This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

Developer Micro Interaction Metrics for Software Defect Prediction

Taek Lee, Jaechang Nam, Donggyun Han, Sunghun Kim, and Hoh Peter In

Abstract—To facilitate software quality assurance, defect prediction metrics, such as source code metrics, change churns, and the number of previous defects, have been actively studied. Despite the common understanding that developer behavioral interaction patterns can affect software quality, these widely used defect prediction metrics do not consider developer behavior. We therefore propose micro interaction metrics (MIMs), which are metrics that leverage developer interaction information. The developer interactions, such as file editing and browsing events in task sessions, are captured and stored as information by Mylyn, an Eclipse plug-in. Our experimental evaluation demonstrates that MIMs significantly improve overall defect prediction accuracy when combined with existing software measures, perform well in a cost-effective manner, and provide intuitive feedback that enables developers to recognize their own inefficient behaviors during software development.

Index Terms—Defect prediction, software quality, software metrics, developer interaction, Mylyn

1 Introduction

QUALITY assurance is a typical resource-constrained activity when the time-to-market requirements of software delivery must be met. In an embedded software market, for example, it is reported that only a four-week delay in software delivery can cause a serious 22% revenue loss if the overall software lifetime is 52 weeks [1]. Demand for the rapid release of software to the market is a critical issue for companies in most sectors of software markets. Even though it is important to meet such an urgent demand, careless quality assurance can entail technical debt [64]. The negative impact of a defective software reputation is often fatal in the market. Thus, quality assurance becomes critical immediately before the software release; however, at that stage, time and human resources are typically insufficient for eliminating every latent defect by the deadline. Developers or quality assurance managers therefore urgently require a technique that effectively predicts defects and enables the application of best efforts in resolving them.

For this reason, defect prediction has been an active research area in software engineering [2], [3], [4], [5], [6], [7], [8], [9], [10]; many effective defect prediction metrics have been proposed. In particular, source code metrics (CMs) and change history metrics (HMs) have been widely used with reasonable prediction accuracy as a de facto standard for performance benchmarks [11], [12], [61] and industry practices. For example, Chidamber and Kemerer (CK) metrics [13] and McCabe's cyclomatic complexity [14] are the most popular CMs in use. The number of revisions, authors, and past fixes, along with the age of a file, are commonly used HMs for defect prediction [15]. Microsoft built the CRANE [16] system for predicting the failure-proneness of code components in Windows Vista based on software measures. In addition, Google[1] leverages past bugfix information [62] to highlight areas of code that are creating issues. The effectiveness of CMs and HMs has been widely discussed in industry and academia.

However, despite the understanding that developer behavioral interaction can affect software quality, currently available CMs and HMs do not address developer behavior. Developers can err with an ineffective or inefficient habit in development processes; consequently, defects can be introduced. In previous studies, for example, LaToza et al. [18] surveyed the work habits of developers and found that work interruptions and frequent task switching affected software quality. In addition, Parnin et al. [19] reported that interruptions have negative effects on context recovery during programming tasks. Ko et al. [20] identified the possible behavioral causes of programming errors by using a breakdown model of human cognitive processes. These studies detected correlations between the behavior of developers and quality of software production. Accordingly, it is desirable to exploit developer interaction information when building defect prediction models.

Studying developer behavior is integral to the long-term perspective on managing sustainable software quali-

- *T. Lee and H. P. In (corresponding author) are with Korea University, Seoul, South Korea. E-mail: {comtaek, hoh_in}@korea.ac.kr*
- *J. Nam is with University of Waterloo, ON, Canada. E-mail: jc.nam@uwaterloo.ca*
- *D. Han is with University Colleage London, London, United Kingdom. E-mail: d.han.14@ucl.ac.uk*
- *S. Kim is with the Hong Kong University of Science and Technology, Hong Kong, China. E-mail: hunkim@cse.ust.hk.*

[1] Bug Prediction at Google, http://googleengtools.blogspot.com/2011/12/bug-prediction-atgoogle.html

ty. Behavior-based software metrics can enlighten developers about best practices for improving the quality of software production. Developers must understand what is taking place in the development process and receive corrective feedback if any repeated error or inefficiency exists in their work behavior during development.

To this end, we propose micro interaction metrics (MIMs) to capture the behavioral interactions of developers during development. We employ Mylyn[2], an Eclipse plug-in for task-context storage and recovery [21]. Because developer interaction patterns affect software quality and developer productivity [18], [19], [20], metrics based on developer interactions can be important indicators for predicting defects. In this paper, we compare the defect prediction performance of MIMs, CMs, and HMs using Eclipse subprojects that contain Mylyn data during the period between Dec 2005 and Jun 2010. Our evaluation results show that MIMs significantly improve overall prediction accuracy when used along with CMs and HMs (Section 5.1). Moreover, they facilitate code inspection in a cost-effective way in terms of required effort, providing significant benefits with considerable defect detection (Section 5.2). Additionally, we acquire and compare MIMs from two different domains (open source versus closed source projects) to explore any difference of MIM ranks in terms of prediction contribution (Section 6.4).

In addition to providing outstanding performance, MIMs have several other promising attributes. First, unlike CMs and HMs, MIMs can provide fine-grained information. For example, a single commit to a source code repository such as CVS records the final snapshot of the changed code no matter how many micro changes have actually occurred on source code files. This abstraction is unfavorable in terms of collecting detailed information about code changes. Thus, MIMs can create a synergy with CMs and HMs to bolster the informative power of software metrics in practice.

Second, MIMs are available early in the development process: they can be extracted as soon as task session logs are available in the integrated development environment (IDE) before code is committed to a repository. This is a major benefit, particularly in defect prediction applications. Early defect prediction can reduce the potential costs associated with late defect detection and remediation.

Finally, MIMs provide inherently intuitive feedback to developers about their interactions that are most relevant to defect occurrence. By visualizing the MIMs, developers can realize their "as is" and "to be" states of quality management activities. If MIMs can be implemented in an IDE-centric tool, the feedback can provide developers with a warning and assist them in taking corrective actions if necessary.

The full list of 24 MIMs is described in Table 1. Some MIMs are from our previous study [23]; these are marked with the dagger "†" symbol in Table 1. The unmarked 10 MIMs are newly proposed in this paper. The contributions of this study are outlined as follows:

- Propose a new complementary set of MIMs that capture developer interaction information.
- Conduct an empirical evaluation on MIMs as defect predictors by using various measures (e.g., F-measure, cost-effectiveness, and predictive power based on gain ratio).
- Present three additional case studies in industrial sectors and report how the use of MIMs in industrial projects is different from that in open source projects (Eclipse).
- Verify if main MIMs really depict well working habits of developers as they think by interview questions
- Discuss empirical findings and the implications of the evaluation results and an application of MIMs as a software development tool.

2 Related Work

Many researchers have proposed new defect prediction algorithms and/or new metrics. However, they do not explore and explain the research question of correlations between the behavioral working patterns of developers and post-release software defects. MIMs are a dedicated metrics suite that can be used to address this question.

2.1 Defect Prediction Methods

CMs such as Chidamber and Kemerer (CK) [13] are widely used for defect prediction. Basili et al. [24] applied CK metrics to eight information management systems. Ohlsson et al. [14] used several graph metrics, including McCabe's cyclomatic complexity, on a telecom system. Subramanyam et al. [25] used CK metrics on a commercial C++/Java system; Gyimothy et al. [26] performed a similar analysis on Mozilla. Nagappan and Ball [27] estimated the pre-release defect density of Windows Server 2003 using a static analysis tool. Nagappan et al. [6] used CMs to predict post-release defects at the module level in five Microsoft systems. Zimmermann et al. [28] applied several code metrics to Eclipse.

In addition, HMs have been proposed and widely used for defect prediction. Nagappan et al. [29] proposed the code churn metric, which is related to the amount of changed code. They showed that code churn is very effective for defect prediction. Moser et al. [15] used the number of revisions, authors, past fixes, and file age as defect predictors. Kim et al. [4] used previous defect information to predict future defects. Hassan [2] adopted the concept of entropy for change metrics. They determined that their approach is often better than both the code churn approach and the method based on previous bugs. D'Ambros et al. [11] conducted an extensive comparison of existing bug prediction approaches using CMs, HMs, past defects, and the entropy of change metrics. In addition, they proposed two novel metrics: the churn and entropy of source code metrics.

[2] Eclipse Mylyn, http://www.eclipse.org/mylyn

TABLE 1
LIST OF MICRO INTERACTION METRICS. COLUMN PI SHOWS THE PERFORMANCE INDEX (PI) DENOTED BY THE NORMALIZED GAIN RATIO VALUE IN SECTION 5.3.

Type	**Level**	**Metrics Name**	**PI**	**Desciption**
Editing Interaction	File	*NumEditEvent*†	0.73	Number of edit events observed for a file.
		NumEditingDevelopers	1.00	Number of developers that edited a file in the history. A single file can be edited by more than one developer.
	Task	*NumRareEdit*†	0.65	Number of edit events with low DOI attribute values (less than the median of the DOIs of all events in a task session).
		NumParallelEdit†	0.24	Number of files edited in parallel in a task session. For example, if task T has file edit events over time, such as f1, f2, f2, and f2, task T has file editions for only two distinct files; i.e., f1 and f2. Thus, *NumParallelEdit* becomes 2 for task T.
		NumRepeatedEdit†	0.27	Number of files edited more than one time during a task session. In the above example of *NumParallelEdit*, *NumRepeatedEdit* becomes 1 for task T. Only the file f2 was edited more than one time (three times).
Browsing Interaction	File	*NumSelectionEvent*†	0.5	Number of selection events observed for a file.
	Task	*NumParallelBrws*†	0.22	This definition is similar to that of *NumParallelEdit* except that it is for browsing events. Browsing events are a special case of selection events with an event time duration that is more than or equal to 1 s. The time duration of an interaction event can be computed by referencing StartDate and EndDate attributes in a task session log (=EndDate-StartDate)
		NumRareBrws†	0.47	A definition similar to that of *NumRareEdit*, except for browsing events.
		NumRepeatedBrws†	0.22	A definition similar to that of *NumRepeatedEdit*, except for browsing events.
Time Interval	File	*TimeSinceLastTask*	0.23	Time elapsed since the last task for a file. It measures how recently developers have accessed the file.
		(Avg†*|Max)TimeIntervalEditEdit* *(Avg*†*|Max)TimeIntervalBrwsBrws* *(Avg|Max)TimeIntervalBrwsEdit*	(0.24 | 0.31) (0.29 | 0.33) (0.2 | 0.34)	Average/maximum time interval between sequential interaction events (e.g., between two editing events, between two browsing events, and between a browsing event and an editing event).
	Task	*NumInterruptions*	0.16	Number of pauses in a task session. Count the cases in which the time gap between interaction events during a task session is greater than 15 min. Like Parnin et al. [19], we used the threshold of 15 mins.
Time Spent	File	*HourPerEditing* *HourPerBrowsing*	0.54 0.65	Average time spent per an editing or browsing interaction event for a file.
	Task	*TimeSpent*† *TimeSpentBeforeEdit*† *TimeSpentAfterEdit*†	0.16 0.21 0.22	*TimeSpent*: Total time in finishing a task session. *TimeSpentBeforeEdit* and *TimeSpentAfterEdit*: Spans of time before the initial edit and after the last edit during a task session.
Work Effort	Task	*NumMultiTasks*†	0.28	Number of multiple tasks assigned to the same developer during a working period of time for a given task.
		RatioCodeUnderstandingEffort	0.27	Time spent browsing files divided by the total task session time (= browsing time + editing time). We assumed a portion of the time spent in browsing files was for understanding code given in a task.

Other defect prediction methods have likewise been proposed. Khomh et al. [31] studied the impact of anti-patterns; i.e., deficient design choices against object-oriented systems. They found that the presence of anti-patterns is a powerful predictor of both defects and change rates in systems. Zimmermann and Nagappan [9] predicted defects in Windows Server 2003 using network analysis of dependency graphs among binaries. They used dependency graphs to identify defect-prone central program units. Meneely et al. [32] proposed developer social network-based metrics to capture the developer collaboration structure and predict defects using it. Bacchelli et al. [33] proposed popularity metrics based on e-mail archives. They assumed that the most discussed files were more defect-prone.

2.2 Developer Interaction History

Recently, researchers have used developer interaction histories to facilitate software development and maintenance. Kersten et al. [21], [34] proposed a task context model and implemented Mylyn to store/restore the task context when developers switch their task context. Because Mylyn is becoming increasingly popular, the data sets of developer interaction histories are being frequently captured. Murphy et al. [35] analyzed statistics relating to IDE usage by employing Mylyn data. They identified the most frequently employed user interface components and commands. By analyzing Mylyn task session data, Ying and Robillard [36] found that different types of tasks (e.g., bug-fixing or feature enhancement) are associated with different editing styles. This is useful information for software development tool designers.

Parnin et al. [19] presented an in-depth study about developer interruption times and suitable strategies for resuming an interrupted programming task. Parnin et al. showed that the time interval between developer interaction activities during a task session is usually less than one minute (i.e., developer interruption time generally exceeds one minute when an interruption occurs). In addition, they showed that developers engage in a variety of non-editing activities (e.g., navigation of recorded notes, assigned task history, and code revision history) to recover task context before making their first edit in a session.

