A Large-Scale Empirical Study of Just-In-Time Quality Assurance

Yasutaka Kamei, *Member, IEEE,* Emad Shihab, Bram Adams, *Member, IEEE,*
Ahmed E. Hassan, *Member, IEEE,* Audris Mockus, *Member, IEEE,*
Anand Sinha, and Naoyasu Ubayashi *Member, IEEE*

Abstract—Defect prediction models are a well-known technique for identifying defect-prone files or packages such that practitioners can allocate their quality assurance efforts (e.g., testing and code reviews). However, once the critical files or packages have been identified, developers still need to spend considerable time drilling down to the functions or even code snippets that should be reviewed or tested. This makes the approach too time-consuming and impractical for large software systems. Instead, we consider defect prediction models that focus on identifying defect-prone ("risky") software *changes* instead of files or packages. We refer to this type of quality assurance activity as "Just-In-Time Quality Assurance", because developers can review and test these risky changes while they are still fresh in their mind (i.e., at check-in time). To build a change risk model, we use a wide range of factors based on the characteristics of a software change, such as the number of added lines, and developer experience. A large-scale study of 6 open source and 5 commercial projects from multiple domains shows that our models can predict whether or not a change will lead to a defect with an average accuracy of 68% and an average recall of 64%. Furthermore, when considering the effort needed to review changes, we find that using only 20% of the effort it would take to inspect all changes, we can identify 35% of all defect-inducing changes. Our findings indicate that "Just-In-Time Quality Assurance" may provide an effort-reducing way to focus on the most risky changes and, thus, reduce the costs of developing high-quality software.

Index Terms—Maintenance, Software metrics, Mining software repositories, Defect prediction, Just-In-Time prediction

1 Introduction

Software quality assurance activities (e.g., source code inspection and unit testing) play an important role in producing high quality software. Software defects[1] in released products have expensive consequences for a company and can affect its reputation. At the same time, companies need to make profit, so they should minimize the cost of maintenance activities. To address this challenge, a plethora of software engineering research focuses on prioritizing software quality assurance activities [4], [6], [49].

The majority of quality assurance research focused on defect prediction models that identify defect-prone modules (i.e., files or packages) [17], [19], [30], [45]. Although such models can be useful in some contexts, they also have their drawbacks. We can summarize the drawbacks of such approaches as follows:

1) **Prediction units are coarse-grained:** Predictions at the package or file granularity leave it to developers to locate the risky code snippets in these files.
2) **Relvant experts are not identified:** Once a file is identified as risky, someone needs to find the expert for inspection/testing and other quality improvement activities. This may be a nontrivial task: for example, some files in the Mozilla project are touched by up to 155 developers.
3) **Predictions are made too late in the development cycle:** Practitioners want to know about quality issues as soon as possible while the change details are still fresh in their minds.

Therefore, researchers have proposed to perform predictions at the change-level, focusing on predicting defect-inducing changes [26], [41]. We illustrate the advantage of change-level prediction with the following example. Alice is a software development manager for a large legacy system that is subject to ongoing maintenance. Bob and Chris are two developers of the system, and Bob has modified a file A. A year later, in the course of estimating the expected quality of the upcoming release, a package/file-level prediction model used by Alice indicates that file A is fault-prone. Given the length of time that passed since the change, Alice does not know who's change

- *Y. Kamei and N. Ubayashi are with Graduate School and Faculty of Information Science and Electrical Engineering, Kyushu University, Japan. E-mail: kamei@ait.kyushu-u.ac.jp*
- *E. Shihab is with the Department of Software Engineering, Rochester Institute of Technology, Rochester, NY, USA.*
- *A. E. Hassan is with the School of Computing, Queen's Univ., Kingston, Canada.*
- *B. Adams is with the Département du Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, Montréal, Canada.*
- *A. Mockus is with Avaya Labs Research.*
- *A. Sinha is with Mabel's Labels.*

1. The term "defect" is used to mean a static fault in source code [18].

introduced the defect. Even if Alice would assign the work to Bob, Bob would now need to remember the rationale and all the design decisions of the change to be able to evaluate if the change introduced a defect. Using change-level prediction, Alice (or, preferably Bob) would be notified immediately (upon check-in) that there is a high probability that a defect was introduced to file A. Bob can quickly revisit the rationale and design and, if necessary, fix the defect, because he still has the context fresh in his mind.

Mockus and Weiss [41] assess the risk of Initial Modification Requests (IMR) in 5ESS network switch project, i.e., the probability that such requests are defect-inducing. IMRs may consist of multiple Modification Requests (MR), which, in their turn, may contain multiple changes. Kim *et al.* [26] used the identifiers in added and deleted source code and the words in change logs to classify changes as being defect-prone or clean.

We refer to this type of quality assurance activity as "Just-In-Time Quality Assurance". In contrast to current approaches (i.e., package/file-level prediction), Just-In-Time Quality Assurance aims to be an earlier step of continuous quality control because it can be invoked as soon as a developer commits code to their private or to the team's workspace. The main advantages of performing change-level predictions are:

1) **Predictions are made at a fine granularity:** Predictions identify defect-inducing changes, which are mapped to small areas of the code and provide large effort savings over coarser-grained predictions [23].
2) **Predictions can be expressed as concrete work assignments for a developer to fix a defect due to a change:** Changes can be easily mapped to the person who committed the change. This can save time in finding the developer who introduced the defect (if a defect is induced), because the person who made the change is most familiar with it.
3) **Predictions are made early on:** Only change properties are used for predictions. Therefore, the prediction can be performed at the time when the change is submitted for unit test (private view) or integration. Such immediate feedback ensures that the design decisions are still fresh in the minds of developers.

Prior work has considered change-level predictions in a limited context. First, Mockus and Weiss [41] analyzed only a single large telecommunications system, whereas Kim *et al.* [26] considered exclusively open source systems. Second, prior change-level prediction studies did not consider the effort required for quality assurance (e.g., testing and code reviews), a crucial aspect for any practical application as shown in our analysis. Third, earlier studies consider only metrics specific to their particular context.

We, therefore, perform an empirical study of change-level predictions on a variety of open source and commercial projects from multiple domains. Our multi-domain, multi-company data set is composed of six open source projects and five commercial projects, containing more than 250,000 changes and covering two of the most popular programming languages (C/C++ and Java). We use a broad array of features to assess what type of factors provide good predictive power across this wide sample of projects. To accomplish that, we describe operationalizations of these features for all of these projects, that is, how we map these features to metrics, collect the metrics and pre-process the metrics (e.g., collinearity, skew and imbalance).

We formulate our study in the form of three research questions:

RQ1 *How well can we predict defect-inducing changes?*

Mockus and Weiss [41] evaluated the prediction performance using one commercial project. To better understand the generalizability of the results, we use 6 open source projects and 5 commercial projects developed by two companies. To identify defect-inducing changes, we build a change-level prediction model based on a mixture of established and new metrics (e.g., the distribution of modified code across each file [19]). We are able to predict defect-inducing changes with 68% accuracy and 64% recall.

RQ2 *Does prioritizing changes based on predicted risk reduce review effort?*

More recently, research on file-level defect prediction started to evaluate prediction performance in a more practical setting by taking into consideration review effort [23], [35]. For example, while a traditional prediction model could identify a large file to have five defects, the effort needed to inspect the whole file to find those defects could be much larger than the effort needed to inspect five smaller files, each having one defect. Hence, a model recommending the latter five files would be more advantageous in practice. Note that we assume that every defect has the same weight, i.e., we do not distinguish defects with different severity. We mention the implications of this assumption in Section 7.

Inspired by this work, we apply effort-aware evaluation to JIT Quality Assurance. To our knowledge, such an evaluation has not been reported before at change-level. Our findings indicate that spending only 20% of the total available required effort (measured as

the percentage of modified code) suffices to identify up to 35% of all defect-inducing changes.

RQ3 *What are the major characteristics of defect-inducing changes?*
We study which factors have the largest impact on our predictions, i.e., we identify the major characteristics of risky changes. Our findings for the open source projects show that the number of files and whether or not the change fixes a defect are risk-increasing factors, whereas the average time interval since the previous change is a risk-decreasing factor in RQ1 and RQ2. For the commercial projects, we also find that the diffusion factors are consistently important in RQ1 and RQ2. However, in RQ1 they are risk-increasing, whereas in RQ2 they are risk-decreasing.

The rest of the paper is organized as follows. Section 2 introduces background and related work. Section 3 explains the change measures, which are metrics to build a prediction model at change-level. Section 4 provides the design of our experiment and Section 5 presents the results. Section 6 discusses our findings. Section 7 reports the threats to validity and Section 8 presents the conclusions of the paper.

2 Background and Related Work

In this section, we review the related work. The majority of this work is about long-term and short-term quality assurance.

2.1 Long-term quality assurance

Used metrics. A plethora of studies use various metrics to predict defects. Prior work used product metrics such as McCabe's cyclomatic complexity metric [33] and the Chidamber and Kemerer (CK) metrics suite [9] and code size (measured in lines of code) [1], [12], [15], [29], [21]. Other work uses process metrics to predict defect-prone locations [15], [19], [43], [44], [46]. Graves *et al.* [15] use process metrics based on the change history (e.g., number of past defects and number of developers) to build defect prediction models. They show that process metrics are better defect features than product metrics like McCabe's cyclomatic complexity. Nagappan and Ball [46] use relative code churn metrics, which measure the amount of code change, to predict file-level defect density. Jiang *et al.* [22] compare the use of design and code metrics in predicting fault-prone modules and find that code-based models outperform design-level models. Moser *et al.* [43] show that process metrics could perform at least as well as code metrics in predicting defect-prone files in the Eclipse project. Hassan [19] shows that scattered changes are good indicators of defect-prone files. We leverage the knowledge from these prior studies and use 14 different factors extracted from code changes to predict whether or not a change will induce a defect.

