 2001. An Introduction to Statistical Methods and Data Analysis, fifth ed. Duxbury Press.

Porter, A.A., Selby, R.W., 1990. Empirically guided software development using metric-based classification trees. IEEE Software 7 (2).

Salton, G., McGill, M., 1983. Introduction to Modern Information Retrieval. McGraw-Hill Book Company.

Tian, J., Troster, J., 1998. A comparison of measurement and defect characteristics of new and legacy software systems. Systems and Software 44 (12).

Verbeke, G., Molenberghs, G., 2001. Linear Mixed Models for Longitudinal Data, second ed. Springer-Verlag, New York.

Wheeler, D.A., 2005. SLOC count user's guide. <http://www.dwheeler.com/sloccount/sloccount.html>.

David Binkley is a Professor of Computer Science at Loyola College in Maryland. He received is doctorate from the University of Wisconsin in 1991 at which time he joined the faculty at Loyola. His work has largely focused on dependence analysis. From 1993 to 2000 Professor Binkley also worked as a faculty research at the National Institute of Standards and Technology (NIST), which included work on the Unravel program slicer. In 2000, He worked with Grammatech Inc. on the System Dependence Graph (SDG) based slicer CodeSurfer. Development of the SDG was part of Professor Binkley's PhD work at Wisconsin. Professor Binkley resent NSF funded research focuses on improving semantics-base software engineering tools. This work has recently broadened from considering only programming-language semantics to also include natural-language semantics through the use of Information Retrieval applied to source code and its supporting documents. Current work also includes a seven school collaborative project aimed at increasing the representation of undergraduate women and minorities in computer science.

Henry Feild is an MS-PhD student at the University of Massachusetts Amherst. He graduated with a BS in computer science from Loyola College in 2007. As an undergraduate, his research focused on source code analysis using information retrieval methods. As a graduate student, his research focuses on modeling users of information retrieval systems and interactive search.

Lawrie earned my her PhD from the University of Massachusetts, Amherst. Dr. Lawrie is currently an associate professor of Computer Science at Loyola University Maryland where her research interests include the organization of information and applying information retrieval techniques to source code in order to improve comprehension and quality.

Maurizio Pighin is a Professor in the Department of Mathematics and Computer Science of the University of Udine, and currently teaches advanced courses of Software Engineering and Information Systems. His major research interests are in the area of Software Engineering, ERP and Data Warehouse Systems. He is the author of more than 60 scientific publications in international journals, books and refereed conference proceedings.
He worked at several national and international research and development projects. He is referee of various international journals. He has been involved in the organization of important events in the fields of Software Engineering and Information Systems.