Robbes and Lanza [37] developed a software evolution monitoring prototype to understand fine-grained development session information that is not usually recorded by current version control systems. By monitoring developers' IDE usage pattern, their approach captures semantic change of individual operations occurred between developer commits. In addition, Robbes et al. [38] take advantage of the fine-grained semantic changes and propose new logical-coupling measurements to detect logically coupled software entities by measuring how often they changed together during development. Zou et al. [39] discussed how interaction coupling could be detected in task interaction histories and their case study showed that information pertaining to interaction coupling is helpful for comprehending software maintenance activities.

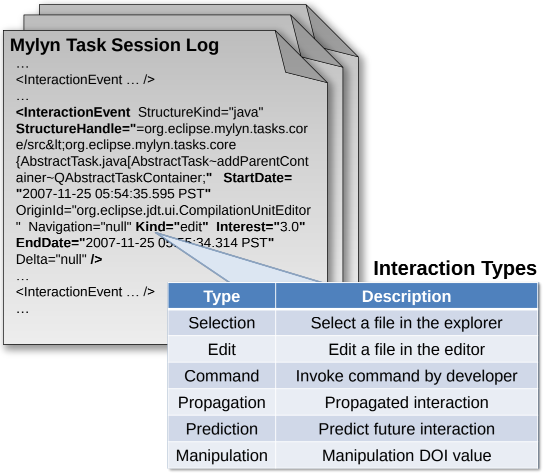
Shin et al. [10] used developer activity metrics, such as team cohesion, miscommunication, and misguided effort, to predict software vulnerabilities. Bettenburg et al. [40] investigated how information relating to the social structures and communications between developers and users could be used to predict software quality. Bettenburg et al. quantified the degree of social communication interactions among people. Shin et al. and Bettenburg et al. [10], [40] addressed defect prediction issues by using information pertaining to developer activities and communications. However, the above studies do not explore the effects of comprehensive interactions of a developer in the IDE.

3 Micro Interaction Metrics

In this section, we introduce background of Mylyn and explain our proposed MIMs in detail.

3.1 Mylyn

To extract MIMs, we used Mylyn, which records and shares the context of developer tasks and interactions. Technically, the Mylyn Monitor[3] enables collecting information about developer activities in Eclipse. Figure 1 shows Mylyn task session logs, which consist of several 'InteractionEvent' tags in XML format. The 'Kind' attribute denotes the interaction type—selection, edit, command, propagation, prediction, and manipulation—as shown in Figure 1. For example, edit events are recorded when developers edit a file. Propagation events occur when a developer uses automatic refactoring features in Eclipse.



Mylyn Task Session Log

```
…
<InteractionEvent … />
…
<InteractionEvent StructureKind="java"
StructureHandle="=org.eclipse.mylyn.tasks.cor
e/src&lt;org.eclipse.mylyn.tasks.core
{AbstractTask.java[AbstractTask~addParentCont
ainer~QAbstractTaskContainer;" StartDate=
"2007-11-25 05:54:35.595 PST"
OriginId="org.eclipse.jdt.ui.CompilationUnitEditor
" Navigation="null" Kind="edit" Interest="3.0"
EndDate="2007-11-25 05:55:34.314 PST"
Delta="null" />
…
<InteractionEvent … />
…
```

Interaction Types

Type	Description
Selection	Select a file in the explorer
Edit	Edit a file in the editor
Command	Invoke command by developer
Propagation	Propagated interaction
Prediction	Predict future interaction
Manipulation	Manipulation DOI value

Fig. 1. Mylyn log data and interaction event type

Each event is recorded with attributes such as 'StartDate,' 'EndDate,' 'StructureHandle,' and 'Interest.' The 'StartDate' and 'EndDate' attributes represent the event start and end times. The 'StructureHandle' attribute denotes the corresponding files and methods in the event. For edit events, for example, the 'StructureHandle' attribute indicates the file being edited. The 'Interest' value, or degree of interest (DOI), indicates the developer's interest in the corresponding file. The DOI value is measured by

[3] http://wiki.eclipse.org/Mylyn_Integrator_Reference

the frequency of interactions with a file element and the recency of interactions, which help developers identify relatively important files for a task. Mylyn automatically computes DOI values [21], [34]. More information about Mylyn and its data are available at the Mylyn project home page.

In this study, we considered only the direct events 'Selection' and 'Edit' that are generated when accessing files.

3.2 Design of MIMs

In this section, the design rationale for MIMs and their two design levels—file and task—which are derived from Mylyn task session data, are described.

3.2.1 Design Rationale

The design goal of MIMs is to catch developer interactions that are associated with committing errors. We considered the following hypothetical possibilities for error-prone developer interactions from existing exploratory studies: interruptions during task sessions [19], [20], frequent task switching (i.e., a short time for code understanding before starting a new or previous task context) [18], any out-focused file accessing off the mainstream of the task context [34], and repeated editing of files with many previous changes [4]. These may directly or indirectly have a negative impact on developers' productivity and/or the quality of their code.

TABLE 2
MIM DESIGN IN TERMS OF THE GOAL QUESTION METRIC

Goal: Find behavioral interactions of developers that may degrade software quality	
Goal-driven Questions	**Related MIMs**
GQ1 - How frequently do developers edit files?	*NumEditEvent* *NumParallelEdit* *NumRareEdit* *NumRepeatedEdit* *NumEditingDevelopers*
GQ2 - How frequently do developers browse files?	*NumSelectionEvent* *NumParallelBrws* *NumRareBrws* *NumRepeatedBrws*
GQ3 - When was the most recent work? (How much time has passed since the last task?)	*TimeSinceLastTask*
GQ4 - How much time do developers spend to work on files?	*HourPerEditing* *HourPerBrowsing* *TimeSpent* *TimeSpentBeforeEdit* *TimeSpentAfterEdit*
GQ5 - How many times are developers interrupted? (Or: How long are time intervals with no activity?)	*NumInterruptions* *(Avg|Max)TimeIntervalEditEdit* *(Avg|Max)TimeIntervalBrwsBrws* *(Avg|Max)TimeIntervalBrwsEdit*
GQ6 - How many tasks do developers undertake at a given time?	*NumMultiTasks*
GQ7 - How many times do developers work on rarely accessed files in a task context?	*NumRareBrws* *NumRareEdit*
GQ8 - How much time do developers take to understand an assigned task (or to recover a task context) before making their first edit, or to review what they have done after their last edit?	*TimeSpentBeforeEdit* *TimeSpentAfterEdit* *RatioCodeUnderstandingEffort*

The Goal Question Metric (GQM) [43] was internally used at a high-level to answer several interesting questions regarding MIM measures. GQM defines a measurement model with three levels: conceptual (a goal is defined to explain measurement reasons), operational (questions are asked to study and achieve the specific goal), and quantitative (metrics are associated with questions to enable the answering of each one in a measurable way). Table 2 outlines our objective, interesting questions, and associated MIMs to answer the corresponding questions; some MIMs are included in multiple question categories. In Section 5.4, these question categories (GQs) are evaluated to show which ones are relatively more effective at predicting defects.

3.2.2 File vs. Task Design Levels

Mylyn data basically comprises task-level session information; a single task can involve one or several files (see Figure 2). Therefore, we captured properties from two different dimensions when designing MIMs: file level and task level. Some MIMs are computable at the file level, while others are computable at the task level. The design level of each MIM is delineated in Table 1.

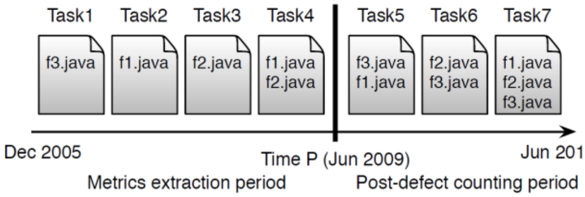


Fig. 2. Time split for extraction of metrics and post-defect counting periods

File-level MIMs capture specific interactions of a developer on a certain file in task sessions. *NumEditEvent* is a file-level MIM; it counts the number of edit events for a file in a task session. In Figure 2, for example, Task 4 has edit events for files 'f1.java' and 'f2.java.' *NumEditEvent* for the specific file 'f2.java' will be just one in terms of file level, even though Task 4 has a total of two file-edit events in a task-level unit.

By contrast, task-level MIMs capture task-scoped properties over a task session, rather than file-specific interactions. For example, *NumInterruptions* is a representative task-level MIM; it counts the number of temporal pauses between interaction events during the overall task session.

Thus, task-level MIMs characterize a global property over a task session; moreover, they affect the local file-level activities of developers within the given session. For example, using information from Figure 2, suppose that a developer worked on Task 4 in very distracting conditions. *NumInterruptions* for Task 4 would be measured with a high value. In the meantime, Task 4 has interaction events for two files, 'f1.java' and 'f2.java,' meaning that these two Task 4 files were edited in negative working conditions (i.e., with a high frequency of interruptions). Thus, the surrounding properties (workspace conditions) measured by the task-level MIMs would affect the qualities of the sub-associated activities, such as file editing and browsing. Therefore, file instances with the same task session ID would share the same properties as task-level MIMs. Conclusively, in the Task 4 example of Figure 2, the two file instances, 'f1.java' and 'f2.java,' would have the same metric value of *NumInterruptions*.

4 Experimental Setup

In this section, we describe our experimental setup for comparing the performance of MIMs with CMs and HMs for defect prediction. In addition, we present our research questions, overall experimental process, and a baseline dummy classifier for the performance comparison.

4.1 Research Questions

To evaluate MIMs, we define four research questions as shown in Table 3. RQ1 and RQ2 were studied with Eclipse project data. Then, case studies on three industrial projects were conducted to address RQ3. For RQ1, we tested whether MIMs are useful as cost-effective indicators of software quality. Then, for RQ2, we identify positive and negative developer behavioral interactions for software quality assurance. Lastly, for RQ3 and RQ4, we investigated how MIMs function in commercial projects and MIM ranks vary in 7 different project domains (i.e., open source plus commercial projects). The evaluation of each of the above questions is respectively presented in Sections 5.1 (RQ1), 5.3 (RQ2), 6.2 (RQ3), and 6.3 (RQ4).

TABLE 3
Research Questions

RQ1	Can MIM improve defect prediction performance when used with existing code metrics (i.e., CM and HM)?
RQ2	What MIMs are particularly effective contributors to defect prediction improvement?
RQ3	Are MIMs effective as defect predictors in commercial projects compared to a random predictor?
RQ4	How do MIM ranks vary in different project domains (e.g., open source plus commercial projects)?

4.2 Experimental Process

In defect prediction experiments, we used the common bug classification process [7], [8], [15], as shown in Figure 3, which helps predict whether a given unknown file instance is buggy.

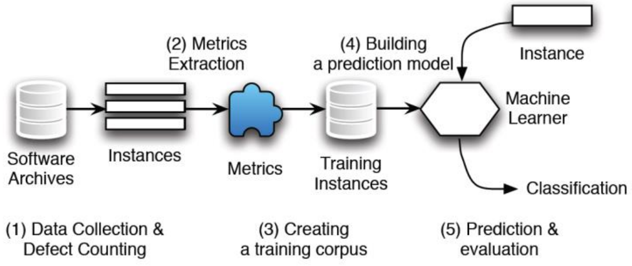

(2) Metrics Extraction
(4) Building a prediction model
Instance
Machine Learner
Software Archives
Instances
Metrics
Training Instances
Classification
(1) Data Collection & Defect Counting
(3) Creating a training corpus
(5) Prediction & evaluation

Fig. 3. Steps involved in defect prediction process

First, we collect all valid files from Mylyn task session logs as instances (in the machine learning sense) and count the number of post-defects in each file. Here, post-defects refer to the number of defects reported after a software release (e.g., time *P* in Figure 2). We label a file as 'buggy' if it contains at least one post-defect. Otherwise, we label the file as 'clean.' The post-defect counting process is detailed in Section 4.2.2. Then, we extract the MIMs, CMs, and HMs for each instance, as explained in Section 4.2.3. Finally, we train the prediction models using the machine-learning algorithms implemented in Weka [45]. The trained prediction models classify instances as 'buggy' or 'clean.'

4.2.1 Data Integrity Checking

Before data collection, we checked the data integrity of the Mylyn task session log by investigating an actual task scenario of a developer of one of the commercial projects of Table 11 and the recorded Mylyn log. We asked for permission to record video while the developer was working on the task. The video[4] could correctly explain what he did over time and when he was interrupted during the task session. Therefore, we could check if interaction events recorded in the Mylyn task session log in fact captured the developer's actual behaviors inside the video. Table 4 is the Myln task session log recorded for the video. By comparing the Mylyn log with the video, we obtained a clear understanding of the general structure of the Mylyn task session logs and recognized what was required to preprocess over the raw log data. Following is what we determined:

- The Mylyn log correctly depicted four files that were actually browsed or edited in the video. The length of the recorded Mylyn log and length of the actual work scenario of the video were consistent. The total editing and browsing times in the Mylyn log also corresponded to the ones observed in the video.