Generalizability of findings. There are many empirical studies on defect prediction using open source projects and commercial projects. Briand *et al.* [7] analyze the relationship between design and software quality in a commercial object-oriented system. The result of their case study showed that the frequency of method invocations appears to be the main driving factor of defect-proneness. In order to draw more general conclusions, Gyimothy *et al.* [17] also analyze the relationship of Chidamber and Kemerer's metrics with software quality using data collected from the open source Mozilla project across several releases (version 1.0-1.6). They reported that CBO (Coupling Between Object classes) seems to be the best measure for predicting defect-prone classes. Zimmermann *et al.* [62] focus on defect prediction from one project to another using seven commercial projects and four open source projects. They show that there is no single factor that leads to accurate predictions. We contribute to the body of work in defect prediction by performing our case study on 11 different projects, i.e., six open source and five commercial projects. This increases the external validity of our findings.

Effort-aware models. Recent work by Arisholm *et al.* [2], Menzies *et al.* [37], Mende and Koschke [35], and Kamei *et al.* [23] examine the performance of effort-aware defect prediction models. Mende and Koschke [35] use the number of lines of code as a measure of reviewing effort, similar to Arisholm *et al.* [2]. Experimental results using publicly available data sets show that the prediction performance of effort-aware models improved from a cost-effectiveness point, compared to traditional, non effort-aware models. Kamei *et al.* [23] revisit some of the major findings in the traditional defect prediction literature by taking into account the effort needed for software quality assurance. Their findings indicate that process metrics outperform product metrics by a factor of 2.6 when considering effort. In this paper, we also consider effort-aware predictions and elaborate on the impact of effort on predicting defect-inducing changes.

The previous work, mentioned thus far, focuses on long-term quality assurance, that is it predicts the probability of defects or the number of defects for a particular software location (i.e., file or package). In contrast to previous studies, we focus on predicting the probability of a software *change* inducing a defect at check-in time.

2.2 Short-term quality assurance

Some previous studies focus on the prediction of risky software changes. Mockus and Weiss [41] predict the risk of a software change in an industrial project. They use change measures, such as the number of

subsystems touched, the number of files modified, the number of lines of added code and the number of modification requests. Motivated by their previous work, we study the risk of software change on a set of six open source and five commercial projects and also evaluate our findings when considering the effort required to review the changes. The predictions of Mockus and Weiss were done at a coarse-grained granularity, i.e., Initial Modification Requests (IMR) that consist of multiple modification requests, which are made up of multiple changes. On the other hand, our predictions provide results with a fine-grained granularity (i.e., at the individual change-level).

Sliwerski *et al.* [57] study defect-introducing changes in the Mozilla and Eclipse open source projects. The authors find that defect-introducing changes are generally a part of large transactions and that defect-fixing changes and changes done on Fridays have a higher chance of introducing defects. More recently, Eyolfson *et al.* [13] study the correlation between a change's bugginess and the time of the day the change was committed and the experience of the developer making the change. They perform their study on the Linux kernel and PostgreSQL and find that changes performed between midnight and 4AM are more buggy than changes committed between 7AM and noon and that developers who commit regularly produce less buggy changes. Yin *et al.* [60] perform a study that characterizes incorrect defect-fixes in Linux, OpenSolaris, FreeBSD and a commercial operating system. They find that 14.8 - 24.2% of fixes are incorrect and affect end users, that concurrency defects are the most difficult to correctly fix and that developers responsible for incorrect fixes usually do not have enough knowledge about the code being changed.

Aversano *et al.* [3] and Kim *et al.* [26] use source code change logs to predict the risk of a software change. For example, Kim *et al.* sampled a balanced subset of changes in several open source projects and obtained 78% accuracy and a 60% recall. Our results on a full set of changes in open source and commercial projects show that we are able to predict defect-inducing changes with 68% accuracy and 64% recall despite a much lower fraction bug inducing changes[2]. The combination of change logs and our change measures contributes to the improvement of prediction performance. In addition to conventional criteria (e.g., accuracy and recall), our empirical evaluation at the change-level also takes effort into account.

3 Change Measures

To predict whether or not a change introduces a future defect, we consider 14 factors grouped into five dimensions derived from the source control repository data of a project. As shown in Table 1, we use these factors since they perform well in traditional defect prediction research and we want to explore their performance in the context of defect-inducing changes. We now describe each dimension and its factors in more detail.

Diffusion dimension: Diffusion of a change is one of the most important factors in predicting the probability of a defect [41]. A highly distributed change is more complex to understand and generally requires keeping track of all locations that must be changed. For example, Mockus and Weiss [41] show that the number of subsystems touched is related to the probability of a defect. Hassan [19] shows that scattered changes are good indicators of defects.

We expect that the diffusion dimension can be leveraged to determine the likelihood of a defect-inducing change. A total of four different factors make up the diffusion dimension, as listed in Table 1.

We use the root directory name as the subsystem name (i.e., to measure NS), the directory name to identify directories (i.e., ND) and the file name to identify files (i.e., NF). To illustrate, if a change modifies a file with the path, "org.eclipse.jdt.core/jdom/org/eclipse/jdt/core/dom/Node.java", then the subsystem is org.eclipse.jdt.core, the directory is org.eclipse.jdt.core/jdom/.../dom and the file name is Node.java.

To measure entropy, we use measures similar to Hassan [19]. Entropy is defined as: $H(P) = -\sum_{k=1}^{n}(p_k * \log_2 p_k)$, where probabilities $p_k \geq 0, \forall k \in 1, 2, ..., n$, n is the number of files in the change, P is a set of p_k, where p_k is the proportion that $file_k$ is modified in a change and $(\sum_{k=1}^{n} p_k) = 1$. Entropy aims to measure the distribution of the change across the different files. If, for example, a change modifies 3 different files, A, B and C and the number of modified lines in file A, B and C is 30, 20 and 10 lines, respectively, then the Entropy is measured as 1.46(=$-\frac{30}{60}log_2\frac{30}{60} - \frac{20}{60}log_2\frac{20}{60} - \frac{10}{60}log_2\frac{10}{60}$). The formula for Entropy above has been normalized by the maximum Entropy $log_2 n$ to account for differences in the number of files across changes, similar to Hassan [19]. The higher the normalized Entropy, the larger the spread of a change.

Size dimension: A large change has a higher chance of introducing a defect, since more code has to be changed or implemented. For example, Nagappan and Ball [46] and Moser *et al.* [43] show that the size of a change (e.g., the number of lines of code added in a revision) is a good feature of defect-prone modules.

We conjecture that larger changes are more likely to introduce defects. The size dimension consists of three different factors (LA, LD and LT), as shown in Table 1. These factors can be measured directly from the source control repository.

Purpose dimension: A change that fixes a defect is more likely to introduce a new defect [16][52]. The intuitive reasoning behind this is that files that had

2. As noted later, the accuracy and precision drop dramatically with the fraction of defect-inducing changes.

TABLE 1
Summary of change measures

Dim.	Name	Definition	Rationale	Related Work
Diffusion	NS	Number of modified subsystems	Changes modifying many subsystems are more likely to be defect-prone.	The defect probability of a change increases with the number of modified subsystems [41].
	ND	Number of modified directories	Changes that modify many directories are more likely to be defect-prone.	The higher the number of modified directories, the higher the chance that a change will induce a defect [41].
	NF	Number of modified files	Changes touching many files are more likely to be defect-prone.	The number of classes in a module is a good feature of post-release defects of a module [47]
	Entropy	Distribution of modified code across each file	Changes with high entropy are more likely to be defect-prone, because a developer will have to recall and track large numbers of scattered changes across each file.	Scattered changes are more likely to introduce defects [10], [19].
Size	LA	Lines of code added	The more lines of code added, the more likely a defect is introduced.	Relative code churn measures are good indicators of defect modules [43], [46].
	LD	Lines of code deleted	The more lines of code deleted, the higher the chance of a defect.	
	LT	Lines of code in a file before the change	The larger a file, the more likely a change might introduce a defect.	Larger modules contribute more defects [27].
Purpose	FIX	Whether or not the change is a defect fix	Fixing a defect means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.	Changes that fix defects are more likely to introduce defects than changes that implement new functionality [16][52].
History	NDEV	The number of developers that changed the modified files	The larger the NDEV, the more likely a defect is introduced, because files revised by many developers often contain different design thoughts and coding styles.	Files previously touched by more developers contain more defects [32].
	AGE	The average time interval between the last and the current change	The **lower** the AGE (i.e., the more recent the last change), the more likely a defect will be introduced.	More recent changes contribute more defects than older changes [15].
	NUC	The number of unique changes to the modified files	The larger the NUC, the more likely a defect is introduced, because a developer will have to recall and track many previous changes.	The larger the spread of modified files, the higher the complexity [10], [19].
Experience	EXP	Developer experience	More experienced developers are less likely to introduce a defect.	Programmer experience significantly decreases the defect probability [41].
	REXP	Recent developer experience	A developer that has often modified the files in recent months is less likely to introduce a defect, because she will be more familiar with the recent developments in the system.	
	SEXP	Developer experience on a subsystem	Developers that are familiar with the subsystems modified by a change are less likely to introduce a defect.	

defects in them previously tend to have more defects in the future [15].

To determine whether or not a change fixes a defect, we search the change logs for keywords like "bug", "fix", "defect" or "patch", and for defect identification numbers. A similar approach to determine defect fixing changes was used in other work [26].

History dimension: The history of a change contains useful information that can help us determine whether or not the change will be defect-inducing in the future. For example, previous studies showed that the number of previous changes and defect fixes to a file are a good indicator of the file's bugginess [15]. Matsumoto *et al.* [32] show that the files previously touched by many developers contain more defects.

Three factors fall under the history dimension, as shown in Table 1. We use an example to illustrate how we measure these factors. NDEV is the number of developers that previously changed the touched files. For example, if a change has files A, B and C, file A thus far has been modified by developer x, and files B and C have been modified by developers x and y, then NDEV would be 2 (x and y). AGE is the average time interval between the current and the last time these files were modified. For example, if file A was last modified 3 days ago, file B was modified 5 days ago and file C was modified 4 days ago, then AGE is calculated as 4 (i.e., $\frac{(3+5+4)}{3}$). NUC is the number of unique last changes of the modified files. For example, if file A was previously modified in change α and files B and C were modified in change β, then NUC is 2 (i.e., α and β).

