On the Relationship between Comment Update Practices and Software Bugs

Walid M. Ibrahim[a], Nicolas Bettenburg[a], Bram Adams[a,*], Ahmed E. Hassan[a]

[a]*Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada*

Abstract

When changing source code, developers sometimes update the associated comments of the code (a consistent update), while at other times they do not (an inconsistent update). Similarly, developers sometimes only update a comment without its associated code (an inconsistent update). The relationship of such comment update practices and software bugs has never been explored empirically. While some (in)consistent updates might be harmless, software engineering folklore warns of the risks of inconsistent updates between code and comments, because these updates are likely to lead to out-of-date comments, which in turn might mislead developers and cause the introduction of bugs in the future. In this paper, we study comment update practices in three large open-source systems in C (FreeBSD and PostgreSQL) and Java (Eclipse). We find that these practices can better explain and predict future bugs than other indicators like the number of prior bugs or changes. Our findings suggest that inconsistent changes are not necessarily correlated with more bugs. Instead, a change in which a function and its comment are suddenly updated inconsistently, whereas they are usually updated consistently (or vice versa), is risky (high probability of introducing a bug) and should be reviewed carefully by practitioners.

Keywords: code quality, software bugs, software evolution, source code comments, empirical studies

1. Introduction

Source code comments play a central and important role in understanding legacy systems. Comments are often the only form of documentation available for a system, explaining algorithms, informally specifying constraints like pre- and post-conditions, or warning developers of the peculiarities of complex

*Corresponding author

Email addresses: walid@cs.queensu.ca (Walid M. Ibrahim), nicbet@cs.queensu.ca (Nicolas Bettenburg), bram@cs.queensu.ca (Bram Adams), ahmed@cs.queensu.ca (Ahmed E. Hassan)

code [1]. Such documentation is crucial to avoid that developers lose grasp of the system as it ages and evolves [2, 3].

When changing the source code, developers either update the associated comments (a "consistent update") or not (an "inconsistent update"). Updating a comment without the associated code (up to 50% of the comment changes [4]) can also be considered as an inconsistent update. The ill-effects of inconsistent updates between code and comments are often noted as anecdotes by researchers and practitioners. An example of such anecdotes is the change comment attached to change #27068 on October 15, 2007 in the PostgreSQL project (we highlight in bold the most relevant part):

> *"Fix pg_wchar_table[] to match revised ordering of the encoding ID enum. Add some comments so hopefully the next poor sod doesn't fall into the same trap. (**Wrong comments are worse than none at all. . .**)"*

Siy et al. [5] made the same observation during one of their case studies, and noted:

> *"According to most developers we talked to, once they encounter an inconsistent comment, they lose confidence in the reliability of the rest of the comments [. . .] and they ignore the remainder of the comments"*

Indeed, inconsistent updates can be quite critical, as they likely introduce out-of-date comments, which in turn might mislead developers and lead to bugs in the future. For example, a recent manual analysis of the bug reports for the FreeBSD project found sixty bugs that are due to out-of-date comments [6]. However, Fluri et al. [4] found that API (Application Programming Interface) changes, although they impact countless other developers, are typically not commented until later revisions.

While prior research has empirically demonstrated the negative impact of code churn and prior bugs on future code quality, for example in the form of bugs, little is known about the impact of comment update practices. Sundbakken [7] found that the average number of comment lines per C++ class is a good indicator of source code maintainability, yet it is not clear how this relates to comment update practices. While some (in)consistent updates might be harmless (or even expected), others might lead to out-of-date comments, and hence possibly to bugs.

In this paper, we study comment update practices in three large open-source systems in C (FreeBSD and PostgreSQL) and Java (Eclipse) to determine the impact of comment update practices on future bugs. We refine two traditional bug prediction models with information about the comment update practices of the three systems (mined from the source code repositories) to study whether comment update practices are able to explain and predict future bugs. We indeed find a strong relation between comment update practices and future bugs. Closer analysis of our bug models shows that a deviation in the common update practices for a particular function (i.e., the function and its comment are always consistently updated, until suddenly an inconsistent update occurs, or vice versa) is a risk that practitioners must review carefully.

The main contributions of this paper are as follows:

- We empirically study the evolution of comment update practices in three large, long-lived, open-source systems (FreeBSD, PostgreSQL and Eclipse).
- We establish an empirical link between comment update practices and future bugs.

Overview of the Paper: Section 2 provides the necessary background and related work on comments and bug prediction. Section 3 introduces the comment update practices considered in our study, whereas Section 4 explains how we extract the comment update practices from source code repositories. Section 5 studies the distribution over time of comment update practices and future bugs, followed by an exploration of the relation between the comment update practices and bugs in Section 6. Section 7 discusses our findings. Section 8 elaborates on possible threats to the validity of our findings, and Section 9 concludes this work.

2. Background and Related Work

This section discusses typical use case scenarios of source code comments, and presents related work on the evolution of source code comments and on code quality analysis.

2.1. The Use of Comments in Source Code

The most widely-known use of source code comments is to document developer knowledge and assumptions about the source code. A survey of software maintainers done by Souza et al. finds that developers use comments as a key element to understand source code [8]. Similarly, Nurvitadhi et al. report on the significant impact of code comments on improving program understanding among students [9]. Both studies highlight the important and critical role of comments for software development and maintenance.

Recent studies show that comments are also used for other purposes beyond documenting the knowledge of developers. Ying et al. observed that commercial developers use comments to communicate with colleagues through messages such as "TODO" or "FIXME" and to address code-related questions from specific team members [10]. Storey et al. report similar findings through an online survey on a larger, more varied population of developers working on different types of projects [11]. These findings demonstrate the extensive use of source code comments for collaboration and communication throughout the software development process.

A work that is closer to ours is by Tan et al., who use natural language processing techniques to automatically identify and extract locking-related programming rules from comments [6]. Tan et al. then analyze the source code to locate deviations from these extracted rules. This analysis locates *current* bugs in the Linux kernel code based on inconsistencies between source code and comments, whereas we study the impact of inconsistent comment updates on *future* bugs in the whole software system, not just the commented code snippet.

Sundbakken [7] performed an empirical study to identify the major indicators of maintainability of open-source C++ software systems. Although the total number of comment lines is not a good indicator, the average number of comment lines per C++ class is a major indicator of maintainability, because it includes information about the spread of comments across all classes. This work differs from our work in multiple ways. Instead of measuring comments in software *releases* to analyze their role in the ability of developers to *maintain* source code, we consider the *changes* in between releases to identify whether (in)consistent updates to comments are related with *future bugs*.

2.2. Updating Code Comments

The work most closely related to this paper is that on co-evolution of comments and source code by Fluri et al. [4]. The authors perform a fine-grained, AST-level analysis of how comments evolve over time. Based on empirical analysis of eight open-source and commercial systems, the authors find that comments and source code grow similarly (normalized for size) and that 51% to 69% of all comment changes were driven by source code changes, usually in the same commit. Our study builds on the latter finding, yet analyzes comments at the function-level, not statement-level. Instead of focusing on the quality of *comments*, we try to find an empirical link between comment update practices and the quality of a *software system*, in terms of future bugs.

Marin shows that developers are more likely to update the comments of well-commented code. Developers are also more likely to update comments for large and complex changes [12]. Similar results were found by Malik et al., who built a model to predict the likelihood of updating a comment [13]. Their model is influenced in particular by the complexity of the performed change, the time of the change, and developer characteristics. The fact that the complexity of a function is a good indicator for the appearance of bugs [14] suggests that updating a comment (or missing to do so) is likely to lead to future bugs.

Arafat et al. [15] found that commenting source code is an important practice in open-source development. They studied the density of comments in source code changes, i.e., the proportion of new comment lines in a source code change, in more than 5,000 open-source systems. Small changes turn out to have the highest comment density ($> 60\%$ for 1-line changes). Larger changes have ever smaller density, asymptotically approaching 19% (1 comment line per 5 code lines). Comment density is independent of project and team size, but depends on the programming language.