- Some events in the log had zero time duration; their StartDate and EndDate attributes were the same (e.g., ID 17 or 22 in Table 4). This type of event occurred when a file was initially opened by double-clicking or when it was skimmed without meaningful actions. Thus, these events did not actually capture a valid time duration of developer interactions. Therefore, we filtered out the events with zero time duration when extracting MIMs that quantify developer activities of actual file browsing and editing interactions in Figure 3.

- The attribute StartDate of an event was initially marked when a file (or a method) of StructureHandle gets a developer's focus on the Package Explorer window tab or on the Editing window tab in the Eclipse IDE. The attribute EndDate of an event was updated when the focus was lost from the Editing window tab by moving onto other entities of StructureHandle.

[4] http://youtu.be/gfZ0T3AjCaM

- There was a certain span of an empty time zone in which no update of StartDate or EndDate was observed. During the period of empty time, we could confirm that the developer was doing nothing in the Eclipse IDE. For example, there was no update of StartDate or EndDate in an event between 17:01:46 and 17:05:06 in Table 4; we confirmed that the developer engaged in activities of web surfing to obtain some information and was working on a database during that very time period in the video (16m:59s–20m:19s).

TABLE 4
LOG EXAMPLE OF A MYLYN TASK SESSION USED IN DEVELOPER'S VIDEO INVESTIGATION

ID	Kind	StartDate	EndDate	StructureHandle
1	selection	16:44:47	17:10:14	BuildArchApproach.java
2	selection	16:48:44	16:52:37	BuildArchApproach.java
3	edit	16:48:45	16:49:28	BuildArchApproach.java
4	selection	16:45:50	16:48:38	BuildArchApproach.java
5	edit	16:45:53	16:47:34	BuildArchApproach.java
6	selection	16:57:55	17:09:21	BuildArchApproachModel.java
7	selection	16:49:36	17:10:14	BuildArchApproach.java
8	edit	16:49:39	17:10:17	BuildArchApproach.java
9	selection	16:57:53	17:09:21	BuildArchApproachModel.java
10	selection	16:47:38	16:47:38	BuildArchApproach.java
11	edit	16:47:55	16:47:55	BuildArchApproach.java
12	selection	16:49:34	16:49:34	BuildArchApproach.java
13	selection	16:45:19	17:09:47	AKToolMain.java
14	selection	16:45:24	16:45:24	BuildArchApproach.java
15	selection	17:09:47	17:09:47	AKToolMain.java
16	selection	17:16:32	17:18:02	BuildArchApproachDB.java
17	edit	17:18:05	17:18:05	BuildArchApproachDB.java
18	selection	16:45:08	16:45:21	BuildArchApproach.java
19	edit	16:45:21	16:45:22	BuildArchApproach.java
20	selection	16:57:54	16:57:54	BuildArchApproachModel.java
21	selection	16:58:14	17:10:21	BuildArchApproachDB.java
22	selection	16:53:10	16:53:10	BuildArchApproach.java
23	selection	16:58:14	17:18:07	BuildArchApproachDB.java
24	edit	16:58:23	17:18:01	BuildArchApproachDB.java

4.2.2 Data Collection and Defect Counting (Step 1)

In our experiments, we used a total of 5,973 Mylyn task sessions from Eclipse Bugzilla between December 2005 and June 2010.[5]

We set the release time *P* of Eclipse 3.5 (Galileo at June 24, 2009) to explicitly separate the metric extraction period and post-defect counting period, as shown in Figure 2. All metrics (MIMs, CMs, and HMs) for instances were computed before *P;* post-defects were counted after *P*.

The instances had to be present in both time periods (i.e., before and after time *P*) for the collecting of metrics and counting of defects. If a file was not present during the metric collection period, no metrics were available to build prediction models. On the other hand, if a file was not present in the post-defect counting period, the defect number for the file was always zero, which was misleading. To ensure that we only used files that existed in both periods, we confirmed their existence in the Eclipse CVS repository.

We used edited file information in Mylyn tasks to count post-defects. Each task session log is directly attached to a bug report; therefore, we checked whether the resolution of the corresponding bug report indicated a fixed bug or not. If it was not fixed or not a bug (e.g., it was a feature enhancement or trivial issue), we did not count the edited files in the task as post-defects. For example, suppose that Tasks 5 and 6 were registered in the fixed bug reports, while Task 7 was registered as a feature enhancement bug report, as shown in Figure 2. In this case, the post-defect number of 'f3.java' would be two, because Task 7 was for feature enhancement.

In our experiments, we did not use conventional post-defect counting heuristics [46], [47] because they are limited in obtaining defect information. The links between bugs and committed changes (e.g., CVS) are typically automatically mined from change logs and bug reports using heuristics, such as searching for specific keywords (e.g., 'bug' or 'fix') and bug IDs in the change logs. However, the accuracy of these heuristics depends on the quality of the change logs. Bird et al. [51] found that there are many missing links because of the absence of bug references in change logs. Developers possibly do not leave change logs even if they actually fixed bugs in the code change, or they provide a wrong bug reference in the logs. The absence of bug references in change logs results in biased defect information and affects defect prediction performance.

CVS logs are manually recorded by developers, whereas Mylyn task logs are automatically recorded by the Eclipse IDE tool and linked (attached) to the corresponding bug report in the Bugzilla. Therefore, Mylyn logs are more beneficial than CVS logs.

4.2.3 Metric Extraction (Step 2)

The extraction of MIMs is straightforward. First, file-level MIMs are computed for file instances. For task sessions, only the specific interaction events that target a file instance are aggregated from the metric extraction period (Figure 2). Then, file-level MIMs for the file instance are computed with the specific event data. Second, task-level MIMs are computed and propagated to sub-activities of associated files, as explained in Section 3.2.2.

If a file has been edited many times during several tasks, the file will have multiple MIM values that have been computed from each task. Thus, the average of these values is adopted for the file instance. As shown in Figure 2, for example, 'f1.java' takes two different values computed from Tasks 2 and 4 so that the two values are averaged and handled as the final MIM value of the file. However, in the case of *MaxTimeInverval** metrics, the maximum of the multiple values is handled as a final MIM value.

To evaluate performance, CMs and HMs were additionally extracted during the metric extraction period (Figure 2). Because CMs are snapshot metrics, they are extracted at time *P*. The Understand tool[6] was used to

[5] Mylyn was released in December 2005 and has been widely used since then.

[6] Understand 2.0, http://www.scitools.com/products/understand/

extract the CMs. The tool extracts 24 file-level and 18 class-level metrics, such as CK [13] and object-oriented metrics. If a file has more than one class, the file-level metrics are derived from multiple class-level metrics. The Understand tool provides two types of metrics: 'Avg*' and 'Count*.' 'Avg*' class-level metrics are averaged to generate file-level metrics from multiple classes in a file. However, the values are summed together when the file-level metrics are extracted from 'Count*' class-level metrics. All 42 of the CMs used in our experiments are listed in Table 5.

TABLE 5
LIST OF SOURCE CODE METRICS (CMS)

Metrics	Description
AvgCyclomatic	Average cyclomatic complexity
MaxCyclomaticStrict	Maximum strict cyclomatic complexity
CountLine	Number of all lines
CountLineBlank	Number of blank lines
RatioCommentToCode	Ratio of comment lines to code lines
MaxCyclomaticModified	Maximum modified cyclomatic complexity
AvgCyclomaticModified	Average modified cyclomatic complexity
AvgEssential	Average Essential complexity
CountDeclFunction	Number of functions
CountStmtExe	Number of executable statements
CountStmt	Number of statements
CountLineCodeDecl	Number of lines containing declarative code
CountSemicolon	Number of semicolons
CountLineCode	Lines of code
AvgCyclomaticStrict	Average strict cyclomatic complexity
CountLineCodeExe	Number of lines containing executable code
MaxCyclomatic	Maximum cyclomatic complexity
CountLineComment	Number of lines containing comment
CountDeclClass	Number of classes
CountStmtDecl	Number of declarative statements
SumCyclomaticStrict	Sum of strict cyclomatic complexity
SumCyclomatic	Sum of cyclomatic complexity (WMC)
SumCyclomaticModified	Sum of modified cyclomatic complexity
SumEssential	Sum of essential complexity of methods
AvgLine	Average number of lines
AvgLineBlank	Average number of blank
AvgLineCode	Average number of code lines
AvgLineComment	Average number of comment lines
PercentLackOfCohesion	Lack of cohesion in methods (LCOM)
CountClassBase	Number of immediate base classes
CountClassCoupled	Coupling between object classes (CBO)
CountClassDerived	Number of child classes (NOC)
CountDeclClassVariable	Number of class variables (NIV)
CountDeclInstanceMethod	Number of instance methods (NIM)
CountDeclInstanceVariable	Number of instance variables
CountDeclMethod	Number of local methods (NOM)
CountDeclMethodAll	Number of local methods (RFC)
CountDeclMethodDefault	Number of local default methods
CountDeclMethodPrivate	Number of local private methods (NPM)
CountDeclMethodProtected	Number of local protected methods
CountDeclMethodPublic	Number of local public methods (NOPM)
MaxInheritanceTree	Maximum depth of Inheritance Tree (DIT)

In addition, 15 HMs were collected by using the approach of Moser et al. [15]. All HMs were collected from the change history stored in the Eclipse CVS repository (http://archive.eclipse.org/arch/). They are listed in Table 6.

The *Refactorings* metric, which is an indicator of whether a file change involved refactoring [15], was obtained by mining CVS commit logs. We counted the number of refactored revisions of a file by searching the keyword 'refactor' in commit logs [15]. The *Age* metric indicates the period when a file existed [15]. The *BugFixes* metric represents the number of fixed bugs. To count *BugFixes*, we used a search for explicit Bugzilla bug IDs in the commit logs.

Fixed bugs (and not feature enhancements) were marked as bug-fix changes [15]. Specific keywords, such as 'bug' or 'fix,' were searched ('postfix' and 'prefix' were excluded [15]).

TABLE 6
LIST OF HISTORY METRICS (HMS)

Metrics	Description
Revisions	# of file revisions
Refactorings	# of times a file was refactored
BugFixes	# of times a file was involved in bug fixes
Authors	# of distinct authors committing a file
LOC_Added	Sum of the lines of code added to a file
Max_LOC_Added	Maximum number of lines of code added
Avg_LOC_Added	Average number of lines of code added
LOC_Deleted	Sum of the lines of code deleted in a file
Max_LOC_Deleted	Maximum number of lines of code deleted
Avg_LOC_Deleted	Average number of lines of code deleted
CodeChurn	Sum of (added LOC - deleted LOC)
Max_CodeChurn	Maximum CodeChurn for all revisions
Avg_CodeChurn	Average CodeChurn per revision
Age	Age of a file in weeks
Weighted_Age	Age considering LOC_Added

4.2.4 Creating a Training Corpus (Step 3)

To evaluate MIMs under the different subjects listed in Table 7, a training corpus of Eclipse subproject groups was constructed for the *Metrics extraction period* in Figure 2. Actually, the selected projects (subjects) consist of a larger volume of source code files than the number of instances listed in Table 7. Nevertheless, for experimental purposes, we used only the file instances that simultaneously existed in both Mylyn task session logs and CVS change logs.

We divided the subjects as follows: *Mylyn*, *Team*, *JDT/Core*, *Etc.*, and *All*. The Mylyn subject comprised the collected files belonging to the package names *org.eclipse.mylyn* or *org.eclipse.mylar*[7]; i.e., the Eclipse subproject Mylyn. The *Team* subject included files that belonged to the package name *org.eclipse.team*; i.e., the Eclipse platform subproject *Team.* Likewise, the instance files belonging to the package names *org.eclipse.jdt* and *org.eclipse.core* were grouped to the *JDT/Core* subject. All remaining package names were grouped as the *Etc.* sub-

[7] Mylar is the former name of Mylyn.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

ject, which consisted of the Eclipse plug-in development environment (26%), Eclipse platform user interface (13.5%), JFace (10.3%), Eclipse platform compare (7.4%), Equinox (6.8%), and others (36%). Lastly, the *All* subject included all files of *Mylyn*, *Team*, *JDT/Core*, and *Etc*.

TABLE 7
FILE INSTANCES AND POST-DEFECTS COLLECTED BEFORE AND AFTER P, RESPECTIVELY, IN FIGURE 2

Subjects	# of instances (files)	% of defects	# of involved developers	% of change history coverage
Mylyn	1084	16.9%	37	73%
Team	364	40.1%	3	1%
JDT/Core	244	12.7%	12	9%
Etc.	1385	11.2%	39	19%
All	3077	16.7%	91	20%

The term "coverage" in Table 7 represents the percentage that the instances cover the change history in the CVS logs. For example, 20% coverage of the *All* subjects indicates that the 3,077 file instances covered 20% of all the change files observed in the CVS logs during the data collection period.

4.2.5 Building a Prediction Model (Step 4)

A process of feature selection [49] is required to select effective features (i.e., metrics) for use in model construction. The metrics extracted in Section 4.2.3 were used as features to build a prediction model. We used the correlation-based feature subset (CFS) [59] for feature selection. Using CFS is to resolve the multicollinearity problem between correlated features [49]. CFS is an algorithm that is used to search for a more greatly reduced size of a subset of features without irrelevancy and redundancy between features in the classification problems [60]. For our purpose, CFS can be used to select an appropriate size of an effective subset of metrics, thereby avoiding the model construction overfitting problem. CFS evaluates the worth of a subset of metrics by considering the individual predictive ability of each metric along with the degree of redundancy between them. Subsets of metrics that are highly correlated with the buggy class while having a low inter-correlation are preferred.