Experience dimension: Although some reports [51] suggest that using information about developers does not improve the prediction of defects, the personnel involved with a change actually is responsible for in-

ducing most of the defects. For example, Mockus and Weiss [41] show that higher programmer experience significantly decreases a change's defect probability. Matsumoto *et al.* [32] show that defect injection rates vary among different developers.

The experience dimension is composed of three factors, as shown in Table 1. Similar to prior work [32][41], developer experience (EXP) is measured as the number of changes made by the developer before the current change. Recent experience (REXP) is measured as the total experience of the developer in terms of changes, weighted by their age. It gives a higher weight to changes that are more recent. Subsystem experience (SEXP) measures the number of changes the developer made in the past to the subsystems that are modified by the current change.

We use the following weighting scheme to measure REXP: $\frac{1}{(n+1)}$ where n is measured in years. For example, if a developer of a change made 3 changes in the current year, 4 changes 1 year ago and 3 changes 2 years ago, then REXP is 6 (i.e., $= \frac{3}{1} + \frac{4}{2} + \frac{3}{3}$).

4 Study Setup

In this case study, we aim to answer the three research questions posed earlier. Here, we detail the systems used in our case study and the data extraction and processing steps used.

4.1 Studied systems

Previous studies on change risk examined the risk of changes in open source projects only [26] or commercial projects only [41]. To improve the generalizability of our results, and produce more concrete findings, we use 11 different projects. Six are large, well-known open source projects (i.e., Bugzilla, Columba, Mozilla, Eclipse JDT, Eclipse Platform and PostgreSQL) and five are large, commercial projects (which we refer to as C-1, C-2, C-3, C-4 and C-5). The projects used are written in C/C++ and/or Java.

To conduct our case study, we extracted information from the CVS repositories of the projects and combined it with bug reports. We used the data provided by the MSR 2007 Mining Challenge[3] to gather the data for the Bugzilla and Mozilla projects. The data for the Eclipse JDT and Platform projects was gathered from the MSR 2008 Mining Challenge[4]. For Columba[5] and PostgreSQL[6], we mirrored the official CVS repository. The commercial projects C-1, C-2, C-3 and C-4 are written in Java, and contain hundreds of thousands of lines of code. Each project is developed by hundreds developers. Project C-5 is a high availability telephone switching system that contains tens of millions of lines of code, mostly in C and C++. The project is developed by thousands practitioners. We do prediction on a small subset of changes that were delivered as software updates and, therefore, were under particular scrutiny. If any of the software updates failed, a root cause analysis was conducted, including the identification of the change causing the update to fail. We used the subset of changes from these analyses to identify defect-inducing changes among all changes involved in software updates.

Table 2 summarizes the statistics of the data sets collected from all the projects. The table shows the total number of changes and between parentheses the percentage of all changes that are defect-inducing. In this study, we consider a change to be defect-inducing if it induces one or more defects (since in certain cases, a change may induce more than one defect). The exact number of defects induced is not as important for our models and prediction. The percentage of defect-inducing changes are considerably lower for the commercial systems. Unfortunately, we cannot disclose the percentage of the defect-inducing changes of the commercial systems and the average file size for project C-1, C-2, C-3 and C-4 for confidentiality reasons.

Table 2 shows average values for the number of LOCs at the file-level and the change-level, the number of modified files per change and the number of changes per day. The table also shows the maximum and average number of developers that modified a single file. For example, if file A is modified by developer x, file B is modified by developer y and file C is modified by developer x, y and z, then the maximum and average numbers of developers are 3 and 1.7. This value shows whether a file has only one or more responsible developers.

4.2 Extraction of changes

Non-transaction based SCM systems such as CVS permit a developer to submit only a single source code file per commit. We grouped related one-file commits into one software change using common heuristics [26]. We consider all commits by the same developer, with the same log message, made in the same time window as one change, as suggested by Zimmermann *et al.* [63]. Similar to Zimmermann *et al.* [63], we set the time window to 200 seconds. Note that we do not need this step for the commercial projects, because they use transaction-based SCM systems.

4.3 Identification and recovery of defect-inducing changes

To know whether or not a change introduces a defect, we use the SZZ algorithm [57]. This algorithm links each defect fix to the source code change introducing

3. http://msr.uwaterloo.ca/msr2007/challenge/index.html
4. http://msr.uwaterloo.ca/msr2008/challenge/index.html
5. rsync://columba.cvs.sourceforge.net/cvsroot/columba/
6. rsync://anoncvs.postgresql.org/cvsroot/pgsql/

TABLE 2
Statistics of the studied projects

	Period	The total number of changes		Average LOC File	Average LOC Change	# of modified files per change	# of changes per day	# dev. per file Max	# dev. per file Avg
Bugzilla	08/1998 - 12/2006	4,620	(36%)	389.8	37.5	2.3	1.5	37	8.4
Columba	11/2002 - 07/2006	4,455	(31%)	125.0	149.4	6.2	3.3	10	1.6
Eclipse JDT	05/2001 - 12/2007	35,386	(14%)	260.1	71.4	4.3	14.7	19	4.0
Eclipse Platform	05/2001 - 12/2007	64,250	(14%)	231.6	72.2	4.3	26.7	28	2.8
Mozilla	01/2000 - 12/2006	98,275	(5%)	360.2	106.5	5.3	38.9	155	6.4
PostgreSQL	07/1996 - 05/2010	20,431	(25%)	563.0	101.3	4.5	4.0	20	4.0
OSS-Median	-	27,909	(20%)	310.1	86.7	4.4	9.4	24	4.0
C-1	10/2000 - 12/2009	4,096		-	16.4	2.0	1.2	-	-
C-2	10/2000 - 12/2009	9,277		-	19.2	2.4	2.8	-	-
C-3	07/2002 - 12/2009	3,586		-	16.6	2.0	1.3	-	-
C-4	12/2003 - 12/2009	5,182		-	12.9	1.8	2.4	-	-
C-5	10/1982 - 12/1995	10,961		303.0	39.0	4.8	2.3		-
COM-Median	-	5,182		-	16.6	2.0	2.3	-	-

†The percentage in brackets shows the percentage of defect-inducing changes to all changes.

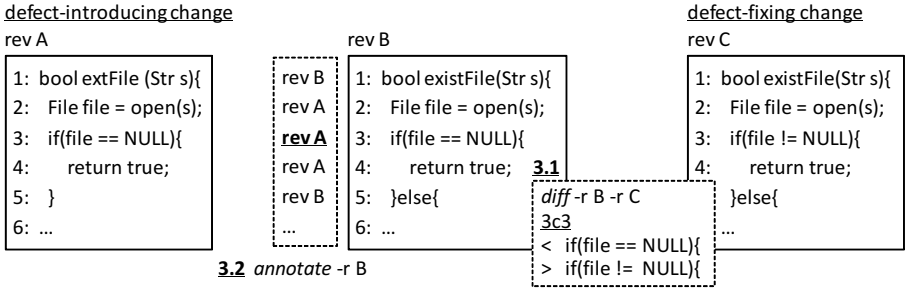


Fig. 1. Example of defect-fixing and defect-introducing changes [23].

the original defect by combining information from the version archive (such as CVS) with the bug tracking system (such as Bugzilla).

The SZZ algorithm consists of three steps. First, it identifies the change that fixes a defect. SZZ searches for keywords such as "Fixed" or "Bug", and for defect identifiers in the change comments. We examine the implications of this assumption in Section 7.

The second step confirms whether that change is really a defect fixing change using information from Bugzilla. We link the changes and the defects in Bugzilla using the numerical defect identifier (e.g., bug 12345) from the change comments. Other heuristics to confirm whether a change is really a defect-fix change are discussed elsewhere [57].

The third step identifies when the defect is introduced. We use the *diff* command to locate the lines that were changed by the defect fix. Then, we use the *annotate* command to trace back to the last revision that changed those patched lines. For example, in Figure 1, we compare the file revision after the defect fixing change (rev. C) and the file revision before the defect fixing change (rev. B) using the *diff* command (3.1) to get the actual lines modified by the defect fix. We then identify the most recent revision (rev. A) in which the modified line #3 of rev. B was changed using the *annotate* command (3.2) and label the change of the identified revision as the defect-introducing change.

Note that, for C-5, all defect-introducing changes and defect-fixing changes were manually identified through a root-cause analysis and stored in C-5's change management system by project experts. We used the manually collected data instead of the SZZ algorithm for that project.

Approximate SZZ (ASZZ). In the case of Columba and PostgreSQL, we use an approximate algorithm (ASZZ) to identify whether or not a change is defect-prone, because the defect identifiers are not referenced in the change logs. In such cases, we have no way of verifying whether or not the change we identified as defect-fixing is really defect-fixing. The algorithm simply looks for keywords that are associated with a defect fixing change (e.g., "Fixed" or "Bug") and assumes that the change fixed a defect without linking the change to a Bugzilla report.

4.4 Data preparation

Dealing with collinearity: Before using the collected factors in our models, we need to make sure to remove highly correlated factors [55]. To deal with the risk of multi-collinearity, we manually remove the most highly correlated factors, then use stepwise variable selection. We manually chose which of the most correlated factors to remove to keep the most fundamental (in our opinion) factors in the model.

Automatic techniques may not work reliably in case of very strong collinearity.

We found that NF and ND, and REXP and EXP are highly correlated. Therefore, we excluded ND and REXP[7] from our model and instead used NF and EXP. Furthermore, we found LA and LD to be highly correlated. Nagappan and Ball [46] reported that relative churn metrics perform better than absolute metrics when predicting defect density. Therefore, we normalized LA and LD by dividing by LT, similar to Nagappan and Ball's approach. We also normalized LT and NUC by dividing by NF since these metrics have high correlation with NF.

Dealing with skew: Since most change measures are highly skewed, we performed a logarithmic transformation to alleviate this effect [55]. We applied a standard log transformation to each measure, except to "FIX", which is a boolean variable.