Finally, Siy et al. [5] found that updating comments is often a major priority for companies. More than 28% of all issues identified during code inspection are related to documentation problems. In particular, missing or outdated comments, together with incorrect comment layout and typos in the comments, represent the main source of documentation issues. Furthermore, almost half of the source code lines that are changed or added in response to the inspection results are comments.

```
// Set Timing
char* opt = psql_scan(scan_state,
                      OT_NORMAL,NULL,false);
if (opt)
  pset.timing=ParseBool(opt);
else
  pset.timing = !pset.timing;

if (!pset.quiet) {
  if(pset.timing)
    puts(_("Timing is on."));
  else
    puts(_("Timing is off."));
}
free(opt);
```

(a) Inconsistent Change (IC)

```
// Set Timing with optional on/off
// argument.
char* opt = psql_scan(scan_state,
                      OT_NORMAL,NULL,false);
if (opt)
  pset.timing=ParseBool(opt);
else
  pset.timing = !pset.timing;

if (!pset.quiet) {
  if(pset.timing)
    puts(_("Timing is on."));
  else
    puts(_("Timing is off."));
}
free(opt);
```

(b) Consistent Change (CC)

Figure 1: Example of consistent and inconsistent comment updates (PostgreSQL #28555).

2.3. Bug Prediction

Bug prediction models play an important role in the prioritization of testing and code inspection efforts. For example, if a prediction model determines that a particular component will see a significant increase in error-proneness, managers can react by allocating more testing and inspection efforts, instead of having to wait for bug reports from clients after release [16]. Bug prediction models can also be used to model and explain the reasons for bug occurrences. Projects can control such reasons by improving the software development process. In this paper, we use the models to explain and predict future bugs.

Product-based and process-based models are two common types of bug prediction models. Product-based models use code metrics [17, 18], such as the cyclomatic complexity, lines of code (LOC), and the number of nested statements. Process-based models [16, 19, 20, 21] use historical information about the development process to predict the number of future bugs. Examples of such historical information are the number of prior code changes (we name such models "CHANGES models"), and the number of prior bugs [22] (BUGS models). Moser et al. and Graves et al. show that process metrics (especially BUGS models) perform better than code metrics to predict future bugs [19, 23, 24, 25].

To show an empirical link between comment update practices and future bugs, this paper explores whether the ability of comment update practices to explain and predict future bugs is at least as good as that of the established CHANGES and BUGS models.

3. Comment Update Practices

This paper considers the comments of source code functions, which comprise both comments inside the function body (*internal comments*) and outside (*external comments*). In a *consistent change*, a developer would update (add, remove, or modify) a piece of code inside a function and its associated comment

(at least one internal or external comment of the function). If the developer updates a piece of code but does not update its associated comment (or vice versa), then we consider this change as an *inconsistent change*.

Figure 1 illustrates the concepts of consistent and inconsistent changes on change #28555 for the PostgreSQL system. In Figure 1(a), the source code change that introduces an extra command line argument was not accompanied by an update to the code snippet's internal comment (inconsistent change). Ideally, the developer should have noted her change inside the internal comment (or an external one of the function) to make the new behaviour of the method more explicit for her colleagues, as shown in Figure 1(b) (consistent change).

Our definitions of (in)consistent change implicitly assume that each change to a function ideally should be accompanied by a comment change. On the one hand, this assumption is in line with established programming guidelines and common sense. We already discussed in Section 2.2 how open-source projects [15] and companies [5] value comment updates in source code changes. In addition, to avoid outdated, unreliable comments ("comment rot" [26]), well-known software development books recommend that "when you fix, add, or modify any code, [then] fix, add, or modify any comments around it" [27]. Furthermore, comments are meant to explain the "why" of a piece of source code instead of the "how" [28]. When a bug is fixed, it is important to indicate to other developers the tricky piece of code responsible for the bug. When a performance optimization is introduced, it is crucial to indicate the idea behind the optimization. This philosophy has led to a popular practice called "Pseudocode Programming Process" [28], or, more colloquially, "comment-driven development" [29], which complements test-driven development and design-by-contract. Instead of directly programming an algorithm or class, developers describe the design of the functionality in terms of high-level pseudo-code [30, 31]. This pseudo-code is iteratively refined into actual code with the original pseudo-code left as comments. Every time the design changes, the pseudo-code, and hence the comments, should be updated before the actual code can be changed.

On the other hand, the assumption that all source code changes should be accompanied by a comment change is a simplification, because code changes that fix small typos or refactor the code probably do not require updating the comments. Although Arafat et al. [15] found that small changes had comment densities of up to 60%, many small changes might be falsely interpreted as inconsistent. Unfortunately, it is hard to determine which changes require comments and which ones do not, not only for the actual developers, but also for other practitioners or researchers. Hence, based on our industrial experience and the various supporting literature, this paper assumes that each source code change requires a comment change, yet we need to take this simplification into account when discussing the results of our study.

To study the impact of comment update practices on future bugs, a variety of strategies can be used. We choose to compare the ability of the practices to explain and predict future bugs relative to the established BUGS and CHANGES bug prediction models [19, 23, 24, 25]. Our goal is not to present a new prediction model, but to show that comment update practices are somehow related

to the occurrence of bugs, because they are able to explain and predict future bugs with similar performance to real bug prediction models.

Such a comparison approach to established models has been used before by Graves et al. [32, 19] and requires to refine the established models with information about the new concept one wishes to evaluate. In our case, we refine the concept of "change" in the CHANGES model into:

- **Consistent Change (CC)**: represents any change to the source code, either to add a new feature or to perform a maintenance task like bug fixing, that is also reflected in the corresponding source code comment(s).
- **Inconsistent Change (IC)**: represents any change to the source code that is not reflected in the corresponding source code comment(s), or vice versa.

Similarly, we refine the concept of "bug fix" in the BUGS model into:

- **Consistent Bug Fix (CB)**: restricts consistent changes to bug fix changes with the associated comment(s) updated.
- **Inconsistent Bug Fix (IB)**: restricts inconsistent changes to bug fix changes with no update to the associated comment(s), or vice versa.

The remainder of this paper focuses on the comparison of the explanatory and predictive power of the refined CHANGES (in terms of CC/IC) and BUGS (in terms of CB/IB) models to the original CHANGES (in terms of changes) and BUGS (in terms of bugs) models.

4. Data Collection

We now present the systems studied in this paper and we discuss our technique to automatically extract the comment update practices from the development repository of these systems.

4.1. Studied Systems

We use three open-source software systems with a combined development history of about 30 years. These systems come from different domains, to address possible bias in our results towards a particular problem domain or development process. Similar to other work [33], we define a subsystem as any logical or physical collection of source code files. For the C systems, we use file system directories, for the Java systems, packages.

We considered the following subject systems:

- **FreeBSD** [34] is a UNIX-like operating system written in C. The studied system contains 3 MLOC spread across 152 subsystems [35]. We studied the development period between June 1993 and August 2005.
- **PostgreSQL** [36] is an open-source database management system written in C. PostgreSQL contains 0.5 MLOC spread across 280 subsystems [35]. We studied the development period between July 1996 and February 2008.
- **Eclipse** [37] is an open-source integrated development environment (IDE) written in Java. Eclipse contains 1.2 MLOC over 661 subsystems [38]. We studied the development period between November 2001 and December 2004, as provided by Zimmermann et al.'s PROMISE data set [39].

4.2. Recovery of Comment Update Practices

For our analysis, we must recover the number of consistent and inconsistent changes and bug fixes. However, such data is not directly available, as source control systems like CVS and Subversion do not provide fine-grained source code and comment change-tracking. These source control systems only keep track of changed lines of code, regardless of whether the line belongs to a function, a data definition or a comment.

To overcome these limitations of source control systems, we use an evolutionary code extractor (C-REX [40, 32] for FreeBSD and PostgreSQL, and J-REX [41] for Eclipse). These evolutionary code extractors use robust parsing techniques, such as island grammars [42], to map historical code changes to the corresponding code entities and to link internal and external comments to their corresponding entities. External comments followed by a function or data structure definition are linked to that definition, whereas other external comments are linked to the enclosing file. To determine whether or not a (bug fix) change to a function is (in)consistent, we check whether the change modifies any external or internal comment of the function. For example, if a parameter is changed without changing any external or internal comment associated to the function, this change is considered an IC.