In CFS-based feature selection, the ten-fold cross validation process was used. The training corpus data was split into ten folds; CFS selected the best features (metrics) in a fold. This selection process was iterated for each of the ten folds. Finally, only the metrics that were nominated at least more than twice (in two different folds) were finally adopted in the model construction.

Next, a classification algorithm was required to build the prediction model for the created corpus. The random forest algorithm implementation in Weka [47] was primarily used in our experiments because its performance was good, as noted in Section 5.1.3. Random forest [42] is a meta-algorithm consisting of many decision trees that outputs the class that is the mode of the classes output by individual trees. There have been several other studies using this algorithm for bug prediction on account of its good performance [12], [56].

In addition, for the performance comparison experiment outlined in Section 5.1.3, prediction models using other machine learning algorithms were built, such as naive Bayes, logistic regression, decision tree, and random forest.

4.2.6 Prediction and Evaluation (Step 5)

To evaluate the accuracy of our prediction models, F-measure was used. A composite measure of precision and recall, F-measure is widely used in data mining [49], [50]. We used the following outcomes to define precision, recall, and F-measure: predicting a buggy instance as buggy (b→b); predicting a buggy instance as clean (b→c); and predicting a clean instance as buggy (c→b).

- **Precision**: The number of instances correctly classified as buggy ($N_{b\to b}$) divided by the total number of all instances classified as buggy.

$$Precision\ P(b) = \frac{N_{b\to b}}{N_{b\to b} + N_{c\to b}} \quad (1)$$

- **Recall**: The number of instances correctly classified as buggy ($N_{b\to b}$) divided by the total number of real buggy instances.

$$Recall\ R(b) = \frac{N_{b\to b}}{N_{b\to b} + N_{b\to c}} \quad (2)$$

- **F-measure**: A harmonic mean of precision $P(b)$ and recall $R(b)$ for buggy instances.

$$F - measure\ F(b) = \frac{2 \times P(b) \times R(b)}{P(b) + R(b)} \quad (3)$$

As a model validation technique, we used ten-fold cross validation, which has been widely used in previous studies [3], [15], [32], [49] to avoid overfitting. An F-measure value obtained from ten-fold cross validation varies because ten folds are randomly partitioned. Therefore, ten-fold cross validation was repeated 10 times for each model to avoid any sampling bias [60], [65], [66] by randomizing order of the dataset before each cross validation.

For a hypothesis test, particularly concerning RQ1 (Table 3), we used the Wilcoxon rank-sum test (also known as the Mann–Whitney U test) [44] instead of t-test because F-measure outcomes from cross validation did not follow a Normal distribution as shown Figure 4. The Wilcoxon rank-sum test is a non-parametric statistical hypothesis test that assesses whether one of two samples in independent observations has higher values. If the p-value is smaller than 0.05 (at a 95% confidence level), the null hypothesis H_0 is rejected and the alternative hypothesis H_a is accepted. For RQ1, the null hypothesis H_0 is "F-measure has no statistical difference between MIM+CM+HM and CM+HM for the experiment conditions." The alternative hypothesis H_a is "F-measure of MIM+CM+HM is higher than that of CM+HM in the experiment conditions."

In addition, the prediction models were evaluated in terms of cost effectiveness (effort). Different defect prediction models may have variable quality assurance costs when they are adopted. Defect prediction models were

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

used to prioritize files from the highest likelihood to the lowest likelihood in terms of defect proneness; then, the prioritized files were inspected in turn. Therefore, the cost of code inspection to find defects was reduced by as much as the prediction model was accurate. The details are presented in Section 5.2.

4.3 Dummy Classifier

To simplify the performance comparison, a baseline was introduced: the so-called dummy classifier, in which a change file is randomly guessed as buggy or clean. Because there are only two labels of changes, buggy and clean, the dummy predictor can also achieve a certain prediction accuracy. For example, if 16.7% of changes in a project are buggy, by predicting all changes as buggy, the buggy recall would be 1, and the precision would be 0.167. In addition, the dummy predictor can randomly predict that a change is buggy or clean with a 0.5 probability. In this case, the buggy recall would be 0.5; however, the precision would still be 0.167. In our experiment, the F-measure of the dummy predictor was used as a baseline when showing the classification results; i.e., assuming that the dummy predictor randomly predicts 50% as buggy and 50% as clean. For example, for a project with 16.7% buggy changes, as shown in Table 7, the dummy buggy F-measure would be 0.25 $(= 2 \times \frac{0.5\times0.167}{0.5+0.167})$.

5 Performance Evaluation Results

In this section, we present evaluation results primarily for the research questions RQ1 and RQ2 shown in Table 3.

5.1 Performance Improvement with MIM

This section evaluates the contribution of MIMs in terms of performance improvement under several experimental conditions such as different subjects, different model training periods, and different machine learners.

5.1.1 Different Subjects

To evaluate MIM prediction performance, we compared metric sets, as shown in Figure 4. Defect prediction models were built for the five different subjects, as shown in Table 7. Only the metrics selected by CFS (Section 4.2.5) were used in the model building.

TABLE 8
METRICS SELECTED BY CFS IN EACH METRICS SUITE FOR ALL SUBJECT

Metrics Suite	Selected Metrics
MIM+CM+HM	MIM-HourPerBrowsing, MIM-NumEditEvent, MIM-NumEditingDevelopers, MIM-NumRareEdit, CM-CountLineComment, CM-AvgLineCode
CM+HM	CM-CountStmtExe, CM-CountDeclMethodAll, HM-#ofBugFixing, HM-#OfRevision, HM-Age, HM-WeightedAge
MIM	HourPerBrowsing, NumEditEvent, NumEditingDevelopers, NumRareEdit
HM	#ofBugFixing, #OfRevision, Age, WeightedAge
CM	MaxCyclomaticStrict, CountLine, RatioCommentToCode, AvgCyclomaticModified, CountDeclFunction, CountStmtExe, CountLineComment, SumCyclomaticStrict, SumEssential, AvgLine, AvgLineCode, CountDeclInstanceMethod, CountDeclMethodAll

Table 8 shows the selected metrics for each metrics suite in the *All* subject. The model training period covered December 2005 to June 2009. The trained models predicted post-defects reported during the future period of one year after the Eclipse release time *P* (June 2009 in Figure 2).

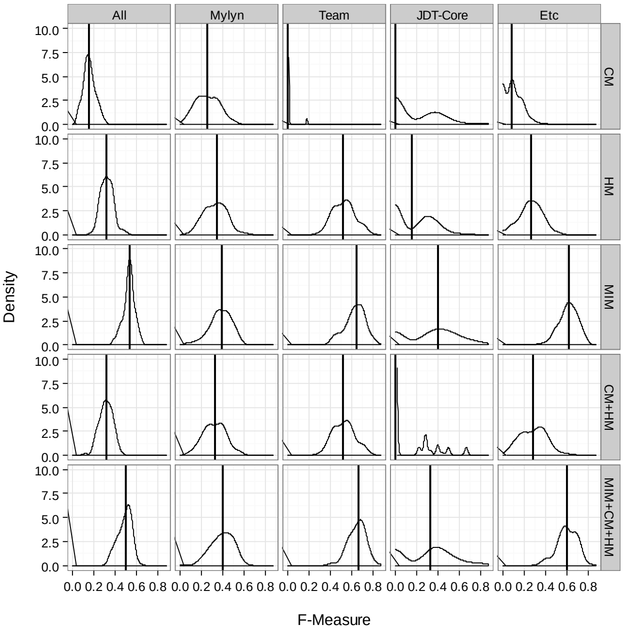


Fig. 4. Performance comparison of prediction models by different subjects. Vertical lines are each median of F-measure distributions.

TABLE 9
F-MEASURE MEDIANS AND EFFECT SIZES FOR EACH METRIC SET IN DIFFERENT SUBJECTS. FIGURES WITHOUT PARENTHESES ARE MEDIANS, AND THOSE WITH PARENTHESES ARE EFFECT SIZES COMPARATIVE TO MIM+CM+HM

Subject	MIM+CM +HM	CM+HM	MIM	HM	CM	Dummy
All	0.494	0.313 (-0.941)	0.53 (+0.356)	0.315 (-0.927)	0.154 (-1)	0.25
Mylyn	0.4	0.321 (-0.376)	0.385 (-0.073)	0.338 (-0.353)	0.25 (-0.629)	0.25
Team	0.666	0.52 (-0.646)	0.642 (-0.103)	0.52 (-0.646)	0 (-1)	0.44
JDT-Core	0.333	0 (-0.454)	0.366 (+0.099)	0.076 (-0.321)	0 (-0.267)	0.20
Etc	0.6	0.285 (-0.952)	0.62 (+0.072)	0.263 (-0.982)	0.086 (-0.999)	0.18

Figure 4 shows the performance of each set of metrics in different subjects in terms of F-measure distributions from 10 times ten-fold cross validations. We gained 100 F-measures from the repeated cross validation process and drew density plots (histograms) with the 100 F-measures.

Although the F-measure values varied, there was a clear trend in which MIM+CM+HM outperformed CM+HM for every subject. In other words, adding MIM to CM+HM improved the prediction performance of the existing CM and HM. In addition, there was an overall trend in which the MIM performed better than the CM and HM for all the subjects, as shown in Figure 4.

Table 9 lists the median of the F-measure distributions for each metric set and the effect size compared to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

case of MIM+CM+HM. The effect size is Cliff's *delta* [17], which is computed using the formula *delta* = (2*W*/*mn*)-1, where *W* is the *W* statistic of the Wilcoxon rank-sum test, and *m* and *n* are the sizes of two compared distributions. The magnitude of effect size is usually assessed using the thresholds provided in the study by Romano et al. [63]. That is, |*delta*|<0.147 "negligible", |*delta*|<0.33 "small", |*delta*|<0.474 "medium", and otherwise "large". For example, at the first row (*All* subject) in Table 9, the effect size between MIM+CM+HM and CM+HM is -0.941 whose sign is minus because median of MIM+CM+HM is greater than median of CM+HM and magnitude of the effect size is regarded as "large". In Table 9, the gray shaded cells indicate that the Wilcoxon rank-sum test rejected the null hypothesis of RQ1 (p-value < 0.05). Conclusively, it was statistically confirmed that combining MIMs with CMs and HMs can improve the overall defect prediction performance.

5.1.2 Different Model Training Periods

Because MIMs are a type of process metrics like HMs, their performance can be influenced by different periods of model training. In this section, we therefore describe prediction models built by using MIM, CM, and HM metrics extracted from three time periods: one year (June 2008 – June 2009), two years (June 2007 – June 2009), and over three years (December 2005 – June 2009). The aim of this experiment was to consider the performance sensitivity of MIMs and HMs for incremental periods of metric collection and model training. In contrast, CM was independent of these model training periods because it is a snapshot metric of code structure at the time *P* (Figure 2).

A new version of Eclipse is annually released in June. Thus, the 'one year' scenario assumes that the prediction model is trained with historical data of one period of the Eclipse release (the past year for training and the next year for prediction). The 'two years' scenario assumes that the prediction model is trained with historical data of two Eclipse releases (the past two years for training and the next year for prediction). Likewise, the 'over three years' scenario is for the case of model training with historical data of more than three Eclipse releases (more than the past three years for training and the next year for prediction).

In this experiment, the percentage of defects was a static 16.7% for the three experiments as the *All* subject in Table 7 was used. Metrics selected by CFS (Table 8) were used for model construction in each period of model training.

Figure 5 shows the performance of each set of metrics in different training periods in terms of the F-measure distributions from 10 times ten-fold cross validations. We obtained 100 F-measures from the repeated cross validation process and drew density plots (histograms) with the 100 F-measures.

As shown in Figure 5 and Table 10, MIM could consistently improve the performance of CM+HM no matter what period of model training was applied. The improvements were still statistically significant; the gray shaded cells in Table 10 indicate cases with p-values lower than 0.05. The MIM and HM performances were sensitive to differences in the model training periods, while CM was not. Incremental periods of model training tended to improve the overall performance of the MIM and HM process metrics.

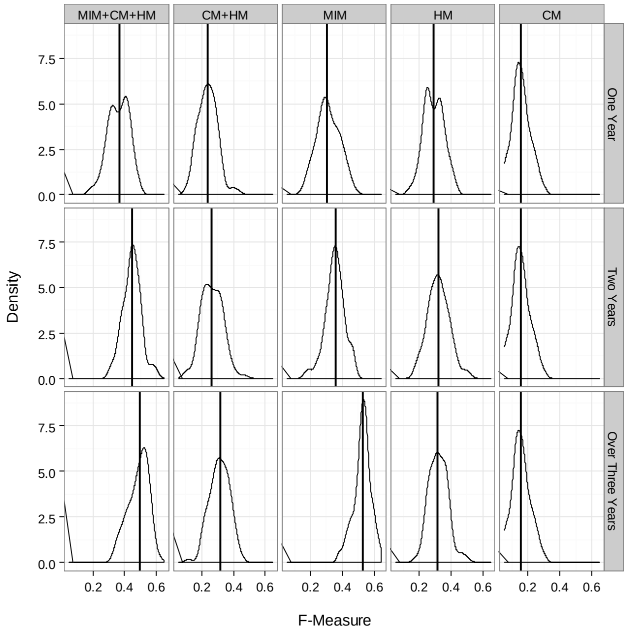


Fig. 5. Performance comparison of prediction models by different training periods. Horizontal lines are each median of F-measure distributions.