Dealing with imbalance: Our data sets are relatively imbalanced, i.e., the number of defect-inducing changes represents only a tiny percentage of all changes. This imbalance causes performance degradation of the prediction models [24], [25] if it is not handled properly. To deal with this issue of data imbalance, we use a re-sampling approach for our training data. To achieve this, we decrease the majority class instances (i.e., non-defect-inducing changes in the training data) by deleting instances randomly such that the majority class drops to the same level as the minority class (i.e., defect-inducing changes). Note that the test data is not re-sampled.

5 Case Study Results

In this section, we present our case study results and answer our research questions.

RQ1: How well can we predict defect-inducing changes?

Overview. To answer RQ1, we build a prediction model for the risk of a software change using the factors in Table 1, then evaluate the model's performance. We compare the performance of the prediction models for the open source projects versus the prediction models for the commercial projects.

Validation Technique and Data used. We use 10-fold cross-validation [11]. The data set is divided into ten folds of which the first 9 are used as training data, and the last fold is used as testing data. The model is then trained using the training data and its accuracy is tested using the testing data. Afterwards, the nine other folds each become testing data set and a new model is built and evaluated. The accuracy results are then aggregated across all 10 folds.

7. The lowest Spearman Rank-Order Correlation is 0.91 for the PostgreSQL project. One possible reason for this high correlation is that the projects that we studied contain a large amount of long term contributors. Long term contributors work on the projects for long periods of time making EXP and REXP collinear in these projects.

TABLE 3
Confusion matrix

	True class	
Classified as	Defect	Non defect
Defect	TP	FP
Non defect	FN	TN

Approach. Similar to previous work, we use a logistic regression model to perform our prediction [4], [8], [17]. The logistic regression model outputs a probability, i.e., a value between 0 and 1, for each change. We use a threshold value of 0.5, which means that if the model-predicted probability of a defect is greater than 0.5, the change is classified as defect-inducing, otherwise, it is classified as non-defect-inducing [16], [17].

To avoid over-fitting our models, we select a minimal set of factors to include as independent variables of the models. First, we manually remove highly correlated factors, then we use stepwise variable selection based on Mallows' Cp criterion [31] to remove the remaining collinear metrics and those metrics that do not contribute to the model. This selection technique proceeds by deleting the worst variable from the full model until deleting any remaining variables would no longer be beneficial.

To evaluate the prediction performance, we employ the commonly-used accuracy, precision, recall and F1-measures. These measures can be derived from a confusion matrix, as shown in Table 3. A change can be classified as defect-inducing when it is truly defect-inducing (true positive, TP); it can be classified as defect-inducing when it is actually not (false positive, FP); it can be classified as non-defect-inducing when it is actually defect-inducing (false negative, FN); or it can be correctly classified as non-defect-inducing (true negative, TN).

The accuracy measures the number of correctly classified changes (both the defect-inducing and non-defect-inducing) over the total number of changes. It is defined as: $Accuracy = \frac{TP+TN}{TP+FP+TN+FN}$. Recall is the ratio of correctly predicted defect-inducing changes to the actual defect-inducing changes ($Recall = \frac{TP}{TP+FN}$) and precision is the ratio of correctly predicted defect-inducing changes to all changes predicted as defect-inducing ($Precision = \frac{TP}{TP+FP}$). There is a trade-off between recall and precision. To compare this trade-off between different projects, we use the F1-value, which is the harmonic mean of recall and precision: $F1 - measure = \frac{2 \times Recall \times Precision}{Recall + Precision}$.

These criteria (e.g., precision and recall) depend on the particular threshold used for the classification. Choosing another threshold might lead to different results, hence we calculated the measures for different thresholds. To get an overall idea of the performance

TABLE 4
Summary of prediction performance (RQ1).

	Acc.	Prec.	Recall	F1	AUC	% improved	
		threshold: 0.5				Prec.	AUC
BUZ	67%	54%	69%	60%	74%	48.9%	48.0%
COL	70%	51%	67%	58%	75%	64.3%	51.0%
JDT	69%	26%	65%	37%	74%	87.6%	47.0%
PLA	67%	27%	70%	38%	75%	89.3%	50.3%
MOZ	77%	13%	63%	22%	77%	168.4%	54.9%
POS	74%	49%	65%	56%	79%	97.1%	57.9%
Avg	71%	37%	67%	45%	76%	92.6%	51.5%
Med	69%	38%	66%	47%	75%	88.4%	50.6%
C-1	66%	35%	58%	44%	68%	> 20.0	35.4%
C-2	63%	31%	61%	41%	68%	> 20.0	35.1%
C-3	63%	47%	62%	54%	69%	> 20.0	37.8%
C-4	66%	41%	61%	49%	69%	> 20.0	37.4%
C-5	70%	5%	66%	10%	73%	> 20.0	46.9%
Avg	66%	32%	62%	39%	69%	> 20.0	38.5%
Med	66%	35%	61%	44%	69%	> 20.0	37.4%
All-Avg	68%	34%	64%	43%	73%	> 20.0	45.6%
All-Med	67%	35%	65%	44%	74%	> 20.0	47.0%

"> 20.0" means that the improvement is greater than 20.0%.

across thresholds, we additionally use the AUC (the area under the curve) of ROC (the receiver operating characteristics) [28]. The range of AUC is [0,1] and larger AUC value indicates better prediction performance. Any predictor achieving AUC above 0.5 is more effective than the random predictor. The advantage of the AUC of ROC is its robustness toward imbalanced data, since the ROC is obtained by varying the classification threshold over all possible values.

Results. We performed our predictions and present the results in Table 4. We compare our prediction results to the baseline, which randomly predicts defect-inducing changes. The last column of Table 4 shows the improvement (in percentage) using our predictor over the random predictor, in terms of precision and AUC. Our predictor achieves an average precision of 37% and recall of 67% for open source projects, which translates to an average improvement of 90% over the random predictor.

We note from Table 4 that our prediction models achieve high recall values (average recall of 64%), which is the more important measure of performance in highly skewed data sets (which is the case in all of our projects). For example, the average recall in the prior study by [26] is 62% (Details in Section 6.3). Menzies *et al.* [36] commented on the fact that there are many industrial situations where low-precision and high-recall prediction models are still extremely useful when the data is imbalanced. This is because practitioners can live with the idea of checking slightly more files or changes than needed as long as most "bad situations" are avoided. Furthermore, the precision of conventional defect prediction models at the file-level is typically low (e.g., the precision is 14%) [24], [36].

To see why our approach is actually useful in practice, despite the low precision, consider Table 2. This table shows that in the open source projects, on average, the total number of changes and the number of defect-inducing changes per day is 9.4 and 1.9[8], respectively. Hence, based on our results (37% of precision and 67% of recall), our prediction model flags on average 3.4 changes per day as defect-inducing[9]. Among them, 1.3 changes are truly defect-inducing changes and 2.1 changes are actually not (i.e., false-positive). This result means that the developers need to double check only 2.1 changes per day unnecessarily. For this reason, we believe that the prediction model could be useful in practical settings.

The average results are similar in the case of the commercial projects, achieving an average precision of 32% and recall of 62%. This translates into an improvement of more than 20% over the random predictor. Furthermore, our prediction model (average AUC of 73%) also outperforms the random predictor (average AUC of 50%) in terms of the AUC, i.e., across all thresholds. One commercial project, C5, has a very low precision. This is caused by the extreme imbalance (ratio of only 2.5% of defect-inducing to non-defect-inducing changes) in the data. Even with such a low precision, the prediction model outperformed the random predictor by more than 20%.

Predicting defect-inducing changes is effective in open source and commercial projects. On average, we are able to detect 64% (i.e., recall) of the actual defect-inducing changes with an accuracy of 68% and a precision of 34%.

RQ2: Does prioritizing changes based on predicted risk reduce review effort?

Overview. As pointed out by Mende and Koschke, conventional prediction models at file-level typically ignore the effort needed to review the modified code [35]. A prediction model that prioritizes the largest files as most defect-prone would have a very high recall, i.e., those files likely contain the majority of defects, yet inspecting all those files would take a considerable time. A model that recommends slightly less defect-prone files that are easier to inspect will yield more tangible results faster. Hence, in this question we factor the effort required to review a change into our approach and evaluate the prediction performance of the resulting models.

Whereas for file-level prediction the number of lines of code in a file is used as a measure of the effort required to review the file [2], [37], this paper uses

8. the value 1.9 is calculated by 9.4 multiplied by 0.2 (i.e., 20%, the percentage of defect-inducing changes) in Table 2

9. The number of defect-inducing changes classified by the prediction model is $TP + FP$ in Table 3. $TP + FP$ is calculated as $\frac{TP}{Precision}$, since $Precision = \frac{TP}{TP+FP}$. As the precision is 37%, we would like to obtain TP. TP is calculated as $Recall * (TP + FN)$, that is, TP is 1.26, since $Recall = \frac{TP}{TP+FN}$, the recall is 67% and $TP + FN$ (i.e., the number of defect-inducing changes) is 1.9. As a result, $TP + FP$ is 3.40 ($= \frac{1.26}{0.37}$ as $\frac{TP}{Precision}$).

the total number of lines modified by a change. For example, if a change adds and deletes many lines, it requires more time to validate than a change that adds or deletes less lines. This type of effort-aware prediction offers a more practical adoption-oriented view of defect prediction results.

Approach. For our effort-aware evaluation, we first build naive models based on our RQ1 models as a first attempt towards effort-aware models. We then build customized effort-aware models, and compare those to random models that we use as a baseline[10]. Similar to RQ2, we use 10-fold cross-validation as validation technique.

Prior work [50] showed that the majority of the defects (approximately 80%) are contained in a small number of files (approximately 20%). Motivated by this prior work, we assume that a limited amount of resources (i.e., 20%) is available to review the changes flagged by us as being defect-inducing. We then determine the maximum number of defects developers could review based on our prediction models using 20% of the total available effort (Figure 2). A naive way to do this, is to prioritize the list of changes in descending order based on the predicted logistic probabilities. Then, we take the most risky changes that we can review with 20% effort from the list and count how many defects these changes would contain. We call this the LR model. As we will show later, this LR model does not perform well.