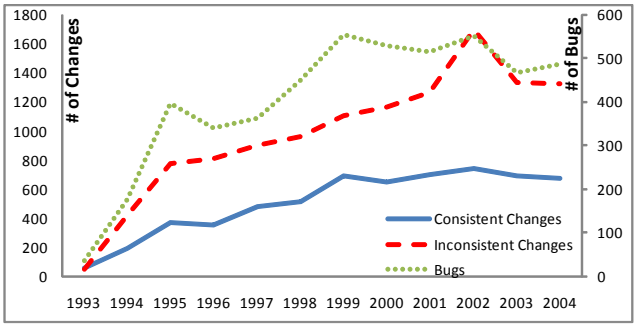
As is common practice in the community (e.g., [24, 32, 43]), we approximate the number of bugs in each subsystem by the number of bug fix changes in the subsystem. This approximation is necessary, because (1) only fixed bugs can be mapped to specific pieces of code, (2) not all reported bugs are real bugs, (3) not all bugs have been reported, and (4) there are many duplicate bug reports.

However, the mapping between changes in source control repositories and bugs in bug tracking systems is rarely explicit. To determine which changes are bug fix changes, we used two techniques. For FreeBSD and PostgreSQL, we use an automatic lexical classification technique [44, 45] that analyzes the change messages attached to each change for keywords indicative of a bug fix, followed by a number. Common keywords include "fix", "bug", or "PR", followed by a unique bug identifier that denotes the bug that is being fixed. We performed a manual verification of a representative sample of the extracted bugs to make sure that they are actual bugs [45]. The average precision for extracted bugs is 91%, while the average recall is 60%. For Eclipse, we use the bug data extracted by Zimmermann et al. [39], which has become a de facto benchmark for bug prediction studies.

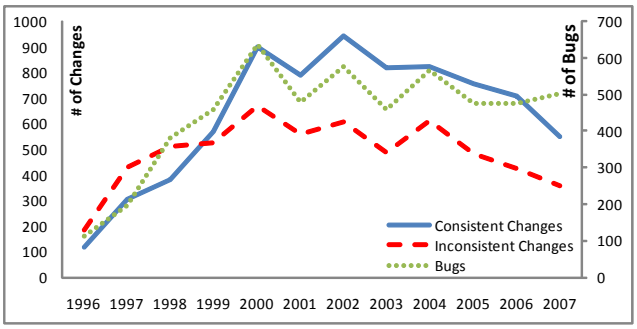
Using the change and bug data extracted for the three systems, we can count the number of consistent and inconsistent changes and bug fixes in FreeBSD, PostgreSQL and Eclipse.

5. Studying Update Practices

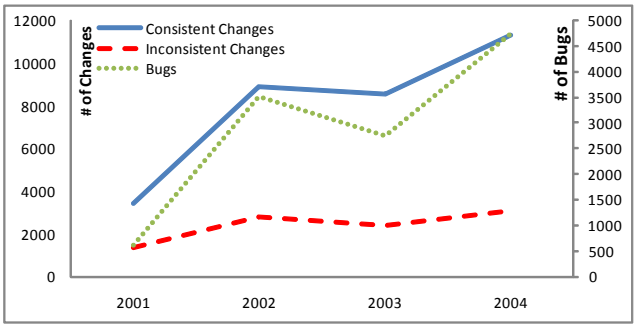
This section analyzes the distributions of (in)consistent changes (CC/IC) and (in)consistent bug fixes (CB/IB) in the three subject systems to determine the appropriate prediction model to build in the next section. Because this



(a) FreeBSD



(b) PostgreSQL

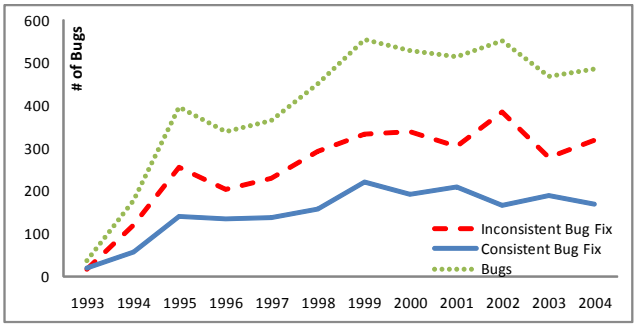


(c) Eclipse

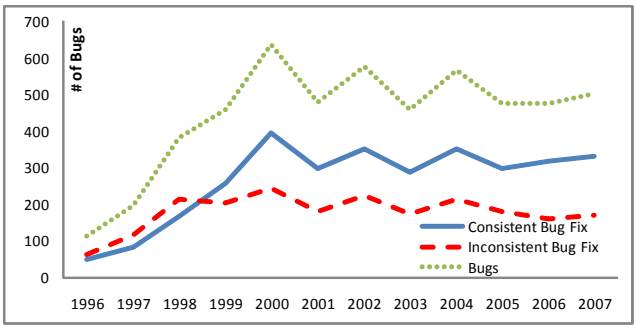
Figure 2: Comment update practices for code changes.

study focuses on future bugs caused by out-of-date comments, we especially look at the relation between the number of CCs and ICs for a given year with the number of bugs in the following year [19, 24].

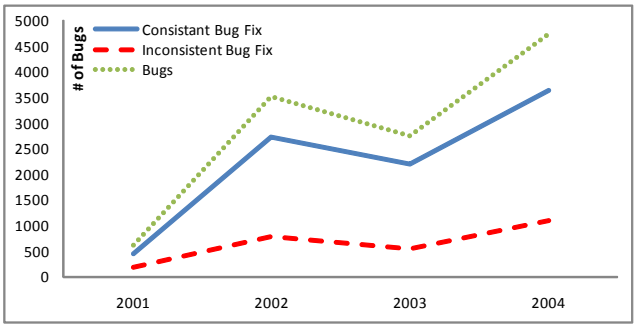
- **Approach.** We perform the following analyses:
- we study the evolution of the total (across all subsystems) number of CCs/ICs/CBs/IBs



(a) FreeBSD



(b) PostgreSQL



(c) Eclipse

Figure 3: Comment update practices for bug fixes.

and the number of extracted bugs over time via plots;

- we graphically examine possible statistical interactions between the number of CCs/ICs (or CBs/IBs) per subsystem in a given year and the number of bugs per subsystem in the following year.
  The absence of statistical interaction would mean that, as the number of

(for example) CCs increases, the number of bugs increases at the same rate, independent of the IC value. However, if there is an interaction, then, as the number of (for example) CCs increases, the number of bugs increases at a different rate depending on the IC value. The presence of an interaction would require us to build a more complex model in the next section.

We use a common statistical interaction analysis [46, 47] that has been used before in empirical software engineering [48]. We classify the subsystems in each system into three groups, based on the number of ICs of each subsystem: low IC (the 33% of the subsystems with the lowest number of ICs), high IC (the 33% of the subsystems with the highest number of ICs), and medium IC (all other subsystems). For each group, we then make a scatterplot that shows, for each subsystem in the IC group, its total number of CCs in one year versus its number of bugs in the following year. The best fitting curves for the scatter plots reveal whether or not there is an interaction between ICs and CCs. If there is no interaction, we expect to see three near-parallel lines — one for each of the low, medium, and high groups. The larger the difference between the slopes, the larger the interaction effect. We use the same approach for CBs and IBs. We use the log of CCs, ICs, CBs, and IBs to handle the high skew in the data.

Comment update practices are system-dependent.