TABLE 10
F-MEASURE MEDIAN VALUES FOR EACH METRIC SET IN DIFFERENT PERIODS OF MODEL TRAINING. FIGURES WITHOUT PARENTHESES ARE MEDIANS, AND THOSE WITH PARENTHESES ARE EFFECT SIZES COMPARATIVE TO MIM+CM+HM

Period	MIM+CM +HM	CM +HM	MIM	HM	CM	Dummy
One Year	0.368	0.236 (-0.83)	0.305 (-0.398)	0.291 (-0.556)	0.154 (-0.978)	0.25
Two Years	0.445	0.263 (-0.955)	0.357 (-0.736)	0.319 (-0.834)	0.154 (-1)	0.25
Over Three Years	0.494	0.313 (-0.941)	0.53 (0.356)	0.315 (-0.927)	0.154 (-1)	0.25

MIM performance was quite sensitive because a short period has a relatively smaller number of developer interaction events in Mylyn task sessions than a longer period. The informative power from MIMs is not sufficient if MIMs are drawn from small amounts of developer interaction logs. However, HM was relatively less sensitive than MIM. In our analysis, the representative HMs selected by CFS (Table 8) showed few changed portions in both buggy and clean instances as the time period was extended.

5.1.3 Different Machine Learners

Choosing a different machine learner can produce different performance results. In this section, we therefore compare the results of prediction models using four different classification algorithms widely adopted in defect prediction studies [45], including Decision Tree, Logistic Regression, Naïve Bayesian, and Random Forest. The *All* instances (Table 7) for the *metric extraction period* (Figure

2) and metrics selected by CFS (Table 8) were used in this model construction experiment.

Figure 6 shows the performance of each set of metrics in different machine learners in terms of the F-measure distributions from 10 times ten-fold cross validations. We obtained 100 F-measures from the repeated cross validation process and drew density plots (histograms) with the 100 F-measures.

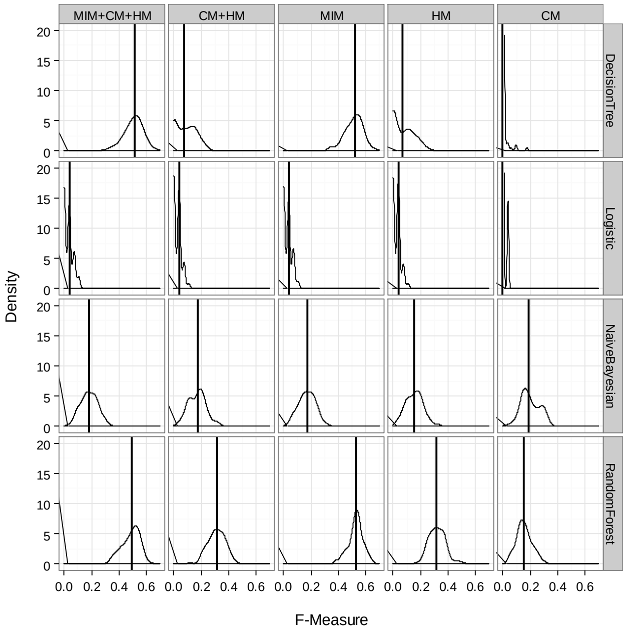


Fig. 6. Performance comparison of prediction models by different machine learners. Horizontal blue line is performance baseline of the Dummy classifier.

TABLE 11

F-MEASURE MEDIAN VALUES FOR EACH METRIC SET IN DIFFERENT MACHINE LEARNERS. FIGURES WITHOUT PARENTHESES ARE MEDIANS, AND THOSE WITH PARENTHESES ARE EFFECT SIZES COMPARATIVE TO MIM+CM+HM

Learner	MIM +CM +HM	CM +HM	MIM	HM	CM	Dummy
DecisionTree	0.511	0.073 (-1)	0.52 (0.069)	0.065 (-1)	0 (-1)	0.25
Logistic	0.036	0.037 (-0.022)	0.036 (-0.0007)	0.036 (-0.033)	0 (-0.4)	0.25
NaiveBayesian	0.179	0.176 (-0.116)	0.171 (-0.081)	0.156 (-0.236)	0.185 (0.129)	0.25
RandomForest	0.494	0.313 (-0.941)	0.53 (0.356)	0.315 (-0.927)	0.154 (-1)	0.25

As shown in Figure 6, the F-measure distributions from different machine learners varied; however, they showed the trend of MIM's better performance over the others, which could improve the combination of CM+HM (except Logistic Regression). The median values between MIM+CM+HM and CM+HM in Decision Tree and Random Forest were statistically different by the Wilcoxon rank-sum test (the gray shaded cells in Table 11).

Among the median values, Random Forest was the best choice in model construction over all the metric sets; therefore, we adopted it in the other experiments mentioned in Section 4.2.5. In contrast, Logistic Regression was the worst choice for model construction in our experiment. F-measure performance of distributions from Logistic Regression skewed a lot to zero. Interestingly, Naïve Bayesian was not the best choice in model construction but the best one for CM.

5.2 Cost Effectiveness of MIM Application

In this section, we describe our test for determining if adopting MIM is practical and cost-effective in terms of a code inspection process by simulating and comparing costs and benefits of different defect prediction models.

The cost of using a defect prediction model is critical. In practice, a defect prediction model is intended to reduce the cost of efforts spent inspecting the code space to find defects. A high performance prediction model should guarantee the detection of most defects with a low cost. Thus, a cost-benefit analysis for using a prediction model is an emerging concern in recent defect prediction studies [30], [52], [53], [54], [55], [56].

To quantify cost, we simply used lines of code (LOC) for files because Arisholm et al. [55] found that the cost of quality assurance activities in a software module tended to be proportional to the size of the module.

To quantify benefits, we counted the number of total defects found by code inspection utilizing a defect prediction model. Accordingly, with the lower cost but higher benefit, improved performance is expected in the cost effectiveness evaluation.

5.2.1 File Prioritization in Defect Proneness

To detect defects as early as possible, the inspection candidates of 3,077 files (the *All* subject in Table 7) had to be prioritized in order of their defect proneness. The file instances were prioritized from the highest to lowest probability in terms of defect proneness using defect prediction models. The prediction models were built for the *All* subject (Table 7); the CFS and random forest algorithms were used for feature selection and model construction, respectively, as explained in Section 4.2.5. To predict the defect proneness probability of an instance, the models were trained with the remaining 3,076 instances, except the instance to predict. Thus, model training and prediction were iterated for obtaining the defect proneness probabilities of all file instances. Then, the instances were sorted by the probabilities. After the prioritization process, a simulation was performed of the top ranked file (the highest probability of defect proneness) being inspected first through its code space. The next ranked files continued to be inspected in turn until all given files were exhaustively inspected and all hidden defects were finally found.

5.2.2 Cost Benefit Analysis

Figure 7 shows the defects found during the code inspection process with prediction models. The horizontal axis represents the percentage of cumulative LOCs inspected through all 3,077 files; the vertical axis represents the percentage of defects cumulatively found by the code inspection process supported with defect prediction models. The optimal model [22] is the other baseline introduced in this section, which orders files according to their defect density (i.e., the number of defects). Defect prediction models should be as close as possible to this optimal model.

As shown in the cost-benefit simulation results of Figure 7, MIM significantly reduced code inspection efforts in identifying hidden defects. For example, the dummy could be regarded as a typical strategy attempted by a code inspector who did not consider using any effective defect prediction model in the inspection process. As shown by the results, the code inspector could identify only approximately 25% of defects using 25% of inspection effort by employing the dummy strategy. However, in contrast, the inspector could find approximately 63% of defects with the same 25% of effort in the case in which MIM was used. Therefore, using a prediction model enables a code inspector to reduce the cost of inspection effort and find hidden defects as early as possible. Using HM or CM was still beneficial; however, it was not as advantageous as using MIM. In comparison, Optimal could find 100% defects with just 21% inspection effort. MIM, HM, and CM could find 59%, 35%, and 21% defects respectively with the same inspection effort.

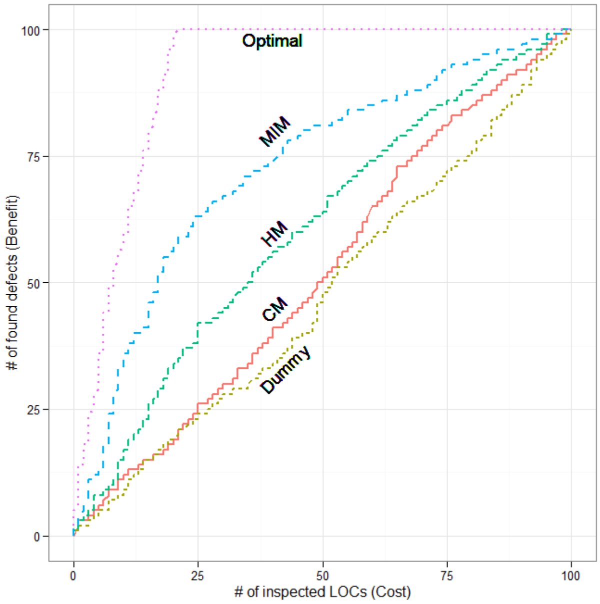


Fig. 7. Cost-benefit simulation result of applying defect prediction models in the code inspection process

5.3 Predictive Power of Individual Metrics

Each metric contributes to defect prediction performance to its own extent. In this section, we present a comparison of the entire suite of 81 metrics, which includes 24 MIMs, 42 CMs, and 15 HMs, as mentioned in Section 4.2.3.

To evaluate the predictive power of each metric, we measured the gain ratio [49] of MIMs, CMs, and HMs by applying ten-fold cross validation in Weka, and ranked them according to the normalized gain ratio values (i.e., by scaling the maximum ratio value to one). The gain ratio indicates how well a metric discriminates instance as buggy or clean. Usually, the effectiveness of the metrics can be variously evaluated depending on the machine learning algorithm used; however, metrics with a high gain ratio are generally considered important [57], [58].

The *All* subject in Table 7 was used in this analysis. Figure 8 presents the top (most effective) 30 out of the 81 metrics in order of the gain ratio values. Many MIMs were ranked higher than HMs and CMs.

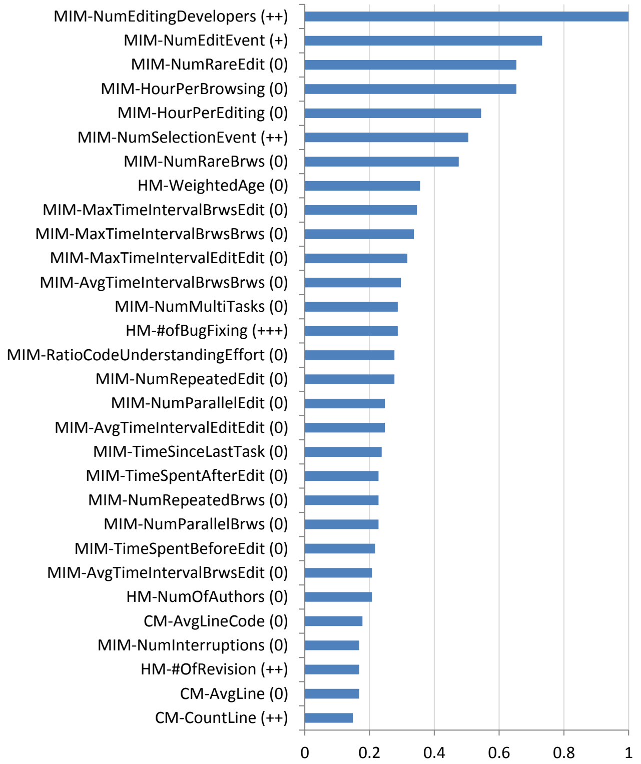


Fig. 8. Predictive power of top ranked metrics. The metrics were prioritized by the normalized gain ratio values.

The best predictor in the MIM category was *NumEditingDevelopers* (the number of developers working on a single file). For code quality prediction, this metric was as a good indicator for determining if files were assigned to and changed by many developers in the history.

In addition, the *NumEditEvent, NumRareEdit, HourPerBrowsing,* and *HourPerEditing* metrics were other good indicators for predicting code quality. It is likely that the quality of code declines according to how long and frequently the developers change the code. Moreover, it is possible that developers can make mistakes (propagate bugs) when working on rarely accessed files (*NumRareEdit*).

Of the HM category, the most effective metrics were *WeightedAge* followed by *#ofBugFixes*. Certainly, temporal information and bug fixing records were good indicators for forecasting post-defects, as confirmed in [4][15]. *NumOfAuthor* (HM) was another good defect predictor; its rationale is similar to that of *NumEditingDevelopers* (MIM). *NumEditingDevelopers* measures the number of file editors observed in a task session, while *NumOfAuthor* measures the number of file editors observed during the whole file lifespan. A task session corresponds to the time gap between opening a bug report and resolving the bug report. In our opinion, it is a good clue for defect prediction to see how many code editors involved in resolving

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

an issued bug.