To improve the prediction, we introduce a customized prediction model based on effort-aware *linear regression* (EALR), similar to Mende and Koschke [35]. In such a prediction model, we predict for $R_d(x) = \frac{Y(x)}{Effort(x)}$ instead of just for Y(x), where Y(x) is 1 if the change is defect-inducing and 0 otherwise, and Effort(x) is the amount of effort required by the change (i.e., the number of modified lines). Then, we prioritize the list of changes based on $R_d(x)$ in descending order. We take all of the changes that we could review with just 20% effort off the list and count how many defects these changes would contain. Note that we use a linear regression model instead of a logistic regression model, since $R_d(x)$ is not binary (i.e., defect or not), but numeric. In addition, similar to earlier work [56], we excluded lines of code added/deleted from the independent variables in the EALR model, since lines of code added/deleted together make up the effort value in the dependent variable of the EALR model.

Finally, we also randomly generate a list of changes and use it as a baseline to compare the prediction performance of our models.

The accuracy measures the percentage of detected defect-inducing changes to all defect-inducing changes when using 20% of all effort (We call *accuracy* in RQ2). We selected 20% effort as the cut-off value. However, selecting another cut-off value might lead to different results. Therefore, we additionally use the P_{opt} evaluation metrics [34] to evaluate the prediction performance of models. P_{opt} is defined as the area Δ_{opt} between the effort(i.e., churn)-based cumulative lift charts of the optimal model and the prediction model (Figure 2). In the optimal model, all changes are ordered by the decreasing actual fault density. While in the predicted model, all changes are ordered by decreasing predicted value. As shown in the following equation, a larger P_{opt} value means a smaller difference between the optimal and predicted model.

$$P_{opt} = 1 - \Delta_{opt} \quad (1)$$

The range of Δ_{opt} is [0, AUC(Optimal model)]. The formula for Δ_{opt} above has been normalized by the maximum Δ_{opt}, similar to Kamei *et al.* [23]. Therefore, the range of normalized P_{opt} is [0,1] and any predictor achieving the P_{opt} above 0.5 is more effective than the random predictor.

Results. We present the results of our effort-aware predictions in Table 5. It can be observed that the simple LR predictions do not perform well in terms of the accuracy (i.e., the percentage of detected defect-inducing changes to all defect-inducing changes when using 20% of all effort). The average accuracy is 5% for open source projects and 8% for the commercial projects. These models perform worse than a basic random model.

However, the customized effort-aware models (i.e., EALR) highly improve the performance. They achieve average prediction accuracies of 28% for open source projects, 43% for the commercial projects and 35% for all projects. That is, the EALR is 39%=(27.8-20.0)/20.0 and 114.5%=(42.9-20.0)/20.0 better than the random predictor in OSS and COM. On the other

10. The random model represents the average over infinite random models generated. Therefore, when plotting the cumulative lift charts like Figure 3, the line for the random model is an ideal straight line.

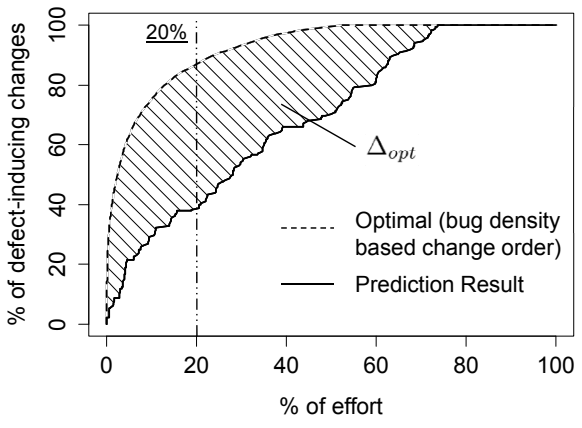


Fig. 2. Example of Effort-based Cumulative Lift Chart for Bugzilla.

TABLE 5
Summary of prediction performance (RQ2).

	LR		EALR	
	Acc.	P_{opt}	Acc.	P_{opt}
Bugzilla	11%	45%	38%	72%
Columba	2%	19%	39%	57%
JDT	1%	27%	20%	48%
Platform	4%	34%	30%	55%
Mozilla	6%	38%	15%	45%
PostgreSQL	3%	27%	26%	51%
Avg	5%	32%	28%	55%
Median	4%	31%	28%	53%
C-1	9%	36%	29%	60%
C-2	5%	32%	47%	72%
C-3	6%	27%	50%	75%
C-4	8%	33%	38%	66%
C-5	14%	34%	51%	71%
Avg	8%	32%	43%	69%
Median	8%	33%	47%	71%
All-Avg	6%	32%	35%	61%
All-Median	6%	33%	38%	60%

{Acc., P_{opt}} for the random predictor is {20, 50}

hand, for P_{opt}, the EALR is 9.6%=(54.8-50.0)/50.0 and 37.8%=(68.9-50.0)/50.0 better than the random one. In short, the approach substantially improves over a random predictor for the accuracy (both OSS and COM) and P_{opt} (COM). P_{opt} also improves for OSS, but not as dramatic (less than 10%).

It is important to note that the result of RQ1 is different from the result of RQ2. RQ1 is based on the non-effort aware model and achieves 64% of recall on average, whereas, RQ2 is based on the effort aware model and achieves 35% of accuracy on average. However, it's not always true that the 35% defect-inducing changes detected by the effort aware model feature amongst the 64% defect-inducing changes detected by the non-effort aware model.

Figure 3 shows the churn-based cumulative lift chart of EALR for C-3, which shows the best performance in the experiment. The x-axis shows the cumulative code churn (i.e., the number of code added and deleted by a change) and the y-axis shows the cumulative number of defect changes. The solid and dashed line plot the cumulative lift chart for changes ordered by decreasing predicted $R_d(x)$ or randomly, respectively. The figure shows that EALR is more effective than randomness for the same amount of effort. Using 20% of all available effort, EALR is able to detect 50% of all defect-introducing changes, compared to 20% for the random model.

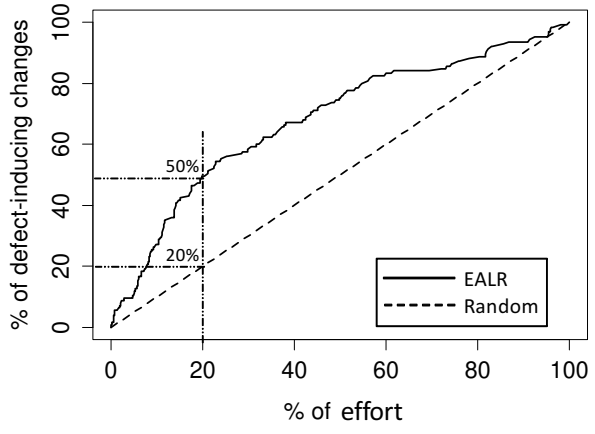


Fig. 3. Churn-based Cumulative Lift Chart for C-3.

What cut-off value provides the best return on investment? Similar to previous work [23][35], we selected 20% effort as the cut-off value. However, an interesting question is what cut-off value provides the best return on investment. To answer this question, we use the number of detected defect-inducing changes divided by the required effort as a performance measure (the higher, the better the return). Then, we calculate the performance by changing the effort from 10% to 90% in increases of 10%.

Our results showed that 10% of the effort provides the best return on investment for all projects except one open source project (Mozilla) and one industrial project (C-1). In C-1, we find that 20% of the effort provides the best return on investment. In Mozilla, we do not find a cut-off value that provides an optimal return on investment, i.e, our prediction model detects less than N% defect-inducing changes when N% effort is spent.

20% of the effort can detect on average 35 % of all defect-inducing changes when using an effort-aware prediction model.

RQ3: What are the major characteristics of defect-inducing changes?

Overview. To address RQ3, we analyze and compare the regression coefficients of the logistic models from RQ1 and RQ2. In order to shed light on and better understand the factors that influence whether or not a change is defect-inducing in different contexts (i.e., open source vs. commercial), we highlight the most important factors in open source projects and compare them to commercial projects.

Validation Technique and Data used. In contrast to RQ1 and RQ2, RQ3 uses the entire data set to build the logistic regression models, since we only need to use the regression coefficients of the logistic regression models and do not perform any prediction.

Approach. To interpret the regression coefficients of the RQ1 logistic models, we use the odds ratio of each factor [8], [55]. The odds ratio of a particular factor is the exponent of the logistic regression coefficient. It corresponds to the ratio by which the probability of a change being defect-inducing increases when the factor increases by one unit. An odds ratio greater than one indicates a positive relationship of a factor on the chance of a change being defect-inducing (i.e., factor is risk-increasing), whereas an odds ratio that

TABLE 6
The impact of change factors on defect-inducing changes in the 6 open source projects and 5 industry projects.

Metrics	All		OSS								COM						
name	+	−	+	−	BUZ	COL	JDT	PLA	MOZ	POS	+	−	C-1	C-2	C-3	C-4	C-5
NS	4	0	3	0	2.09			1.62		1.36	1	0	2.62				
NF	11	0	6	0	2.95	3.00	2.62	3.07	4.29	5.61	5	0	1.62	1.33	4.26	2.10	1.87
Entropy	1	5	0	4		0.44		0.60	0.49	0.62	1	1			0.26		2.28
LA/LT	10	0	6	0	5.62	4.13	8.72	2.74	2.85	20.43	4	0	1.23	1.24	1.35	1.36	
LD/LT	1	0	1	0				1.25			0	0					
LT/NF	9	0	5	0		1.54	1.19	1.38	1.15	1.24	4	0	1.37	1.36	1.34	1.67	
FIX	4	0	4	0	4.54		2.20	3.37	2.07		0	0					
NDEV	3	2	0	2				0.81		0.74	3	0		1.23	2.33	1.73	
AGE	2	6	0	6	0.72	0.84	0.88	0.90	0.85	0.76	2	0	1.13	1.11			
NUC/NF	3	0	3	0				4.50	3.20	7.40	0	0					
EXP	3	3	2	3	0.89	0.79	1.33	1.14	0.92		1	0		1.85			
SEXP	3	3	1	3		1.24	0.72	0.94	0.94		2	0			1.15	1.11	

TABLE 7
The regression coefficients of change factors in the effort-aware models.