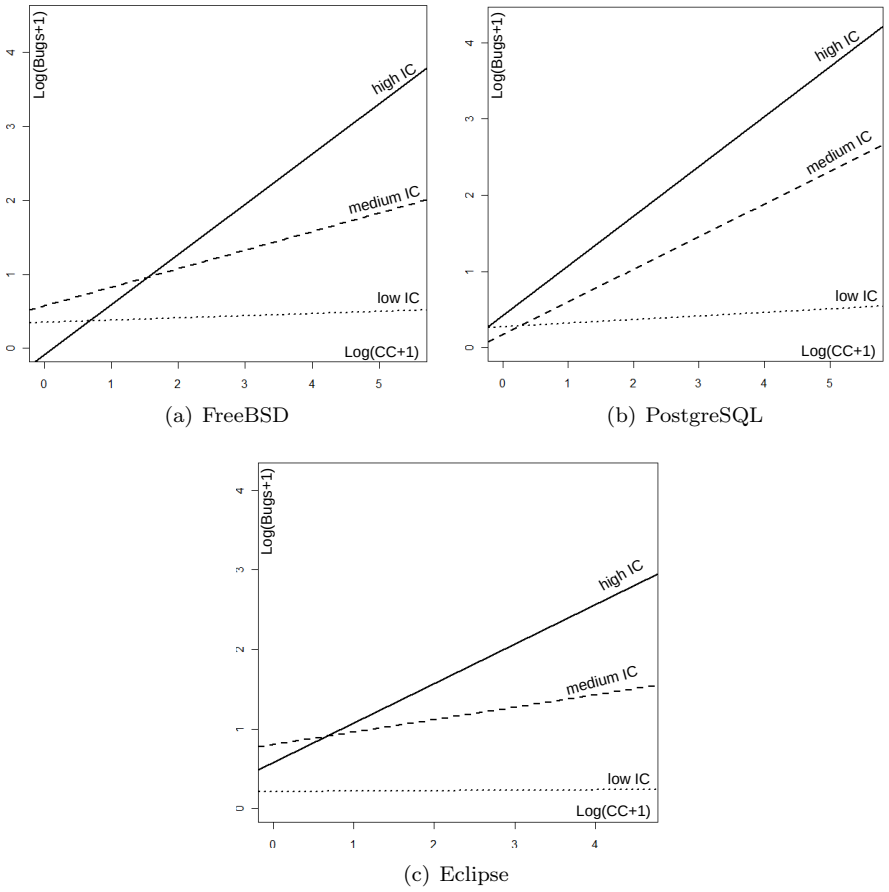
Figure 2 plots the evolution of the total number of CCs and ICs in FreeBSD, PostgreSQL, and Eclipse, together with the number of bugs:

1. ***In FreeBSD:*** There are about twice as many inconsistent changes as consistent changes. The number of ICs is always higher than the number of CCs.
2. ***In PostgreSQL:*** The early years of development show slightly more ICs than CCs, but, from 1999 on, the number of CCs significantly increases to about one and a half times the number of ICs.
3. ***In Eclipse:*** There are about three times as many consistent changes as inconsistent changes. The number of CCs is always higher than the number of ICs.

The differences between FreeBSD, PostgreSQL, and Eclipse indicate that these systems are different, not only in domain, but also in their developers' comment update practices. Despite the system-dependent nature of their comment update practices, all systems show a similar trend in the number of bugs: a fast growth in the first two years, followed by a slowdown. The plots for the number of CBs and IBs (Figure 3) follow a similar trend as the plots for CCs and ICs.

Statistical interaction between comment update practices and future bugs.

Figure 4 shows the interaction curves of the CCs and ICs for the three systems in 2002 (other years are similar). The figure shows a clear interaction effect between CC and IC in all three systems, as each IC group has a different slope. For low IC, the number of bugs remains almost the same (near zero) as the number of CCs increases, while for high IC the number of bugs increases



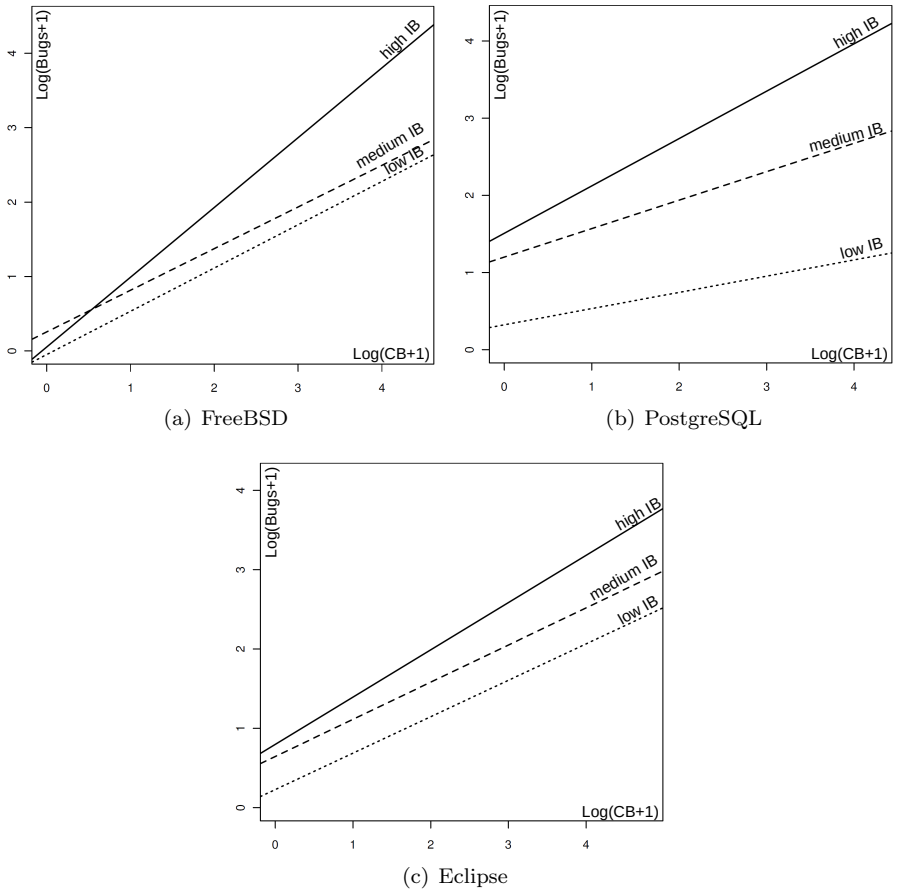
(a) FreeBSD (b) PostgreSQL

(c) Eclipse

Figure 4: The interaction effects of consistent and inconsistent changes in FreeBSD, PostgreSQL and Eclipse.

more rapidly as the number of CCs increases. Therefore, the impact of CC on the number of bugs depends on the value of IC. A similar interaction effect was observed when grouping subsystems based on CC values. The interaction curves for the CBs and IBs (Figure 5) also indicate the presence of (slightly weaker) interaction for FreeBSD and PostgreSQL, whereas Eclipse does not exhibit any interaction at all.

In summary, for most cases, there exists an interaction between comment update practices and future bugs. Hence, the next step is to account for this observed interaction in our models to explain and predict future bugs.



(a) FreeBSD (b) PostgreSQL

(c) Eclipse

Figure 5: The interaction effects of consistent and inconsistent bug fixes in FreeBSD, PostgreSQL and Eclipse.

6. Impact on Future Bugs

To study the impact of comment update practices on future bugs, we compare the ability of update practices to explain and predict future bugs to the ability of two well-known and frequently used measures, i.e., the number of prior changes (CHANGES model) and the number of prior bugs (BUGS model) [19].

For each studied system, we build six different linear regression models to model the number of future bugs in a subsystem. Four of them (Model_C, Model_B, Model_{IC+CC}, and Model_{IB+CB}) are additive models of the form $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$, while the other two models (Model_{IC*CC} and Model_{IB*CB}) are multiplicative models of the form $\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1 \cdot x_2)$. In these formulas, $\hat{y}$ is the predicted number of bugs in a subsystem. The independent variables x_i are summarized in Table 1. They represent the number

Table 1: The independent variables of our six regression models.