The CM category contained the remaining top-ranked metrics after the above-mentioned top MIM and HM predictors outlined in Figure 8.

In Figure 8, the signs in parentheses next to the metric names show the direction of the metric's impact on the number of post-defects. For defective sample files (i.e., >= # of post-defects), we computed the Pearson correlation coefficient between the metric value and the number of post-defects. We denoted the signs with the following rules: +++: coefficient >= 0.4; ++: 0.3 <= coefficient < 0.4; +: 0.2 <= coefficient < 0.3; and 0: -0.2 < coefficient < 0.2. There was a negative impact (e.g., TimeSinceLastTask = -0.13), but it was too small to determine the direction of the impact. Note that a high gain ratio does not always guarantee a high correlation because of some non-linear relationships between the metrics and defects.

5.4 Predictive Power of Question Categories

Table 2 presents the goal-driven questions (GQs) that we organized. To determine how relevant each question category is for identifying post-defects, we evaluated the defect prediction models built with each MIM question category. The more relevant the question, the higher the relative performance of the prediction model built with the MIM question category. A performance comparison of defect prediction models built with different MIM question categories should help users understand the effectiveness of each MIM category and properly select an interesting category for the specific implementation and application.

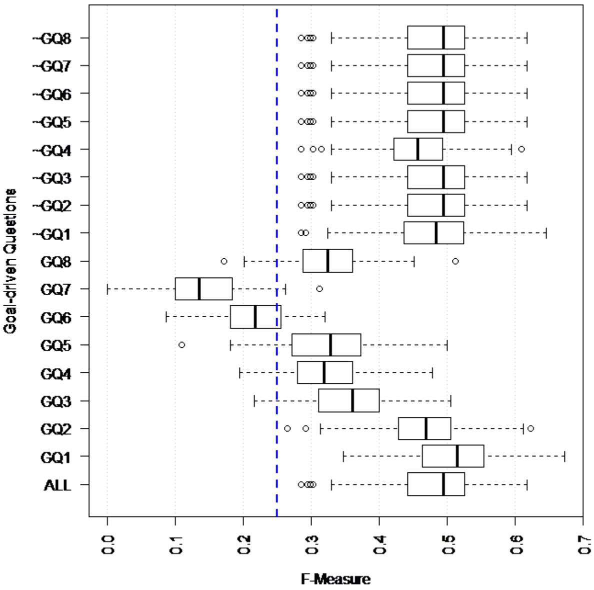


Fig. 9. Performance comparison of the question categories in Table 2

Figure 9 is the performance comparison result. Each boxplot is drawn with 100 F-measures from 10 times ten-fold cross validation. The vertical dashed blue line is the baseline performance of the Dummy classifier (i.e., F-measure 0.25, the *All* subject in Table 7).

Figure 9 includes eight question categories, GQ1 to GQ8, from Table 2. It additionally includes eight question categories denoted with the prefix "~", which means that only the designated MIM question category is excluded from the prediction model construction. For example, ~GQ1 means that the category does not include the MIMs relating to GQ1. These eight additional categories with the "~" prefix are intended to help identify by how much performance is reduced if a particular MIM question category is omitted from the model construction. Lastly, we included the *ALL* reference category, which is comprised of all 24 MIMs listed in Table 1. Note that the CFS feature selection algorithm was applied to ~GQ1 through ~GQ8 and to *ALL* in order to relieve the multicollinearity problem.

As shown in Figure 9, most MIM question categories clearly demonstrated effective predictive power as defect indicators, except for GQ6 and GQ7, whose F-measure medians were lower than the 0.25 F-measure of the dummy classifier.

The highest performance categories were GQ1 (frequent editing activities) and GQ2 (frequent browsing activities), followed by GQ3 (time elapsed since the last task). The performances of GQ1 and GQ2 were almost similar to the performance of ALL, which means GQ1 and GQ2 are predominant from the contribution point of view. In most of the omission cases, the model performances were not degraded but for GQ4 (time spent in working on files). The omission of GQ4 made the performance slightly decreased. Lastly, we could determine that the overall model performance did not depend on some superstars of MIMs, but the different question categories of MIMs were complementary in terms of their performance contributions.

Medians of the GQ1 and GQ2 F-measure distributions were 0.51 and 0.46 respectively, which thereby showed an approximately 200% higher performance than the F-measure (0.25) of the dummy classifier.

6 Commercial Domain Case Study

Developers may show different behavioral interactions depending on the project domain; e.g., open source or commercial. These differing circumstances can yield differing work motivations, time, physical space, and so on for developers. Consequently, these differences can alter developers' working habits or patterns. In this section, we describe a case study of three commercial projects for exploring RQ3 and RQ4 listed in Table 3.

6.1 Data Collection and MIM Extraction

To collect Mylyn data and extract MIMs from it, we studied three more Java projects. Table 12 briefly summarizes the three projects relating to development domains, coding periods, number of involved developers, and the number of file instances that we studied. Actually, the projects consist of a larger volume of source code files than the number of instances listed in Table 12. Nevertheless, for experimental purposes, we used only the files that are available and permitted. We obtained permission from the project directors and thereby acquired Mylyn log data of involved developers (however, we could not access source code on account of intellectual property constraints).

In Table 12, the numbers in parentheses show the developers that were actually involved; the numbers without parentheses show the developers analyzed in our study. We only examined the developers who were cooperative in providing Mylyn data.

TABLE 12
SUMMARY OF STUDIED COMMERCIAL PROJECTS

Project	Domain	Period	# of involved developers	# of instances (files)
P1	Web and internal information systems for a newspaper publishing company	8 months	9 (15)	553
P2	Software architecture decision making tool	2 months	1 (1)	62
P3	Smart TV and Android phone applications	3 months	1 (2)	308

We collected Mylyn data and extracted MIMs from the developers during the project periods as explained in Section 4. However, we could not count the number of post-defects, as was done in the study in Section 4.2.2, because the analyzed versions of projects (Table 12) were at a pre-market release stage; therefore, no information of post-delivered field defects existed. Instead, we used information from internal reports of bugs that developers found and reported from their quality assurance activities (e.g., code inspection and peer review processes).

For MIM performance evaluation, we split the project development period into three parts. Two early parts (two-thirds) of the period were used for prediction model training. Then, the trained model predicted the number of defects reported in the remainder (one-third) of the future period.

6.2 Developer Interview for MIM Validation

For the MIM validation, we interviewed 11 developers of Table 12 when the projects were completed and asked them to answer some questions about their programming habits. Then, we compared their responses with actual measurements of their relevant MIMs. This section aimed to verify if programming habits captured by MIMs consistently corresponded to the interview results. Details about all interview questions and answer cases are presented in Table 14. The *MaxTimeInterval* notation includes *MaxTimeIntervalEditEdit*, *MaxTimeIntervalBrwsBrws*, and *MaxTimeIntervalBrwsEdit*; that is, *MaxTimeInterval* computes the maximum time interval between file-accessing interaction events in a task session.

For example, in the interview, we questioned the developers on "*How many files do you intensively edit while working on a task?*" They answered the question with multiple choices: "A – *On average, just one or two*" or "B - *Usually several files here and there due to a project property*." Later, we extracted *NumMultiTasks* and *NumParallelEdit* respectively from task sessions of one group of developers who responded with A, and the other group of developers who responded with B. We then compared MIM distributions from each of the two groups. In the example, we expected that *NumMultiTasks* and *NumParallelEdit* of group B might be higher than those of group A in terms of the boxplot comparison.

In the interview, some MIMs could not be questioned with any interview form, whereas the others could be. In interview design, for example, an MIM, such as *NumEditEvent,* is difficult to question because developers do not count and remember their actions. Therefore, we only focused on the topics on which we could do interview.

Conclusively, we confirmed that MIMs predominantly corresponded to the interview results. Some of MIM measurements for each of the response groups (e.g., A or B) were discriminative in terms of median and skewed distribution of boxplots, as shown in Figure 10. We conducted statistical tests to determine whether the medians of the response groups in Figure 10 showed statistically significant differences. We used the Wilcoxon rank sum test for two samples (i.e., groups A and B). Otherwise, we used the Kruskal-Wallis test (i.e., in a case with more than two samples). Table 13 lists the test results. The underlined pairs of questions and metrics are the statistically significant ones (95% confidence level). We could not see a statistical significance in all the cases.

TABLE 13
RESULTS OF STATISTICAL TESTS TO CHECK IF RESPONSE GROUPS OF BOXPLOTS IN FIGURE 10 SIGNIFICANTLY DIFFER

Questions and Metrics	Test method	p-value	Medians				
			A	B	C	D	E
Q1-NumMultiTasks	Wilcoxon	8.80E-12	1	3			
Q1-NumParallelEdit	Wilcoxon	0.9075	3	3			
Q2-NumRareBrws	Wilcoxon	0.00034	6.5	3			
Q2-NumRareEdit	Wilcoxon	0.0018	2	1			
Q3-RatioCodeUnderstandingEffort	Wilcoxon	0.41	0.69	0.72			
Q3-TimeSpentBeforeEdit	Wilcoxon	0.001	0.019	0.098			
Q4-TimeSpentAfterEdit	Wilcoxon	0.003	0.48	0.116			
Q5-RatioCodeUnderstandingEffort	Kruskal-Wallis	0.58	0.709	0.702	0.731		
Q6-AvgTimeIntervalEditEdit	Wilcoxon	0.54	0.16	0.2			
Q7-MaxTimeInterval	Kruskal-Wallis	8.43E-05	0.34	0.13	2.24	1.12	2.5
Q8-NumInterruptions	Wilcoxon	0.1965	3	2			

The interview results showed interesting exceptions to our expectations, as listed in Table 14. For example, two groups of developers with different opinions on "IQ1 - *How many files do you intensively edit while working on a task*?" showed almost similar distributions for *NumParallelEdit*. However, they showed totally different distributions for *NumMultiTasks* when answering the same question. It is possible that many of them misunderstood the concepts of multitasking task sessions and editing several files at the same time. Actually, *NumMultiTasks* could better classify the opinions of the two response groups than *NumParallelEdit* in response to the same question.

In terms of balancing time resource allocations between developer's actual coding and code understanding activities, *RatioCodeUnderstandingEffort* could not capture a statistically significant difference between the three response groups. As shown in the results for the IQ3 and IQ5 response groups, it was not easy to capture opinions on the developer's effort. This could have been because developers tend to exaggerate the degree of effort that they exert on their job or have trouble quantifying it.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2550458, IEEE Transactions on Software Engineering

TABLE 14
INTERVIEW QUESTIONS AND ANSWERS, AND EXPECTED OBSERVATIONS FOR RELEVANT MIMS. THE ANSWERS COLUMN CATEGORIZES CASES IN WHICH DEVELOPERS ANSWERED THE GIVEN QUESTIONS. GROUP SIZE MEANS PORTIONS OF EACH RESPONSE GROUP

Interview Questions	Answers	Relevant MIMs	Expected Observations (Group size in percentage)
IQ1 - How many files do you intensively edit while working on a task?	**A** – On average, just one or two.	*NumParallelEdit* *NumMultiTasks*	Group B might have a higher boxplot than group A. (A:B=44%:56%)
	B – Usually several files here and there due to a project property.		
IQ2 – If you find a bug during code review, where is the bug usually located?	**A** – Bugs tend to be found occasionally at unexpected code files.	*NumRareEdit* *NumRareBrws*	Group A might have a higher boxplot than group B. (A:B=78%:22%)
	B – Bugs tend to be found mainly at code entities (e.g., files or functions) that have been frequently edited so far.		
IQ3 – Do you spend enough time in understanding the code context before beginning necessary code change for an assigned task?	**A** - I usually spend enough long time in understanding context of code or in finding a necessary solution before beginning code modification in earnest for an assigned task.	*TimeSpentBeforeEdit* *RatioCodeUnderstandingEffort*	Group A might have a higher boxplot than group B. (A:B=50%:50%)
	B - I quickly plunge into assigned tasks because I am usually accustomed to them.		
IQ4 – Do you scrupulously review code changes to check if any possible bug exists or to test something further, even after you have finished a task?	**A** – Yes, I tend to carefully review my edited code for any error possibility.	*TimeSpentAfterEdit*	Group A might have a higher boxplot than group B. (A:B=67%:33%)
	B – No, I tend to finish a task right away after the last code editing because typically no problem occurs.		
IQ5 – How do you allocate portions of your effort between actual coding and code understanding in terms of time when working on a given task?	**A** – Actual coding time < code understanding time.	*RatioCodeUnderstandingEffort*	Group A might have a higher boxplot than group C, and group B might have a position between group A and C. (A:B:C=11%:56%:33%)
	B – Actual coding time ≈ code understanding time.		
	C – Actual coding time > code understanding time		
IQ6 – What is your coding style for a given task?	**A** – Once I finish understanding a given task, I tend to quickly and fluently continue editing code.	*AvgTimeIntervalEditEdit*	Group B might have a higher boxplot than group A, because developers of group B can need occasionally additional time in understanding task context between code change activities. (A:B=67%:33%)
	B – I simply start a given task and then edit and understand code, on and off, on the fly.		
IQ7 – How often do you rest while coding?	**A** – Every 10 minutes.	*MaxTimeIntervalEditEdit* *MaxTimeIntervalBrwsBrws* *MaxTimeIntervalBrwsEdit*	Group D and E might have higher boxplots than groups A and B, and group C might have a position in the middle of them. (A:B:C:D:E=10%:10%:20%:40%:20%)
	B – Every 40 minutes.		
	C – Every hour.		
	D – Every two hours.		
	E – Every three hours.		
IQ8 – How often do interruptions occur during a task?	**A** – Every hour (or more often).	*NumInterruptions*	Group A might have a higher boxplot than group B. (A:B=40%:60%)
	B – Every two hours / one or two times a day (for 10~30 minutes).		