Metrics	All		OSS								COM						
name	+	−	+	−	BUZ	COL	JDT	PLA	MOZ	POS	+	−	C-1	C-2	C-3	C-4	C-5
NS	0	0	0	0							0	0					
NF	6	4	6	0	0.03	0.07	0.04	0.05	0.05	0.04	0	4	-0.07	-0.06	-0.07	-0.06	
Entropy	0	6	0	6	-0.10	-0.33	-0.17	-0.24	-0.21	-0.12	0	0					
LA/LT	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
LD/LT	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
LT/NF	3	4	3	3	-0.01	0.03	-0.01	0.01	0.01	-0.01	0	1					-0.01
FIX	4	0	4	0	0.07		0.04	0.06	0.01		0	0					
NDEV	2	7	0	6	-0.04	-0.15	-0.06	-0.06	-0.04	-0.09	2	1		-0.07	0.06	0.07	
AGE	1	6	0	6	-0.02	-0.04	-0.03	-0.03	-0.02	-0.02	1	0					0.00
NUC/NF	0	5	0	5		-0.53	-0.26	-0.32	-0.22	-0.17	0	0					
EXP	3	2	2	1		-0.03	0.03	0.01			1	1		0.05			-0.01
SEXP	1	4	1	3		0.04	-0.02	-0.01	-0.01		0	1	-0.02				

is smaller than one indicates a negative relationship (i.e., factor is risk-decreasing).

For example, in the case of lines added (LA), the odds ratios would indicate the increase (or decrease if the odds ratio is less than one) in the probability of a change being defect-inducing when LA is increased by one unit (i.e., 1 line). Since we apply a standard log transformation to all factors (except the boolean factor "FIX"), we first need to transform back the factors' coefficient to a regular scale (i.e., e^x) to obtain the actual odds ratios.

To interpret the regression coefficients of the EALR models of RQ2, we examine the coefficients of the linear regression models to determine the most important factors. Because in this case the response is not a probability, the odds ratio approach is not applicable. Instead, we count the number of times a factor positively or negatively contributes to a change being defect-inducing across the 11 systems.

Results. Table 6 shows the odds ratios of change factors in the different projects. For example, the odds ratio of the NS factor in the Bugzilla project is 2.09. In the cases where the odds ratio value is blank, the factor did not prove to be statistically significant ($p < 0.05$) for that project (e.g., NS for the JDT project). We used the stepwise technique, detailed in Section 5 to evaluate whether or not a factor is statistically significant.

We characterize the effect of the different factors on the different projects and label the factors based on whether their odds ratio indicates a positive (with a "+") or a negative (with a "-") relationship, similar to Mockus *et al* [42].

We can see that for the open source projects, the number of files (NF), the relative churn metrics (LA/LT and LT/NF) and whether or not the change was to fix a defect (FIX) are the most important, risk-increasing factors. The time when the files were last changed (AGE) is also a very important factor, however, it is risk-decreasing. This means that the more recent a file's last change is, the higher the chance that the current change will induce a defect.

Similar to the open source projects, the number of files (NF) and the relative churn metrics (LA/LT and LT/NF) are risk-increasing in the commercial systems. In addition, the number of developers that touched a file (NDEV) in the past is also risk-increasing. Interestingly, the time when the files were last changed is risk-increasing in commercial projects. There are no risk-decreasing factors in the commercial projects (except for one case of Entropy in C-5).

Table 7 shows the regression coefficients for the change factors in the effort-aware models. The results show that the number of files (NF) and whether or not the change fixes a defect (FIX) are risk-increasing in open source projects. This finding is consistent with

our earlier findings on the odds ratios of Table 6. On the other hand, the diffusion factors (Entropy), the number of developers (NDEV), AGE and NUC/NF are risk-decreasing. The finding that AGE has a negative impact is consistent with our findings on the odds ratios of Table 6.

For the commercial projects, only the diffusion factors (NF) are consistently important for change risk. These size factors are all risk-decreasing in the effort-aware models.

We find that NF and FIX are risk-increasing factors and AGE is a risk-decreasing factor in RQ1 and RQ2 for open source projects. We also find that the diffusion factors are consistently important in RQ1 and RQ2 for commercial projects, but have a different effect in RQ1 and RQ2 (risk-increasing and -decreasing, respectively).

6 Discussion

In this section, we further discuss our findings.

6.1 Why are open source and commercial systems different?

As shown in Table 6, the odds ratios differ between the OSS and Commercial (COM) systems.

First, it is important to note that different projects may have different reasons that explain the change risk. They may depend on the nature of the project, the practices in use, and on the types of defects that are being considered. The models of the risk depend on the particular measures that are used to explain the risk. While we can separate the OSS and COM projects into two broad classes, there appear to be substantial differences even among the projects within a class. From the odds ratios we can see that some features are common for an entire project class, while others vary by project.

A critical difference between OSS and Commercial settings is the nature of defects that are reported (and hence studied). In commercial code, typically only the defects reported by customers on released versions of the software are modeled (e.g., Mockus and Weiss [41], Nagappan *et al.* [47], Cataldo *et al.* [8], Mockus [38]), yet in OSS all defects are considered, because it is not always possible to separate which defects are detected by customers for a stable release, and which ones are detected by developers during development. For example, to make a more direct comparison, Mockus *et al.* [39] compared OSS defects to post-feature-test defects in a commercial software system. While the difference in types of defects, by itself, does not necessarily explain the observed variation in the odds ratios, it is possible that for the same types of defects in OSS and in Commercial software, the differences would, perhaps, disappear or, at least, be less pronounced. Furthermore, the operationalization of defect-inducing changes differs among the projects, and that may account for some of the observed variation as well.

We also should note that most of the prior work focused on predicting defects at the file level, not at the code commit level as we do now. Only Mockus and Weiss and Kim *et al.* considered the prediction at the commit level [26][41]. While we make comparisons to prior work, the two key differences "OSS vs. Commercial defects" and "file vs. commit defects" need to be noted.

The third important aspect is the particular set of factors that are included in the model. A factor (independent variable) in a regression model can only be interpreted conditionally on the values of other factors. The same factor may have a positive or a negative influence depending on what other factors are in the model. Below we discuss the factors that have the most salient differences.

NDEV: In this study, NDEV decreases risk for two of the six OSS projects (Table 6), while it increases risk for three of the five commercial systems. In prior work, Mockus and Weiss found no effect of NDEV on change risk in a very large commercial system [41]. The factor set used was not identical to our models. In particular, it also considers duration (i.e., time difference between the first and last delta of an MR). It is possible that by adding duration to our models we would no longer need to use the number of developers (i.e., it would not help explain the risk beyond what the duration factor explains). Based on the "many eyeballs" theory of Raymond in open source software projects [54], Rahman and Devanbu studied the relationship between the number of authors (i.e., developers) and the probability of defect-inducing changes for four open source software projects. They found that the implicated code is less likely to involve contributions from multiple developers. That study also included other factors such as the number of commits by code owner and non code owner. It is consistent with our results for two of our OSS projects.

On the other hand, Bird *et al.* examined the relationship between ownership and software failures in two large industrial software projects: Windows Vista and Windows 7 [5]. They found that the number of developers has a strong positive relationship with failures. The model used in the study also included source code complexity, size, and churn and the observations in the model were "binaries" not individual files. Bird *et al.* results, thus, are consistent with our findings for three of the five commercial projects. In commercial development, explicit code ownership is more common, and increasing the number of developers might mean that more non-experts (i.e., non-owners) touch the code, thus decreasing the software quality. In particular, for commercial projects studied by Cataldo *et al.* [8] and by Mockus [38], the size of the

organization and of the workflow network increased the chances that the file will contain a defect. These studies contained a comprehensive set of factors including file size, and the number of past changes. In summary, it appears that the observed differences may be caused by the differences in code ownership between OSS and COM projects.

FIX: The odds ratio indicates a positive relationship for four out of six OSSs projects (Table 6). The definition of what constitutes a fix may be paramount here. Few of defect-inducing changes were enhancements in the commercial software, while in OSS a substantial portion of defect-inducing changes were a part of new-feature development. This is because the pre-release fixes done as a part of new feature development were not considered as defects in the commercial projects.

AGE: The odds ratio is **less than one** for all OSS projects (i.e., the **older** the last change is, the less likely a defect was introduced by it), while it is greater than one for two of the five commercial systems (Table 6). Nagappan *et al.* reported that a change burst, which is a sequence of consecutive changes, has the highest predictive power for defect-prone components, but only in the analyzed commercial project, not in OSS systems [48]. The change bursts require a controlled change process, which is most often found in a commercial settings and is related to the system test phase before the release to customers. In OSS systems such dramatic differences are less common and change frequency is more uniform over time. A plausible conjecture for the observed variation may be attributed to the differences in the development process and in the definition of a defect. In particular, in OSS projects the defect may be discovered during the development process, for example during continuous integration. This makes it more likely that recent changes will be be more likely to introduce a defect. In commercial projects, the post-release defects manifest themselves only after the release. If the most active development activities occur much earlier than the release date (very common in commercial setting, for example, the extended code freezes prior to release), the older changes that occurred during the main development phase will account for the bulk of the post-release defects.

EXP: We find that EXP increased risk in two OSS and one commercial project and decreased risk in three OSS projects. SEXP increased risk in one OSS and two commercial projects and decreased it in one OSS project. These differences seem to correspond to variations in project practices. Mockus and Weiss found that increasing experience (EXP) decreased the chance of defect in a change [41], whereas in another study [38], Mockus found that increased experience increases the chances of a defect, presumably because experienced developers were more likely to work on more complex parts of the system (see, e.g., Zhou and Mockus [61]) and, thus, were more likely to introduce a defect [38].