Model	Value of x_i
$\boldsymbol{Model_C}$	x_1 = # changes in a subsystem and x_2 =0.
$\boldsymbol{Model_B}$	x_1 = # bug fixes in a subsystem and x_2 =0.
$\boldsymbol{Model_{CC+IC}}$	x_1 = # CC and x_2 = # IC in a subsystem.
$\boldsymbol{Model_{CB+IB}}$	x_1 = #CB and x_2 = # IB in a subsystem.
$\boldsymbol{Model_{CC*IC}}$	x_1 = # CC, x_2 = # IC, and interaction in a subsystem.
$\boldsymbol{Model_{CB*IB}}$	x_1 = # CB, x_2 = # IB, and interaction in a subsystem.

of bugs, changes, or comment update practices in a subsystem, depending on the particular model. To stabilize the variance of the error in the built models, which is one of the requirements for linear models, we use the natural log of $x_i + 1$ instead of x_i [49].

The multiplicative models capture the interaction $\beta_3(x_1 \cdot x_2)$ between the comment update practices discovered in the previous section. To make sure that this interaction term does not lead to over-fitting, we always compare the multiplicative models to the simple additive models, Model_{CC+IC} and Model_{CB+IB}.

The construction of the six models requires a training and prediction phase:

Training phase: For FreeBSD and PostgreSQL, we train the linear regression models that predict the number of future bugs in a subsystem (dependent variable) for a year F_3 based on the independent variables (Table 1) in the two previous years (F_1 and F_2). We perform year-based prediction, because out-of-date comments have a higher probability of causing bugs as time goes by, with developers forgetting the context of source code changes. Similar to previous work [43, 39], we perform release-based analysis for Eclipse instead of year-based analysis. We train the linear regression models that predict the number of post-release bugs (dependent variable) based on the independent variables (Table 1) collected one year prior to a release.

Testing phase: For FreeBSD and PostgreSQL, we use the trained model for year F_3 to predict the number of bugs in the fourth year (F_4) using input from years F_2 and F_3. We repeat this setup six times, because we have 12 years worth of historical data for FreeBSD and PostgreSQL. We do not use the data of the first two years of each system because of transient effects in those early years, as can be seen in Figure 2. For Eclipse, we use the trained model to predict the number of post-release bugs of the following release. We test our models using data from releases 2.1 and 3.0.

6.1. Measuring the Explanatory Power

Approach. We calculate the proportion of variance (R^2) of our six regression models to assess how well the regression models fit (i.e., explain) the *training* data. The higher the R^2, the better the fit. We compare the change-based models to each other and the bug-based models to each other (i.e., we compare R^2_C with R^2_{CC+IC} and R^2_{CC*IC}, and R^2_B with R^2_{CB+IB} and R^2_{CB*IB}). We also perform an ANOVA analysis ($\alpha = 0.05$) to determine if the interaction

Table 2: The R^2 statistic for the regression models. The greyed cells contain R^2 statistics that are statistically significant compared to $Model_C$ and $Model_B$. The * indicates that the interaction term is statistically significant according to the ANOVA test ($\alpha = 0.05$).

(a) FreeBSD

Model built for year	R^2_C	R^2_{CC+IC}	R^2_{CC*IC}	R^2_B	R^2_{CB+IB}	R^2_{CB*IB}
1999	0.42	0.43	0.44 *	0.38	0.39	0.41 *
2000	0.51	0.53	0.57 *	0.47	0.50	0.51
2001	0.61	0.65	0.70 *	0.62	0.66	0.66
2002	0.56	0.57	0.61 *	0.60	0.61	0.63 *
2003	0.50	0.53	0.58 *	0.53	0.54	0.55
2004	0.60	0.63	0.70 *	0.61	0.63	0.66 *

(b) PostgreSQL

Model built for year	R^2_C	R^2_{CC+IC}	R^2_{CC*IC}	R^2_B	R^2_{CB+IB}	R^2_{CB*IB}
2002	0.80	0.81	0.83 *	0.76	0.76	0.81 *
2003	0.82	0.84	0.85 *	0.86	0.88	0.88
2004	0.77	0.78	0.78	0.77	0.78	0.79 *
2005	0.76	0.78	0.78	0.72	0.73	0.81 *
2006	0.76	0.77	0.82 *	0.80	0.81	0.82
2007	0.73	0.75	0.81 *	0.75	0.76	0.78 *

(c) Eclipse

Model built for release	R^2_C	R^2_{CC+IC}	R^2_{CC*IC}	R^2_B	R^2_{CB+IB}	R^2_{CB*IB}
2.0	0.45	0.49	0.50 *	0.56	0.57	0.57
2.1	0.59	0.64	0.66 *	0.78	0.80	0.80
3.0	0.61	0.66	0.69 *	0.78	0.80	0.80

term of the multiplicative model is needed in our models and to determine the statistical significance of the observed improvement in explanatory power (R^2).

Findings. Table 2 shows the R^2 values for the trained models of the three systems. A * next to a value indicates that the interaction term is statistically significant according to the ANOVA with a 5% level of significance (i.e., $\alpha = 0.05$). We find that:

- For all years and most systems, the additive models (i.e., $Model_{CC+IC}$ and $Model_{CB+IB}$) show an R^2 value that is at least as good as the value for basic models (i.e., $Model_C$ and $Model_B$).
- For all years and systems, the multiplicative models (i.e., $Model_{CC*IC}$ and $Model_{CB*IB}$) show an R^2 value that is at least as good as the value for

the additive models (i.e., $Model_{CC+IC}$ and $Model_{CB+IB}$). For many of the years, the difference between the multiplicative and additive models caused by the interaction term is statistically significant based on the ANOVA tests.
- As expected from the absence of interaction between CB and IB for Eclipse, the corresponding bug-based multiplicative model $Model_{CB*IB}$ performs identical to the additive model $Model_{CB+IB}$.

The CHANGES and BUGS models enhanced with comment update practices show statistically significant improvements in explanatory power over the basic, non-enhanced CHANGES and BUGS models.

6.2. Measuring the Predictive Power

Approach. We compare the prediction performance of the four regression models that use knowledge about comment update practices to the performance of the CHANGES model (Model_C) and the BUGS model (Model_B). To evaluate the overall improvement in performance of each model for a particular system in a particular year, we measure the total prediction error E_{Model} of a model across all n subsystems of the system during that year as: $E_{\text{Model}} = \sqrt{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$, where y_i is the actual number of bug fixes in subsystem i during the prediction year (see Section 4.2), while $\hat{y}_i$ is the predicted number of bugs for subsystem i. In Tables 3 and 4 and Figure 6 , we transform the predicted number of bugs from $\log(\hat{y}+1)$ back to $\hat{y}$ to make our explanation easier to follow.

A model might outperform another model due to chance. Thus, we perform a statistical evaluation of the observed difference in performance. For this purpose, we use statistical hypothesis tests to assess the significance of the difference in prediction errors between each model. Our statistical analysis assumes a 5% level of significance (i.e., $\alpha = 0.05$), when testing the following hypotheses: $H_0 : \mu(e_{A,i} - e_{B,i}) = 0, \quad H_A : \mu(e_{A,i} - e_{B,i}) \neq 0$, where $\mu(e_{A,i} - e_{B,i})$ denotes the population mean of the difference between the total prediction error of the two models A and B for each subsystem. If the p-value is lower than $\alpha = 0.05$, we can reject H_0 with high probability, which means that the difference in performance is statistically significant, i.e., not due to chance.

To test these hypotheses, we conduct a parametric (t-test) and a non-parametric (Wilcoxon signed rank test) paired statistical test. We perform a Wilcoxon test to ensure that non-significant results are not simply due to the departure of the data from the t-test assumptions. For the results presented, both tests are consistent, hence we only report the values of the t-test. We also adjust the level of significance of the t-test using a Bonferroni correction based on the number of tests that are performed [50]. For example, if we would perform 20 hypothesis tests, then 1 out of the 20 tests would show a significance at a confidence level of $\alpha = 0.05$, even if the differences are not statistically significant. Therefore, we must adjust our α and make it stricter. For example,

Table 3: Total prediction error improvement for bug prediction models based on the CHANGES model. A positive value means that $Model_{CC*IC}$ performs better than $Model_C$ (smaller error). The value in parentheses is the percentage of improvement relative to E_C. The greyed error improvements are statistically significant.