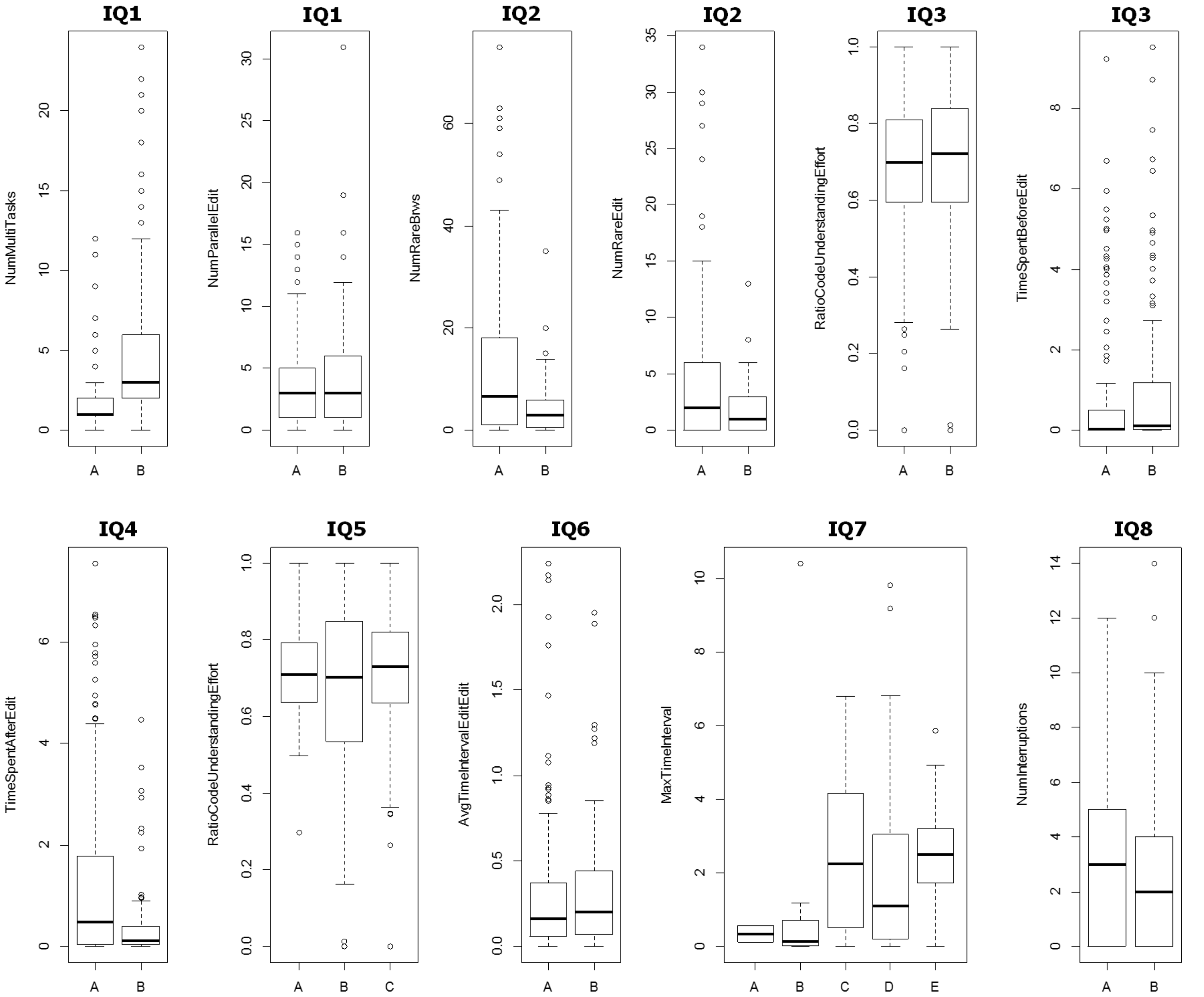


Fig. 10. Consistency comparison between interview results about programming habits of 11 industrial developers and their actual measurements of relevant MIMs. Explanation about the interview questions and answers are presented in Table 12.

As shown in the boxplots for IQ3 of Figure 10, the group of developers who answered, “A - *I usually spend a long time in understanding the context of code or in finding a necessary solution before beginning code modification in earnest for an assigned task*,” did not actually spend a relatively longer time than the other group of developers who answered, “B - *I quickly plunge into assigned tasks because I am usually accustomed to them*.” The MIM measurements of response group B were somewhat higher than those of group A. One possible reason for this phenomenon is that developers actually spend more time in understanding code than they perceive or believe, even though they do not consciously recognize it.

6.3 Performance Evaluation Result

To address RQ3 of Table 3, we built a defect prediction model using the random forest machine learner with a subset of MIMs selected by the CFS algorithm. Table 15 shows the selected MIMs in each project. The number of file instances used in performance evaluation for projects P1, P2, and P3 was 553, 62, and 269 respectively. For the evaluation, we used 10 times ten-fold cross validation.

Figure 11 presents F-measure distributions from the 10 times ten-fold cross validation process for the three projects, P1, P2, and P3. In the experiment, the ratio of buggy samples in P1, P2, and P3 were 81.4%, 19.4%, and 16.4%, respectively. Therefore, the F-measures of the dummy classifier in P1, P2, and P3 were 0.62, 0.28, and 0.25, respectively (Section 4.3).

The F-measure of the MIM-based defect prediction model significantly outperformed that of the dummy classifier in all the P1, P2, and P3 subjects. In Figure 11, F-measure medians of P1, P2, and P3 were 0.88, 1, and 0.8 respectively. Interestingly, the prediction model of P2 could perfectly predict defects in more than 50% of cases. Actually, the project P2 had a quite good condition for a prediction model to predict defects; the project P2 had relatively small size of files under development, and most of developer activities were concentrated especially on some limited files among the development files.

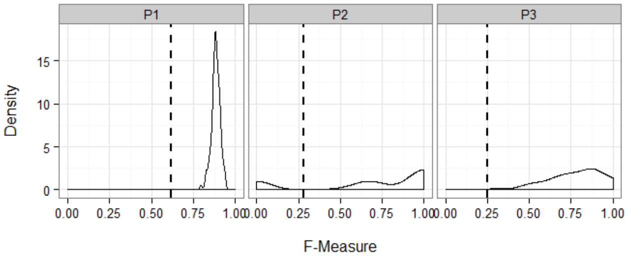


Fig. 11. Performance evaluation result of MIM-based defect prediction models for the three commercial projects (Table 12). Vertical dashed lines are performance baselines of the dummy classifier

TABLE 15
MIMS SELECTED BY CFS IN EACH COMMERCIAL PROJECT

Project	Selected MIMs
P1	TimeSpentAfterEdit, MaxTimeIntervalBrwsBrws, NumSelectionEvent, TimeSinceLastTask
P2	NumEditEvent, NumEditingDevelopers, MaxTimeIntervalBrwsEdit, NumParallelBrws, NumSelectionEvent, RatioCodeUnderstandingEffort
P3	NumEditEvent, NumRareBrws, TimeSpentAfterEdit, NumRepeatedEdit, TimeSpentBeforeEdit, NumParallelBrws, MaxTimeIntervalEditEdit, NumMultiTasks

In our case study, P1 had many defects because they were not defects reported after product release (post-delivery); rather, they were internally reported during intensive testing and QA activities within a certain time. However, P2 and P3 were relatively small-sized projects compared to P1; therefore, testing and debugging could be performed on them on the fly instead of a certain time having to be allocated for some intensive QA activities.

6.4 MIM Ranks in Different Project Domains

In Section 5.3, we investigated MIM ranks based on gain ratio. However, the ranks can be changed if project domain is changed. In this section, we addressed RQ4 of Table 3, investigating how MIM ranks change in different project domains including commercial and Eclipse projects.

Depending on projects, characteristics of developer interactions are inevitably variable. Therefore, it is natural that the MIM ranks can change depending on the project domains. Actually, there are no universally acceptable MIM ranks in terms of contribution to improving prediction performance. Nevertheless, to introduce a reference guide, we attempted to find MIM ranks that considered variation from our seven studied projects (i.e., the three projects in Table 12 and four projects in Table 7).

Figure 12 shows the results that we found from the seven projects. For each of the seven projects, we computed MIM ranks based on the gain ratio, as done in Section 5.3. Consequently, each MIM has seven rank values. The distributions of the seven values of each MIM are presented in boxplots in Figure 12, which are sorted by the median of the seven rank values.

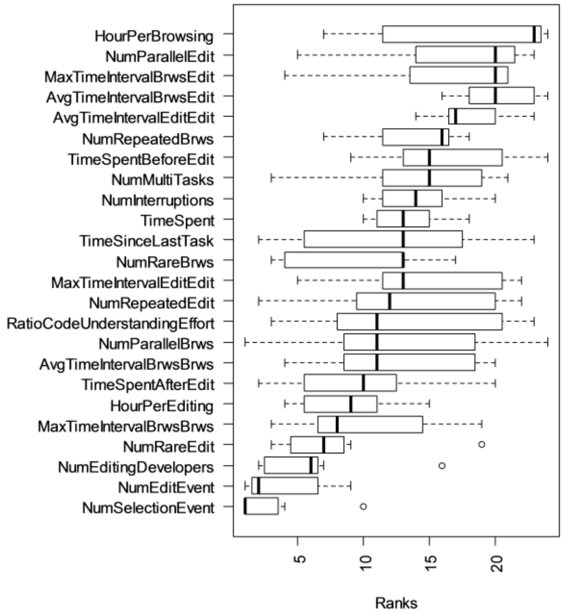


Fig. 12 Variation of MIM ranks observed over the seven projects

As shown in Figure 12, those of *NumSelectionEvent*, *NumEditEvent*, *NumEditingDevelopers*, and *NumRareEdit* were MIMs that generally ranked at high over the seven projects. In addition, those of MIMs seemed to have relatively smaller variations (i.e., shorter boxplot in length) than any others, meaning that they had relatively low project dependency. In other words, those of five MIMs looked reliable in many projects even if the applied project domains were changed.

In contrast, for example, those of *RatioCodeUnderstandingEffort*, *TimeSinceLastTask*, and *HourPerBrowsing* had comparatively larger variations (i.e., longer boxplot in length) than any others, meaning that they had relatively high project dependency. In other words, the influential power of the three MIMs could be quite variable depending on the applied projects.

7 SUMMARIZATION AND DISCUSSION

In this section, we summarize the key findings from our experiments in Sections 5 and 6, and discuss a possible application with the proposded MIMs.

7.1 Summary of Findings

The following findings basically cover our research questions (Table 3) and include additional insights of interest.

- MIMs were promising defect predictors and could improve performance up to 157% (effect size: 0.941; CM+HM: 0.313 → MIM+CM+HM: 0.494; Table 9).
- MIMs could further cost-effectively facilitate the code inspection process; e.g., up to 168% by using only 21% of a full effort (HM: 35% → MIM: 59%; Figure 7).
- MIMs still worked well in the commercial projects with good performance as defect predictors compared to the dummy classifier (Section 6.3).
- *NumSelectionEvents*, *NumEditEvents*, *NumEditingDe-*

velopers, and *NumRareEdit* MIMs were relatively good predictors in file-level defect prediction (Figure 13). Similarly, MIMs relating to the category GQ1 (How frequently do developers edit files?) were good predictors (Section 5.4). MIM ranks were variable depending on different experiment conditions and project domains so that MIM ranks reported in this paper can be different from the ones reported in our previous study [23]. Therefore, by extending case studies, we explored possible variation of the influential MIMs.

- Any long time intervals between developer's file-accessing activities were detrimental. Time interval-related MIMs are indirect indicators associated with develoepr's concentration informing of how to be often disturbed or keep working with attention in work (e.g., Table 15 and Figure 12).

7.2 Discussion on Possible Application

As a future application, the effective MIMs can be used to implement a feature for an IDE tool such as an Eclipse plug-in to detect and warn behavioral anomalies (or any inefficient behaviors) of developers. The logs of Mylyn task sessions are recorded in real-time at a local site while developers edit or navigate codes, so MIMs can be computed and visualized in real-time with a form of metrics dashboard. All observations of particular MIM values of developers who rarely make defects can be collected and modeled during development. Using the information, an IDE tool can provide developers with a warning of on-line feedback to make them cautious and to reduce a potential risk introducing defects. This type of mechanism might help novice developers to learn the good working habits of expert developers. As a result, the overall behaviors of developers can be improved. Even if this application is not feasible as a real-time tool, it can still help developers by providing any regular feedback (e.g., off-line reports) based on the MIMs come from talented developers who produce good quality of codes.