TABLE 8
Comparison between our results and Kim *et al.*'s results

	Our results			Kim *et al.*'s results		
	Prec.	Recall	% of defects	Prec.	Recall	% of defects
Bugzilla	54%	69%	36%	86%	85%	74%
Columba	51%	67%	31%	58%	59%	29%
JDT	26%	65%	14%	–	–	–
Platform	27%	70%	14%	61%	61%	10%
Mozilla	13%	63%	5%	57%	63%	30%
PostgreSQL	49%	65%	25%	43%	44%	24%
Avg	37%	67%	20%	61%	62%	33%

6.2 Why do the LR models perform poorly?

Mende *et al.* [34] reported that the performance of the LR models is not worse than that of the random model, while we found that the LR models perform much worse compared to the random model.

The main reasons for these contrasting findings could be the differences in (1) granularity of the prediction and (2) prediction target. Mende *et al.*'s prediction granularity is a file, while our prediction granularity is an individual change. Changes cover small parts of multiple files, while files are larger and more cohesive. Target-wise, Mende *et al.* predict the *number* of defects in a file, while we predict *whether or not* a change introduces a defect, i.e., whether or not the number of defects in a change is at least one. Whether we spend a lot of effort on a large change or little effort on a smaller change, if either change has at least one defect, they are equally buggy from the point of view of our models.

6.3 How do our results compare to prior work?

The closest work to ours is the work by Kim *et al.* [26]. Therefore, in this section we compare our results to the results achieved by Kim et al. Table 8 summarizes both our results and those of Kim *et al.*'s result [26]. The precision of our study is substantially lower than the precision of Kim *et al.*, especially in the case of Mozilla.

However, there are two key differences in the evaluation setting between our study and Kim *et al.*'s study. First, as described in Section 4.3, we confirm that each change identified by SZZ algorithm is traceable to a Bugzilla issue report, unlike in Kim *et al.*, where all SZZ identified changes are assumed to be defect inducing. Therefore, the percentage of bug-inducing changes in our Mozilla dataset (5%) is much lower than for Kim *et al.* (29.9%). It is very important to stress that performance of predictors with such low frequency of occurrence (5%) is much lower than for predictors of more frequent events (29.9%).

Second, Kim *et al.* used 500 revisions (or 250 revisions) for each project, while we used all revisions, i.e., in between 4,455 and 98,275 changes (i.e., the whole data set). It might be difficult to build a high performance prediction model using data that is collected for such a long period of time, since the characteristics of the project might be change over time. That said, we feel that using all of the data provides a much more realistic evaluation scenario.

6.4 Does JIT quality assurance lead to more reviewing work?

One limitation the reader might point out is that, at first sight, JIT quality assurance requires practitioners to review *more* source code than traditional quality assurance. This is because one line in the shipped version of a software system could have been changed ten times, triggering (in the worst case) ten reviews according to JIT quality assurance. Hence, reviewing a software system once at the end of the development cycle seems more advantageous.

However, this argument is not valid, for a number of reasons. First, at the end of the development cycle, most changes are no longer fresh, hence developers using file-level quality assurance need to consult more context to understand the end result of the various changes. In other words, many of the intermediate changes need to be checked again anyway. In JIT quality assurance, changes are fresh and hence easier to review. Second, if a code region is found to be defect-prone, it is much harder to identify who is responsible for fixing it, since many different developers might have changed that region during development. For example, the column "# of dev. per a file" in Table 2 shows that the maximum number of developers touching a particular file ranges from 10 to 155 across the open source systems we used. In JIT quality assurance, each change is owned by exactly one developer, making it trivial to assign fix activities.

Third, the companies owning C-1, C-2, C-3, C-4 and C-5 never perform end-of-cycle code inspections, but instead inspect every code change. The reason why postponing code inspection until the end is not sustainable is that development might be building on defective changes for weeks without noticing it, possibly resulting in long release delays when the system eventually has to be fixed. Our JIT models help companies to reduce the number of changes to inspect by recommending (predicting) only the most risky ones.

6.5 What are the characteristics of defect-inducing changes that are classified as non-defect-inducing?

To better understand the characteristics of defect-inducing changes and their impact for false-negatives (i.e., the defect-inducing changes that were not classified as defect-inducing), we plotted scatter plots of the 5 factors that were selected the most in our models, i.e., NF, LA/LT, FIX, AGE and EXP in Table 6. Since we used 10-fold cross-validation in RQ1, we obtained 10 times 5 scatter plots for each project, and we manually analyzed each of them. Our analysis showed that the changes touching a small number of files and the changes that are not related to defects are most likely to be false-negatives in our models. It makes sense that changes touching a small number of files lead to false negatives, since such files traditionally are not considered to be error-prone. The fact that changes unrelated to defects lead to false negatives is not that intuitive.

7 Limitations and Threats to Validity

Construct validity. A first threat to the validity of our work, is that, we assume that all the defects that we used in the experiment had the same weight, primarily because the assigned priorities and severities tend to be unreliable. For example, Herraiz *et al.* reported that one of the major problems when assigning severity to a defect is that a reporter might assign different severities to defects that require the same severity [20]. Furthermore, in Mockus *et al.*'s study [39], the priority of defects was unrelated to the time it takes to resolve the defect, because the defect was set by reporters who had less experience than core developers. The work of Shihab *et al.* focused on determining the most important defects (breakages and surprises) [56], but that approach could not be applied to all the projects we are considering in this work. In any case, note that all the defects that we studied were fixed by the developers, and, therefore, were at least important enough to be fixed.

For CVS, we grouped related one-file commits into one software change using common heuristics [63]. These common heuristics seem to fit well to our research environment, since the changes that we consider are fine-grained and we have access to the commit messages in the CVS repositories. If the changes would have been coarse-grained (e.g., a transaction for one month), with no access to commit messages, Vanya *et al.*'s approach could be more applicable [58].
External validity. Although we use data sets collected from 11 large, long-lived systems (six open source and five commercial projects), these projects might not be representative of all projects out there. However, since they cover a wide range of domains and sizes, we believe that our work significantly contributes to the validation of empirical knowledge about Just-In-Time quality assurance and effort-aware models.

There may be other features that we did not measure. For example, we expect that the type of changes (e.g., refactoring [43], [53]), the role of the developer who modified the file (e.g., core or not) and the

number of defect-inducing changes in which the files have been touched in the past might influence the probability of a defect. Further studies using other factors might further improve our predictions.

Internal validity. The SZZ algorithm is commonly used in defect prediction research [26], [43], but SZZ has its own limitations. For example, if a defect is not recorded in the CVS log comment or the keywords for defect identifiers that we use (e.g., Bug and Fix [24]) do not cover those used in the comment, there is no way of mapping the defect back to the defect-inducing change. We believe that the usage of an approach to recover missing links [59] is required to improve the accuracy of the SZZ algorithm to identify defect-inducing changes from repositories.

This study uses the number of lines of code modified in a change as a measure of the effort required to review a change, similar to Mende and Koschke [35]. At the minimum, the number of LoC changed represents a lower bound for the effort, since at least the changes themselves should be checked during review before consulting other files. This assumption is still an open question for future work. Also, replicated studies using other measures of effort will be useful to evaluate the generalizability of our findings.

Similar to previous work [14], [32], [40], [41], we use the number of changes as the developer's project experience. Furthermore, we believe that the number of commits is a better measure of developer's experience than the number of days she has contributed to the project, because the experience with the code increases with each modification task. On the other hand, using the number of days she has contributed to the project is not as desirable since a developer may perform one change and disappear for 1 year and come back. In such a case, even if the developer has spent many days in the project, her experience may not have increased much.

Statistical conclusion validity. The threat to statistical conclusion validity arises when inappropriate statistical tests are used. Before using the collected factors in our models (i.e., logistic regression in RQ1 and linear regression in RQ3), we removed highly correlated factors due to multi-collinearity among the factors.

8 Conclusions

In this paper, we empirically evaluated a "Just-In-Time (JIT) Quality Assurance" approach to identify in real-time software changes that have a high risk of introducing a defect. Our study validates this change-level prediction through an extensive study on 6 open source and 5 commercial projects. We operationalize a wide array of factors (metrics) for all 11 projects, thus providing an example of how each factor can be calculated under a variety of open source and commercial problem tracking and version control systems.

Our findings show that different factors are effective for open source and for commercial projects in RQ1 (i.e., how well can we predict defect-inducing changes?) and RQ2 (i.e., does prioritizing changes based on predicted risk reduce review effort?). The number of files and whether or not the change fixes a defect are risk-increasing factors and the average time interval between the last and the current change is a risk-decreasing factor in RQ1 and RQ2 for open source projects. The size factors are consistently important in RQ1 and RQ2 for commercial projects. That is, the churn factors are risk-increasing and -decreasing factors in RQ1 and RQ2, respectively. We also found that a change-level prediction model can predict changes as being defect-prone or not with 68% accuracy, 34% precision and 64% recall. Furthermore, when factoring in the effort required to review the changes into our predictions, we found that using only 20% of all effort suffices to identify 35% of all predicted defect-inducing changes.

With JIT quality assurance, developers can better focus their effort on the changes that are most likely to induce a defect. We expect "Just-In-Time Quality Assurance" to provide an effort-minimizing way to focus on the most risky changes and, thus, reduce the costs of building high-quality software.

REPEATABILITY: To enable repeatability of our work, and invite future research, we will be providing all OSS datasets and R scripts that have been used to conduct this study at http://reseaerch.cs.queensu.ca/~kamei/jittse/jit.zip.

ACKNOWLEDGEMENT: This research was conducted as part of the Grant-in-Aid for Young Scientists(A) 24680003 and (Start-up) 23800044 by the Japan Society for the Promotion of Science.

References

[1] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 8–17, 2006.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *The Journal of Systems and Software*, 83(1):2–17, 2010.

[3] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE'07)*, pages 19–26, 2007.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[5] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 4–14, 2011.