(a) FreeBSD

Pred. Year	E_C	E_{CC+IC}	E_{CC*IC}	E_C - E_{CC*IC}(%)
1999	85.8	83.0	76.8	9.0 (10.5%)
2000	76.3	69.7	54.9	21.4 (28.0%)
2001	82.7	73.3	65.7	17.0 (20.6%)
2002	95.3	90.4	48.5	46.8 (49.1%)
2003	115.6	110.1	63.0	52.6 (45.5%)
2004	122.3	116.7	53.5	68.8 (56.3%)

(b) PostgreSQL

Pred. Year	E_C	E_{CC+IC}	E_{CC*IC}	E_C - E_{CC*IC}(%)
2002	57.2	55.4	49.7	7.5 (13.1%)
2003	43.4	43.8	43.2	0.2 (0.5%)
2004	61.2	58.1	49.9	11.3 (18.5%)
2005	56.1	53.6	39.6	16.5 (29.4%)
2006	57.1	54.4	55.6	1.5 (2.6%)
2007	77.9	74.5	55.7	22.2 (28.5%)

(c) Eclipse

Pred. Release	E_C	E_{CC+IC}	E_{CC*IC}	E_C - E_{CC*IC}(%)
2.1	239.4	230.4	207.9	31.5 (13.2%)
3.0	345.9	345.4	313.4	32.5 (9.3%)

in Table 3(a) we perform 6 t-tests for FreeBSD. Therefore, we use a corrected Bonferroni confidence level of $\alpha = 1-(1-0.05)^{1/6} = 0.0085$ instead of $\alpha = 0.05$.

Findings. Tables 3 and 4 present the total prediction error for the CHANGES and BUGS models. The last column shows the difference between the total prediction error of the basic model and that of the multiplicative model, together with the percentage of improvement of the total prediction error compared to the error of the basic model. Grey cells in the tables indicate differences that are statistically significant. We observe that:

- For most of the years, the additive models (i.e., $Model_{CC+IC}$ and $Model_{CB+IB}$) show better total prediction error values than the basic models (i.e., $Model_C$ and $Model_B$). Yet, none of the improvements are statistically significant.
- For most of the years, the multiplicative models (i.e., $Model_{CC*IC}$ and $Model_{CB*IB}$) show better total prediction error values than the additive (i.e., $Model_{CC+IC}$ and $Model_{CB+IB}$) and basic models (i.e., $Model_C$ and $Model_B$). Most of the recorded reductions of error are statistically significant compared to the basic models (greyed cells). For the change-based multiplicative models, the statistically significant reduction of error for $Model_{CC*IC}$ over $Model_C$ is between 9.3% and 56.3% with an average of 26.8%. For the bug-based multiplicative models, the statistically significant reduction of error for $Model_{CB*IB}$ over $Model_B$ is between -11.5% and 65.0% with an average of 26.0%. $Model_{CB*IB}$ for 2003 is an outlier, since it performs significantly worse than $Model_B$.
- Table 4(c) shows that the prediction error of $Model_{CB*IB}$ for Eclipse is

Table 4: Total prediction error improvement for bug prediction models based on the BUGS model. A positive value means that $Model_{CB*IB}$ performs better than $Model_B$ (smaller error). The value in parentheses is the percentage of improvement relative to E_B. The greyed error improvements are statistically significant.

(a) FreeBSD

Pred. Year	E_B	E_{CB+IB}	E_{CB*IB}	E_B - E_{CB*IB}(%)
1999	83.9	79.6	75.7	8.2 (9.8%)
2000	74.9	61.8	48.1	26.8 (35.8%)
2001	65.8	46.2	50.2	15.6 (23.7%)
2002	76.9	64.0	46.4	30.5 (65.0%)
2003	99.0	87.8	56.4	42.6 (43.0%)
2004	103.5	88.2	61.5	42.0 (40.6%)

(b) PostgreSQL

Pred. Year	E_B	E_{CB+IB}	E_{CB*IB}	E_B - E_{CB*IB}(%)
2002	50.9	49.0	50.1	0.8 (1.6%)
2003	43.5	51.5	48.5	-0.5 (-11.5%)
2004	57.5	53.6	53.2	4.3 (7.5%)
2005	45.1	40.5	34.5	10.6 (23.5%)
2006	37.6	34.9	35.1	2.5 (6.6%)
2007	58.3	59.6	46.7	11.6 (19.9%)

(c) Eclipse

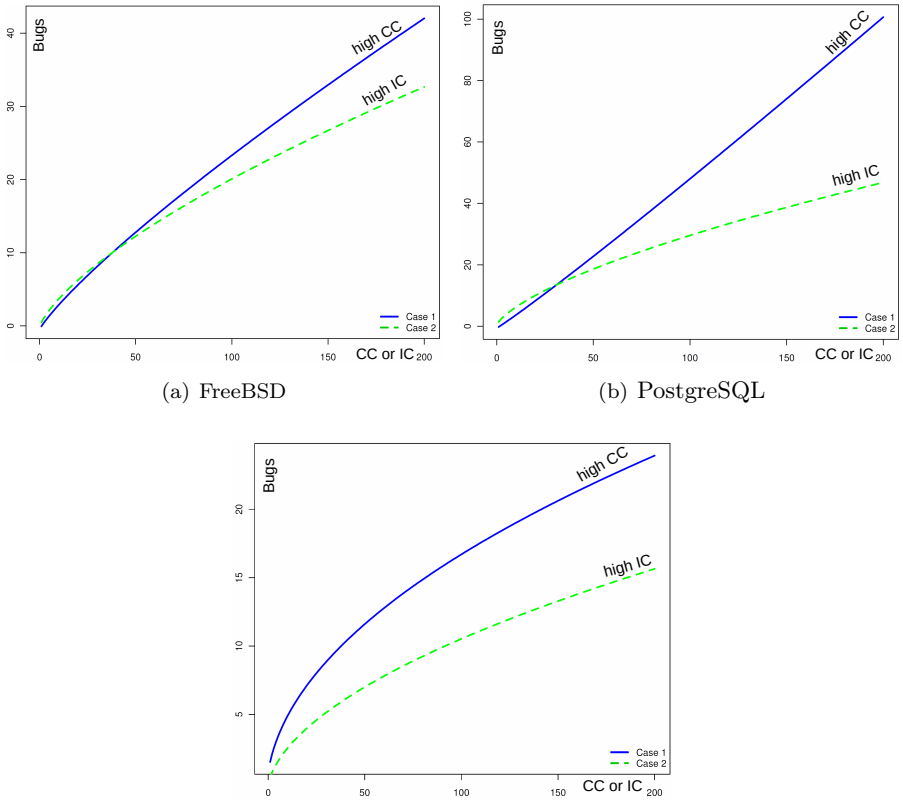
Pred. Release	E_B	E_{CB+IB}	E_{CB*IB}	E_B - E_{CB*IB}(%)
2.1	169.6	140.9	152.1	17.5 (10.3%)
3.0	238.3	209.9	211.2	27.1 (11.4%)

higher than the prediction error for $Model_{CB+IB}$ and that the performance of $Model_{CB*IB}$ is not statistically significant. This result was expected, as the R^2 for $Model_{CB+IB}$ and $Model_{CB*IB}$ was identical in Table 2(c) and our earlier interaction analysis indicated that no interaction existed.

The CHANGES and BUGS models enhanced with comment update practices show statistically significant improvements in predictive power over the basic CHANGES and BUGS models.

7. Discussion

According to our findings, models taking into account comment update practices and the interaction between them show a statistically significant improvement over the explanatory and prediction power of basic bug models. According to the research approach that we have used [32, 19], this statistically significant improvement means that we have demonstrated an empirical link between comment update practices and future software bugs. As a side-effect, the convincing improvement of explanatory and predictive power using comment update patterns suggests that researchers should consider integrating these comment update patterns in future bug models. In particular, our Eclipse comment update models are showing an improvement (although not statistically significant) over the BUGS model, which is the top performing prediction model for the Eclipse data set [23].