8 Threats To Validity

Threats to the validity of this study were identified as follows.

- *Developers do not always submit their task context*. A lack of Mylyn data is a major threat to validity. Sharing Mylyn data is not mandatory. Therefore, developers do not always submit their task context; moreover, they can even choose which parts of their task contexts to submit. Developers are typically not willing to share their personal task context with the public on account of privacy concerns. Therefore, data-sharing security must be agreed upon by a sourcer and analyzer in advance. In this study, we used the publicly available data of Mylyn task sessions attached to Eclipse projects and data permitted in our additional case studies of commercial projects.
- *The systems and developers referenced in this study may not be representative*. We selected projects as subjects that have available Mylyn data, which could lead to a project selection bias. In this study, 101 developers were analyzed (91 from the Eclipse projects and 11 from the commercial projects). However, given the number of developers referenced, we may not have sufficiently addressed a wide spectrum of developer types (e.g., different experience level, working conditions, programming habits, etc.) for our research questions. Therefore, additional studies are required in the future.

- *The Mylyn data that we used could be biased*. As mentioned in the first bullet point, developers may not have submitted their task context, or they could have only submitted a portion of it. The lack of task context information or the inclusion of biased Mylyn data could have affected the reliability of our conclusions. The sample size used in our experiments (e.g., 20% coverage of *All* subjects, Table 7) seemed to be sufficient to study the statistical characteristics of a population. Nevertheless, it was necessary for the sample files to be independently collected from the population to be good representatives. Therefore, to address that issue, we designed and conducted an additional statistical test, a Run test (also called a Wald–Wolfowitz test), to confirm that our sample files from the Mylyn logs were not biased, but were evenly covered distributions of the entire population (i.e., all of the file changes in the CVS logs). We used the following process for the Run test. First, we collected change files for the CVS logs and made a list X of the collected file names without redundancy. We then split list X in half, calling one half X_a and the other half X_b (X_a and X_b had the same size). Second, we collected change files from the Mylyn logs and made a list Y of the collected file names without redundancy. Third, we arbitrarily drew a file y from list Y and checked whether file y was found in either X_a or X_b. If file y was in X_a, we labeled it as "A" and otherwise as "B." However, if file y was in neither X_a nor X_b, we skipped it. We continued to draw the next file y from list Y until Y became empty. Finally, we obtained a sequence outcome like ABBAAABAB…. AB (the size of the label sequence equals the number of matched files between lists X and Y). We believe that if no bias existed in our sample collection (i.e., our studied files from the Mylyn log), the sequence outcome had to have randomness in its label enumeration. Conclusively, we applied the Run test to that sequence outcome and finally confirmed that the null hypothesis (H_0) could not be statistically rejected (the p-value was 0.922 > 0.05, 95% confidence interval). In the Run test, H_0 was "each element in the sequence is independently drawn from the same distribution."

9 Conclusion

In this paper, we proposed and evaluated MIMs for improving defect classification performance. The results of our evaluation demonstrated that MIMs significantly contributed to improving defect classification performance when used together with the existing metrics suite (CM+HM); i.e., an approximate 157% improvement was shown from 0.313 to 0.494 on average in terms of the F-measure (MIM+CM+HM vs. CM+HM; Table 9). In addition, MIMs were shown to cost-effectively facilitate code inspection; i.e., 59% of total defects could be detected by inspecting only 21% of limited source code designated by the MIM-based defect prediction model (Figure 7).

Our study extended existing knowledge in the field of software quality metrics by proposing novel metrics based on the information of micro-level developer interactions. Our findings concur with previous studies [18], [19], [20] that suggest that developer interaction patterns affect software quality.

In terms of future applications, MIMs show significant promise for a variety of IDE-centric tools, such as pre-commit warnings of dangerous changes (interaction logs are available in the IDE without major privacy concerns). Even if developers are not willing to submit their private task logs to a remote repository server, MIMs can be locally implemented.

In this paper, MIMs were designed based on Mylyn. However, the principle of MIMs can be implemented with data from other Mylyn-like alternative tools. We believe studying methods for capturing and understanding developer interactions are an emerging irreversible trend that has been already realized not only by academic studies [35][41] but also by industry products[8]. Therefore, we plan to extend MIMs by leveraging these sources of developer interaction data. In addition, we will apply MIMs to other problems, such as measuring programmer productivity and software quality.

Overall, we expect that future defect prediction models will use more information from developers' direct and micro-level interactions to improve defect prediction. MIMs are a first step in this direction.

All data used in this study is publicly available at https://sites.google.com/site/mimetrics/.

Acknowledgment

The authors are very grateful to Tim Menzies for his valuable comments on using Wilcoxon hypothesis testing, 10 times ten-fold cross validation, and the CFS feature selection algorithm.

References

[1] F. Vahid and T. D. Givargis, Embedded System Design: A Unified Hardware/Software Introduction. John Wiley & Sons, 2001.

[2] A. Hassan, "Predicting faults using the complexity of code changes," in Proceedings of the 31st International Conference on Software Engineering, ser. ICSE '09, 2009, pp. 78–88.

[3] S. Kim, E.J. Whitehead Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?" IEEE Trans. Softw. Eng., vol. 34, pp. 181–196, March 2008.

[4] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in Proceedings of the 29th International Conference on Software Engineering, ICSE '07, 2007, pp. 489–498.

[5] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Softw. Eng., vol. 33, pp. 2–13, January 2007.

[6] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in Proceedings of the 28th International Conference on Software Engineering, ser. ICSE '06, 2006, pp. 452–461.

[7] 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., vol. 31, pp. 340–355, April 2005.

[8] H. Zhang, "An investigation of the relationships between lines of code and defects," in ICSM 2009, 2009, pp. 274 –283.

[9] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in Proceedings of the 30th International Conference on Software Engineering, ser. ICSE '08, 2008, pp. 531–540.

[10] Y. Shin, A. Meneely, L. Williams, and J.A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," IEEE Transactions on Software Engineering, vol. 37, no. 6, pp. 772–787, 2011.

[11] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, May 2010, pp. 31 –41.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," Software Engineering, IEEE Transactions on, vol. 34, no. 4, pp. 485–496, 2008.

[13] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," IEEE Trans. Softw. Eng., vol. 20, pp. 476–493, June 1994.

[14] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," IEEE Trans. Softw., vol. 22, no. 12, pp. 886 –894, Dec. 1996.

[15] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in Proceedings of the 30th International Conference on Software Engineering, ICSE '08, 2008, pp. 181–190.

[16] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice – Experiences from Windows," in Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, 2011, pp. 357–366.

[17] Cliff Norman, "Dominance statistics: Ordinal analyses to answer ordinal questions," Psychological Bulletin, Vol 114(3), Nov 1993, 494-509.

[18] T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in Proceedings of the 28th International Conference on Software Engineering, ser. ICSE '06, 2006, pp. 492–501.

[19] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," Software Quality Journal, vol. 19, no. 1, pp. 5–34, Aug. 2010.

[20] A. Ko and B. Myers, "A framework and methodology for studying the causes of software errors in programming systems," J. Vis. Lang. Comput., vol. 16, pp. 41–84, February 2005.

[21] M. Kersten and G. Murphy, "Mylar: A degree-of-interest model for ides," in Proceedings of the 4th International Conference on Aspect-oriented Software Development, ser. AOSD '05, 2005, pp. 159–168.

[8] Tasktop, https://www.tasktop.com

[22] Thilo Mende and Rainer Koschke, "Revisiting the evaluation of defect prediction models," In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, 2009

[23] T. Lee, J. Nam, D. Han, S. Kim, and H.P. In, "Micro interaction metrics for defect prediction," in Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ser. ESEC/FSE '11.

[24] V.R. Basili, L.C. Briand, and W.L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Trans. Softw. Eng., vol. 22, pp. 751–761, October 1996.

[25] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," IEEE Trans. Softw., vol. 29, no. 4, pp. 297 – 310, april 2003.

[26] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Trans. Softw., vol. 31, no. 10, pp. 897 – 910, oct 2005.

[27] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in Proceedings of the 27th International Conference on Software Engineering, ICSE '05, 2005, pp. 580–586.

[28] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in Proceedings of the Third International Workshop on Predictor Models in Software Engineering, PROMISE '07, 2007.

[29] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proceedings of the 27th International Conference on Software Engineering, ICSE '05, 2005, pp. 284–292.

[30] H. Hata and O. Mizuno, "Bug Prediction Based on Fine-Grained Module Histories," ICSE 2012, 2012.

[31] F. Khomh, M. D. Penta, Y.-G. Gu´eh´eneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," Empirical Software Engineering, vol. 17, no. 3, pp. 243–275, Aug. 2011.

[32] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, pp. 13–23.

[33] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?" in Fundamental Approaches to Software Engineering, vol. 6013, 2010, pp. 59–73.

[34] M. Kersten and G. Murphy, "Using task context to improve programmer productivity," in Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, 2006, pp. 1–11.

[35] G.C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" IEEE Softw., vol. 23, pp. 76–83, July 2006.

[36] A. Ying and M. Robillard, "The influence of the task on programmer behaviour," in Program Comprehension, 2011. ICPC'11. 19th IEEE International Conference on, June 2011.

[37] R. Robbes and M. Lanza, "Characterizing and Understanding Development Sessions," in Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on, 2007, pp. 155–166.

[38] R. Robbes, D. Pollet, and M. Lanza, "Logical Coupling Based on Fine-Grained Change Information," in Reverse Engineering, 2008. WCRE '08. 15th Working Conference on, 2008, pp. 42–46.

[39] Z. Lijie M.W. Godfrey, and A.E. Hassan, "Detecting interaction coupling from task interaction histories," in Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on, June 2007, pp. 135 –144.

[40] N. Bettenburg and A. Hassan, "Studying the impact of social structures on software quality," International Conference on Program Comprehension, vol. 0, pp. 124–133, 2010.

[41] Zhongxian Gu, "Capturing and exploiting fine-grained IDE interactions," Software Engineering (ICSE), 2012 34th International Conference on , vol., no., pp.1630,1631, 2-9 June 2012

[42] A. Liaw and M. Wiener, "Classification and regression by randomforest", R news, vol. 2, no. 3, pp. 18-22, 2002.

[43] G. C. V. Basili and H. D. Rombach, "The Goal Question Metric Approach," 1994.

[44] R.E. Walpole, R. Mers, and S.L. Myers, Probability & Statistics for Engineers & Scientists. Pearson Prentice Hall, 2006.

[45] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, "The weka data mining software: An update," SIGKDD Explor. Newsl., vol. 11, pp. 10–18, November 2009.

[46] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in Proceedings of the International Conference on Software Maintenance, 2000.

[47] J. 'Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in Proceedings of the 2005 International Workshop on Mining Software Repositories, ser. MSR '05, 2005, pp. 1–5.

[48] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu, "Don't touch my code!: examining the effects of ownership on software quality", ESEC/FSE '11, pages 4–14, 2011

[49] E. Alpaydin, Introduction to Machine Learning, 2nd ed. The MIT Press, 2010.

[50] S. Scott and S. Matwin, "Feature engineering for text classification," in Proceedings of the Sixteenth International Conference on Machine Learning, ser. ICML '99, 1999, pp. 379–388.

[51] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Datasets," in ESEC/FSE'09, 2009.

[52] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "BugCache for inspections: hit or miss?" in ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM Request Permissions, Sep. 2011.

[53] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," Empirical Software Engineering, vol. 13, no. 5, pp. 473–498, Oct. 2008.

[54] 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 Engineering, vol. 17, no. 4, pp. 375–407, Dec. 2010.

[55] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," Journal of Systems and Software, vol. 83, no. 1, pp. 2–17, 2010.

[56] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," in Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, 2010, pp. 107–116.

[57] Shivaji, S. and Whitehead, E.J. and Akella, R. and Sunghun Kim, "Reducing features to improve bug prediction," in Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on, nov. 2009, pp. 600 –604.

[58] D. Kim, X. Wang, S. Kim, A. Zeller, S.C. Cheung, and S. Park, "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," IEEE Trans. Softw. Eng., vol. 99, 2011.

[59] M. Hall, "Correlation-based feature selection for machine learning, 1998," Deptartment of Computer Science, Univiversity of Waikato.

[60] M. Hall and G. Holmes, "Benchmarking Attribute Selection Techniques

for Discrete Class Data Mining," IEEE Trans. on Knowl. and Data Eng., vol. 15, no. 6, pp. 1437–1447, 2003.

[61] Foyzur Rahman and Premkumar Devanbu, "How, and Why, Process Metrics Are Better", ICSE 2013.

[62] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? findings from a google case study", ICSE'13, 2013, pp. 372-381.

[63] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the NSSE and other surveys?", Annual meeting of the Florida Association of Institutional Research, 2006.

[64] Philip B. Crosby, "Quality is Free: The Art of Making Quality Certain: How to Manage Quality – So That It Becomes A Source of Profit for Your Business," McGraw-Hill Companies, 1979.

[65] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models", Empirical Software Engineering, 2015.

[66] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," ICSE'15, 2015, vol.1, no., pp.789-800.