[6] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Trans. Softw. Eng.*, 19(11):1028–1044, 1993.

[7] L. C. Briand, J. Wüst, S. V. Ikonomovski, and H. Lounis. Investigating quality factors in object-oriented designs: an industrial case study. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'99)*, pages 345–354, 1999.

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

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[10] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'10)*, pages 31–41, 2010.

[11] B. Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, 1983.

[12] K. E. Emam, W. Melo, and J. C. Machado. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56:63–75, February 2001.

[13] J. Eyolfson, L. Tan, and P. Lam. Do time of day and developer experience affect commit bugginess. In *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, pages 153–162, 2011.

[14] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity.

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

[16] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'10)*, pages 495–504, 2010.

[17] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005.

[18] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic review of fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 99(PrePrints), 2011.

[19] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'09)*, pages 16–24, 2009.

[20] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'08)*, pages 145–148, 2008.

[21] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 21, 2007.

[22] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, pages 11–18, 2008.

[23] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort aware models. In *Proc. Int'l Conf. on Software Maintenance (ICSM'10)*, pages 1–10, 2010.

[24] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K. Matsumoto. The effects of over and under sampling on fault-prone module detection. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'07)*, pages 196–204, 2007.

[25] T. M. Khoshgoftaar, X. Yuan, and E. B. Allen. Balancing misclassification rates in classification-tree models of software quality. *Empirical Softw. Engg.*, 5(4):313–330, 2000.

[26] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34(2):181–196, 2008.

[27] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Trans. Softw. Eng.*, 35:293–304, 2009.

[28] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, 2008.

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

[30] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB inc. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 413–422, 2006.

[31] C. L. Mallows. Some comments on CP. *Technometrics*, 42(1):87–94, 2000.

[32] S. Matsumoto, Y. Kamei, A. Monden, and K. Matsumoto. An analysis of developer metrics for fault prediction. In *Proc. Int'l Conf. on Predictive Models in Softw. Eng. (PROMISE'10)*, pages 18:1–18:9, 2010.

[33] T. J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, 1976.

[34] T. Mende and R. Koschke. Revisiting the evaluation of defect prediction models. In *Proc. Int'l Conf. on Predictor Models in Softw. Eng. (PROMISE'09)*, pages 1–10, 2009.

[35] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Proc. of European Conf. on Software Maintenance and Reengineering(CSMR'10)*, pages 109–118, 2010.

[36] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors' ". *IEEE Trans. Softw. Eng.*, 33:637–640, 2007.

[37] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, 2010.

[38] A. Mockus. Organizational volatility and its effects on software defects. In *Proc. the Int'l Symposium on Foundations of Software Engineering (FSE'10)*, pages 117–126, 2010.

[39] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[40] A. Mockus and J. Herbsleb. Expertise browser: A quantitative approach to identifying expertise.

[41] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[42] A. Mockus, P. Zhang, and P. L. Li. Predictors of customer perceived software quality. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'05)*, pages 225–233, 2005.

[43] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'08)*, pages 181–190, 2008.

[44] J. C. Munson and S. G. Elbaum. Code churn: A measure for estimating the impact of code change. In *Proc. Int'l Conf. on Software Maintenance (ICSM'98)*, page 24, 1998.

[45] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, 1992.

[46] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 284–292, 2005.

[47] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. Int'l Conf. on Softw. Eng. (ICSE'06)*, pages 452–461, 2006.

[48] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE'10)*, pages 309–318, 2010.

[49] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw. Eng.*, 22(12):886–894, 1996.

[50] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

[51] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Programmer-based fault prediction. In *Proc. Int'l Conf. on Predictor Models in Softw. Eng. (PROMISE'10)*, pages 19:1–19:10, 2010.

[52] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.

[53] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *Proc. Int'l Working Conf. on Mining Software Repositories (MSR'08)*, pages 35–38, 2008.

[54] E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Oreilly & Associates Inc, 2001.

[55] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the Eclipse

project. In *Proc. Int'l Symposium on Empirical Softw. Eng. and Measurement (ESEM'10)*, pages 4:1–4:10, 2010.

[56] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 300–310, 2011.

[57] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proc. Int'l Conf. on Mining Software Repositories (MSR'05)*, pages 1–5, 2005.

[58] A. Vanya, R. Premraj, and H. v. Vliet. Approximating change sets at philips healthcare: A case study. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR'11)*, pages 121–130, 2011.

[59] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'11)*, pages 15–25, 2011.

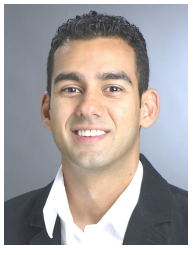
[60] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 26–36, 2011.

[61] M. Zhou and A. Mockus. Developer fluency: achieving true mastery in software projects. In *Proc. the Int'l Symposium on Foundations of Software Engineering (FSE'10)*, pages 137–146, 2010.

[62] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proc. European Softw. Eng. Conf. and Symposium on the Foundations of Softw. Eng. (ESEC/FSE'09)*, pages 91–100, 2009.

[63] T. Zimmermann and P. Weisgerber. Preprocessing cvs data for fine-grained analysis. In *Proc. Int'l Workshop on Mining Software Repositories (MSR'04)*, pages 2–6, May 2004.

Yasutaka Kamei is an assistant professor at Kyushu University in Japan. He has been a research fellow of the JSPS (PD) from July 2009 to March 2010. From April 2010 to March 2011, he was a postdoctoral fellow at Queen's University in Canada. His research interests include empirical software engineering, open source software engineering and mining software repositories (MSR). He received the B.E. degree (2005) in Informatics from Kansai University, and the M.E. degree (2007) and Ph.D. degree (2009) in Information Science from Nara Institute of Science and Technology.



Emad Shihab is an assistant professor in the department of software engieering at the Rochester Institute of Technology. He received his PhD in Computer Science from the School of Computing, Queenfs University, his BEng. and MASc. in Electrical and Computer Engineering from the University of Victoria. His research interests are in Mining Software Repositories, Software Maintenance, Software Quality Assurance, Empirical Software Engineering, and Software Architecture. He worked as a software researcher at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. He held an NSERC Alexander Graham Bell Canada Graduate Scholarship (CGS) and a Google GRAD CS Forum. He served as co-publicity chair for the 2011 International Workshop on Empirical Software Engineering in Practice (IWESEP) and as program chair of the 2012 Mining Software Repositories (MSR) Challenge Track.

Bram Adams is an assistant professor at the École Polytechnique de Montréal, where he heads the MCIS lab on Maintenance, Construction and Intelligence of Software. He obtained his PhD at Ghent University (Belgium), and was a postdoctoral fellow at Queen's University (Canada) from October 2008 to December 2011. His research interests include software release engineering in general, and software integration, software build systems, software modularity and software maintenance in particular. His work has been published at premier venues like ICSE, FSE, ASE, ESEM, MSR and ICSM, as well as in major journals like EMSE, JSS and SCP. Bram has co-organized 4 international workshops, has been tool demo and workshop chair at ICSM and WCRE, and was program chair for the 3rd International Workshop on Empirical Software Engineering in Practice (IWESEP 2011) in Nara, Japan. He currently is program co-chair of the ERA-track at the 2013 IEEE International Conference on Software Maintenance (ICSM), as well as of the 2013 International Working Conference on Source Code Analysis and Manipulation (SCAM), both taking place in Eindhoven, The Netherlands.

Ahmed E. Hassan is the NSERC/RIM Industrial Research Chair in Software Engineering for Ultra Large Scale systems at Queen's University. Dr. Hassan spearheaded the organization and creation of the Mining Software Repositories (MSR) conference and its research community. He co-edited special issues of the IEEE Transactions on Software Engineering and the Journal of Empirical Software Engineering on the MSR topic. Early tools and techniques developed by Dr. Hassan's team are already integrated into products used by millions of users worldwide. Dr. Hassan industrial experience includes helping architect the Blackberry wireless platform at RIM, and working for IBM Research at the Almaden Research Lab and the Computer Research Lab at Nortel Networks. Dr. Hassan is the named inventor of patents at several jurisdictions around the world including the United States, Europe, India, Canada, and Japan.

Audris Mockus studies software developers' culture and behavior through the recovery, documentation, and analysis of digital remains. These digital traces reflect projections of collective and individual activity. He reconstructs the reality from these projections by designing data mining methods to summarize and augment these digital traces, interactive visualization techniques to inspect, present, and control the behavior of teams and individuals, and statistical models and optimization techniques to understand the nature of individual and collective behavior. Audris Mockus received B.S. and M.S. in Applied Mathematics from Moscow Institute of Physics and Technology in 1988. In 1991 he received M.S. and in 1994 he received Ph.D. in Statistics from Carnegie Mellon University. He works at Avaya Labs Research. Previously he worked at Software Production Research Department of Bell Labs.

Anand Sinha is an IT Manager at Mabel's Labels. Prior to joining Mabel's Labels, Anand worked at Research In Motion (RIM) for 18 years, where he pioneered many wireless enablement products that focus on seamless integration of wired applications to the wireless space. He was a member of the team that invented and implemented the BlackBerry Handheld device. He led a team that developed the BlackBerry Enterprise Server to enable wireless email and calendar synchronization between a corporate system and a BlackBerry. In addition, Anand formed many teams working in automated testing, field tool development, automatic provisioning and upgrade of BlackBerrys, statistical analysis of software failure rates, and represented the Software Development team to all major carriers worldwide as part of the BlackBerry Quality Improvement initiatives. During his time at RIM, Anand had 4 software patents submitted in the USA and Europe (2 granted, 2 pending).

Naoyasu Ubayashi is a professor at Kyushu University since 2010. He is leading the POSL (Principles of Software Languages) research group at Kyushu University. Before joining Kyushu University, he worked for Toshiba Corporation and Kyushu Institute of Technology. He received his Ph.D. from the University of Tokyo. He is a member of ACM SIGPLAN, IEEE Computer Society, and Information Processing Society of Japan (IPSJ). He received "IPSJ SIG Research Award 2003" from IPSJ.