(a) FreeBSD (b) PostgreSQL

(c) Eclipse

Figure 6: The plots of case 1 vs. case 2 explaining the interaction term for $Model_{CC*IC}$. The plots are not in log scale.

To fully understand how specific comment update practices influence future bugs, we should perform a more fine-grained study, including a manual analysis of source code changes and bug repositories. We consider this to be future work. However, our regression models allow us to get an initial, high-level idea of the effect of comment update practices and their interaction on future bugs. To make our discussion more concrete, we use the following two representative multiplicative models from the FreeBSD system (built to predict bugs in 2002):

$$\hat{y}_{FreeBSD} = 0.06 - 0.04CC + 0.16IC + 0.14CC * IC \quad (1)$$

$$\hat{y}_{FreeBSD} = 0.22 + 0.24CB + 0.16IB + 0.17CB * IB \quad (2)$$

The negative coefficient of CC in Equation 1 suggests that consistent changes (CC) improve code quality, i.e., reduce the probability of future bugs. Inconsis-

tent changes and bug fixes all increase the number of bugs (positive coefficients). Surprisingly, consistent bug fixes (CB) negatively impact quality (positive coefficient in Equation 2). To understand this observation, we carefully study the interaction terms in Equations 1 and 2 using the interaction analysis techniques of Section 5. We consider the four extreme cases of comment update practices and interpret Equations 1 and 2 for these cases:

1. Changes often update comments (High CC)
2. Changes rarely update comments (High IC)
3. Bug fixes often update comments (High CB)
4. Bug fixes rarely update comments (High IB)

Figure 6 shows the data of FreeBSD, PostgreSQL, and Eclipse for cases 1 and 2, i.e., the impact on the number of bugs for functions whose comments are often updated (High CC, for different numbers of ICs) versus functions whose comments are rarely updated (High IC, for different numbers of CCs). The High IC and High CC interaction lines of a particular system both contain data for the same number of subsystems (i.e., one third of all subsystems, see Section 5). We can observe that:

- The High CC curve is always higher than the High IC curve, which means that failing to update a piece of code that is usually updated (High CC) has a higher probability of introducing bugs than updating a comment that is rarely updated (High IC).
- The High IC curve (the dashed line) shows that the number of bugs increases in a logarithmic fashion. This logarithmic increase indicates that for the average High IC function in a subsystem, the relative impact of updating a comment is higher when the number of comment updates (CCs) is small.
- The logarithmic pattern is not as pronounced for the High CC curves of FreeBSD and PostgreSQL, which indicates that in those systems the relative impact of missing an update is more or less consistent across different numbers of ICs.

In summary, it appears that (1) not updating comments when one usually updates comments and (2) updating comments when one usually does not, are indications of code changes that have a higher probability of introducing a bug than changes that always (not) update comments.

We performed the same analysis for cases 3 and 4, but only for FreeBSD and PostgreSQL, because Eclipse does not experience interaction effects. The findings are identical to those for cases 1 and 2, except that the High IB curve is higher than the High CB curve instead of the other way around, i.e., the impact of inconsistency on code quality is much higher for bug fixes than for regular changes. These results support earlier work [12, 13] that highlights the critical need for tools that can recommend which comments to update. More detailed analysis is needed to understand if these initial observations hold for all regression models and if they generalize to other systems as well.

8. Threats to Validity

External Threats to Validity. We studied the comment update practices of three large, open-source systems in the domains of operating systems, databases, and IDEs. These domains are commonly used in bug prediction studies and cover a wide range of systems, development styles, and system sizes. However, our results might not generalize to systems of different domains, programming paradigms, or documentation paradigms (e.g., business applications). open-source systems have special characteristics that may not hold for commercial software systems. With developers of open-source projects commonly distributed across the globe, they might rely more heavily on source code comments for project documentation and collaboration.

Our findings may not generalize to environments that put emphasis on external documentation (like API documents) as the main source of documentation instead of on code comments. This seems unlikely, given the role and importance of source code comments (see Sections 2 and 3). In addition, even in these environments, the basic problem of keeping code changes consistent with the external documentation remains, as Arisholm et al. demonstrate in a study on the impact of UML documentation on software maintenance [51].

Internal Threats to Validity. We used two specialized extractors to detect changes to comments and changes to code entities (e.g., functions). The extractor used a robust parsing technique called island grammars to parse uncompilable code. There is a chance that this technique might fail due to complex macro definitions or that we might map comment changes to incorrect code entities. However, past research has shown that the extractor that we used has a very high accuracy and reliability, even on large software systems like the Linux kernel [52]. The extractor has also been used independently by other groups on commercial systems [32].

Our approach used heuristics to classify whether a given change is consistent or inconsistent. However, as we did not consider the semantics of the performed changes, we might perform wrong classifications. For example, a developer might change both the source code and the associated comment, but render the comment out-of-date by adding incomplete or wrong information to it. A possible solution is to use natural language processing techniques to automatically identify incomplete or incorrect comment updates.

Similarly, we assumed that all code changes require corresponding comment changes and that otherwise such changes are considered to be inconsistent. Sections 2 and 3 argued that such an assumption makes sense, based on common development guidelines [29, 27, 28] and earlier research [15, 5]. Still, more work is needed to validate this assumption and to explore less strict definitions of consistency.

Our regression models are based solely on comment update practices and as such might ignore additional variables that could increase the model accuracy. Nevertheless, our ANOVA tests show that all of the variables that we used in the model are necessary for prediction and cannot be left out. In addition, the BUGS and CHANGES models are currently the best performing bug prediction

models [23].

Our prediction models cannot show a causal effect of inconsistent changes on future bugs. However, we demonstrate that there is an improvement in explanatory and predictive power when using comment update practices to predict future bugs.

As explained in Section 6, we predict the number of bugs for a given year (FreeBSD and PostgreSQL) or for a given release (Eclipse). Hassan [24] and Graves [19] built their prediction model based on years, while Zimmermann [39] and Nagappan [43] built their prediction models based on releases. We use both approaches to predict future bugs to verify the generality of our findings for cross/intra-release quality management.

9. Conclusion

Comments are essential for software developers and maintainers, yet out-of-date comments can put developers on the wrong track and lead to software bugs, often a long time after the code changes that caused the comments to be out-of-date. In this study, we established an empirical link between comment update practices and future bugs, by showing that established bug prediction models refined by comment update practices statistically significantly improve on the explanatory and predictive power of the original bug prediction models.

However, the relation between (in)consistent code changes and future bugs is not straightforward, because there is a statistical interaction between the number of consistent and inconsistent changes. In other words, inconsistent changes do not necessarily increase the number of future bugs. More work is needed on the interpretation of this finding and its implications on practitioners. Preliminary analysis suggests that the probability of future bugs increases for changes that (1) miss to update a comment in a subsystem whose comments are usually always updated or (2) update a comment in a subsystem whose bug fixes usually do not update its comments. Careful review of such changes and bug fixes is advised.

More detailed, fine-grained analysis is needed to derive more concrete comment updating guidelines and to drive the development of methodologies and tools to prevent out-of-date comments.

Acknowledgments. The authors want to thank Yasutaka Kamei and reviewers of earlier revisions for their insightful suggestions and comments.

References

[1] S. N. Woodfield, H. E. Dunsmore, V. Y. Shen, The effect of modularization and comments on program comprehension, in: Proc. of the 28th Intl. Conf. on Software Engineering (ICSE), 1981, pp. 215–223.

[2] F. P. Brooks, The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2nd Edition, Addison-Wesley Professional, 1995.

[3] D. L. Parnas, Software aging, in: Proc. of the 16th Intl. Conf. on Software Eng. (ICSE), 1994, pp. 279–287.

[4] B. Fluri, M. Wursch, H. C. Gall, Do code and comments co-evolve? on the relation between source code and comment changes, in: Proc. of the 14th Working Conf. on Reverse Engineering (WCRE), 2007, pp. 70–79.

[5] H. Siy, L. Votta, Does the modern code inspection have value?, in: Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM), 2001, pp. 281–290.

[6] L. Tan, D. Yuan, G. Krishna, Y. Zhou, /*icomment: bugs or bad comments?*/, in: Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP), 2007, pp. 145–158.

[7] M. Sundbakken, Assessing the maintainability of c++ source code, Master's thesis, Washington State University, Pullman, Washington (December 2001).

[8] S. C. B. de Souza, N. Anquetil, K. M. de Oliveira, A study of the documentation essential to software maintenance, in: Proc. of the 23rd annual Intl. Conf. on Design of Comm. (SIGDOC), 2005, pp. 68–75.

[9] E. Nurvitadhi, W. W. Leung, C. Cook, Do class comments aid java program understanding?, 33rd Annual Frontiers in Education (FIE) 1 (2003) T3C–13–T3C–17.

[10] A. T. T. Ying, J. L. Wright, S. Abrams, Source code that talks: An exploration of eclipse task comments and their implication to repository mining, in: Proc. of the 2005 Intl. wrksh. on Mining Software Repositories (MSR), 2005, pp. 1–5.

[11] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, J. Singer, Todo or to bug: exploring how task annotations play a role in the work practices of software developers, in: Proc. of the 30th Intl. Conf. on Software Engineering (ICSE), 2008, pp. 251–260.

[12] D. P. Marin, What motivates programmers to comment?, Master's thesis, EECS Department, University of California, Berkeley (Nov 2005).

[13] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, A. E. Hassan, Understanding the rationale for updating a function's comment, in: Proc. of the 24th IEEE Intl. Conf. on Software Maintenance (ICSM), 2008, pp. 167–176.

[14] I. Herraiz, J. M. Gonzalez-Barahona, G. Robles, Towards a theoretical model for software growth, in: Proc. of the 4th Intl. Wrksh. on Mining Software Repositories (MSR), 2007, pp. 21–28.

[15] O. Arafat, D. Riehle, The commenting practice of open source, in: Proc. of the 24th ACM SIGPLAN conf. companion on Object Oriented Programming Systems Languages and Applications (OOPSLA), 2009, pp. 857–864.

[16] E. Arisholm, L. C. Briand, Predicting fault-prone components in a Java legacy system, in: Proc. of the 2006 ACM/IEEE Intl. Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, 2006, pp. 8–17.

[17] V. R. Basili, B. T. Perricone, Software errors and complexity: An empirical investigation, Commun. ACM 27 (1) (1984) 42–52.

[18] L. Hatton, Reexamining the fault density-component size connection, IEEE Software 14 (2) (1997) 89–97.

[19] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Trans. Softw. Eng. 26 (7) (2000) 653–661.

[20] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, J. P. Hudepohl, Data mining for predictors of software quality, International Journal of Software Engineering and Knowledge Engineering 9 (5) (1999) 547–564.

[21] M. Leszak, D. E. Perry, D. Stoll, Classification and evaluation of defects in a project retrospective, J. Syst. Softw. 61 (3) (2002) 173–187.

[22] T.-J. Yu, V. Y. Shen, H. E. Dunsmore, An analysis of several software defect models, IEEE Trans. Softw. Eng. 14 (9) (1988) 1261–1270.

[23] R. Moser, W. Pedrycz, G. Succi, A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction, in: Proc. of the 30th Intl. Conf. on Software Engineering (ICSE), 2008, pp. 181–190.

[24] A. E. Hassan, Predicting faults using the complexity of code changes, in: Proc. of the 31st Intl. Conf. on Software Engineering (ICSE), 2009, pp. 78–88.

[25] N. Nagappan, T. Ball, Use of relative code churn measures to predict system defect density, in: Proc. of the 27th Intl. Conf. on Software Engineering (ICSE), 2005, pp. 284–292.

[26] Aargh, Part Five: Comment Rot, http://blogs.msdn.com/b/ericlippert/archive/2004/05/04/125893.aspx, last accessed, April 2011.

[27] P. Goodliffe, Code Craft: The Practice of Writing Excellent Code, No Starch Press, San Francisco, CA, USA, 2006.

[28] S. McConnell, Code Complete, Second Edition, Microsoft Press, Redmond, WA, USA, 2004.

[29] Comment-Driven Development, http://blogs.sitepoint.com/comment-driven-development/, last accessed, April 2011.

[30] S. H. Caine, E. K. Gordon, Pdl: a tool for software design, in: Proc. of the National Computer Conference and Exposition (AFIPS), 1975, pp. 271–276.

[31] H. R. Ramsey, M. E. Atwood, J. R. Van Doren, Flowcharts versus program design languages: an experimental comparison, Commun. ACM 26 (1983) 445–449.

[32] M. Cataldo, A. Mockus, J. A. Roberts, J. D. Herbsleb, Software dependencies, work dependencies, and their impact on failures, IEEE Trans. Software Eng. 35 (6) (2009) 864–878.

[33] M. W. Godfrey, Q. Tu, Evolution in open source software: A case study, in: Proc. of the Intl. Conf. on Software Maintenance (ICSM), 2000, pp. 131–142.

[34] FreeBSD, http://www.freebsd.org/, last accessed, February 2010.

[35] Z. Li, S. Lu, S. Myagmar, Y. Zhou, Cp-miner: A tool for finding copy-paste and related bugs in operating system code, in: OSDI, 2004, pp. 289–302.

[36] PostgreSQL, http://www.postgresql.org/, last accessed, February 2010.

[37] Eclipse, http://www.eclipse.org/, last accessed, February 2010.

[38] T. Mens, J. Fernandez-Ramil, S. Degrandsart, The evolution of eclipse, in: Proc. of the 24th IEEE Intl. Conf. on Software Maintenance (ICSM), 2008, pp. 386 –395.

[39] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Proc. of the 3rd International Wrksh. on Predictor Models in Software Engineering (PROMISE), 2007, pp. 9–15.

[40] A. E. Hassan, R. C. Holt, C-Rex: An Evolutionary Code Extractor for C, presented at Consortium for Software Engineering Meeting (CSER) (2004).

[41] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, Mapreduce as a general framework to support research in mining software repositories, in: Proc. of the 6th Intl. Working Conf. on Mining Software Repositories (MSR), 2009, pp. 21–30.

[42] L. Moonen, Generating robust parsers using island grammars, in: Proc. of the 8th Working Conf. on Reverse Engineering (WCRE), 2001, pp. 13–22.

[43] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proc. of the 28th Intl. Conf. on Software Engineering (ICSE), 2006, pp. 452–461.

[44] A. Mockus, L. G. Votta, Identifying reasons for software change using historic databases, in: Proc. of the 16th Intl. Conf. on Software Maintenance (ICSM), 2000, pp. 120–130.

[45] A. E. Hassan, Automated classification of change messages in open source projects, in: Proc. of the 2008 ACM Symp. on Applied Computing (SAC), 2008, pp. 837–841.

[46] J. Jaccard, R. Turrisi, Interaction effects in multiple regression, 2nd Edition, SAGE, 2003.

[47] L. S. Aiken, S. G. West, R. R. Reno, Multiple regression: testing and interpreting interactions, 1st Edition, SAGE, 1991.

[48] L. C. Briand, Y. Labiche, M. D. Penta, H. D. Yan-Bondoc, An experimental investigation of formality in UML-based development, IEEE Transactions on Software Engineering 31 (2005) 833–849.

[49] S. Weisberg, Applied Linear Regression, John Wiley and Sons, 1980.

[50] H. Abdi, Bonferroni and Sidak corrections for multiple comparisons, in: Encyclopedia of Measurement and Statistic, Sage, 2007, pp. 103–107.

[51] E. Arisholm, L. C. Briand, S. E. Hove, Y. Labiche, The impact of UML documentation on software maintenance: An experimental evaluation, IEEE Transactions on Software Engineering 32 (6) (2006) 365–381.

[52] A. E. Hassan, Z. M. Jiang, R. C. Holt, Source versus object code extraction for recovering software architecture, in: Proc. of 12th Working Conf. on Reverse Engineering (WCRE), 2005, pp. 67–76.