Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1659

An empirical study of the effect of file editing patterns on software quality

Feng Zhang[1,*,†], Foutse Khomh[2], Ying Zou[3] and Ahmed E. Hassan[4]

[1]*School of Computing, Queen's University, Kingston, ON, Canada*
[2]*SWAT, École Polytechnique de Montréal, Québec, Canada*
[3]*Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada*
[4]*Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Kingston, ON, Canada*

ABSTRACT

Developers might follow different file editing patterns when handling change requests. Existing research has warned the community about the potential negative impacts of some file editing patterns on software quality. However, very few studies have provided quantitative evidence to support these claims. In this paper, we propose four metrics to identify four file editing patterns: concurrent editing pattern, parallel editing pattern, extended editing pattern, and interrupted editing pattern. Our empirical study on three open source projects shows that 90% (i.e. 1935 out of 2140) of files exhibit at least one file editing pattern. More specifically (1) files that are edited concurrently by many developers are 1.8 times more likely to experience future bugs than files that are not concurrently edited; (2) files edited in parallel with too many other files by the same developer are 2.9 times more likely to exhibit future bugs than files individually edited; (3) files edited over an extended period of time are 1.9 times more likely to experience future bugs than other files; and (4) files edited with long interruptions have 2.0 times more future bugs than other files. We also observe that the likelihood of future bugs in files experiencing all the four file editing patterns is 3.9 times higher than in files that are never involved in any of the four patterns. We further investigate factors impacting the occurrence of these file editing patterns along three dimensions: the ownership of files, the type of change requests in which the files were involved, and the initial code quality of the files. Results show that a file with a major owner is 0.6 times less likely to exhibit the concurrent editing pattern than files without major owners. Files with bad code quality (e.g. high McCabe's complexity, high coupling between objects, and lack of cohesion) are more likely to experience the four editing patterns. By ensuring a clear ownership and improving code quality, the negative impact of the four patterns could be reduced. Overall, our findings could be used by software development teams to warn developers about risky file editing patterns.

Received 23 April 2013; Revised 21 May 2014; Accepted 10 June 2014

KEY WORDS: file editing pattern; bug; defect; software quality; change request; ownership; empirical software engineering; mylyn

1. INTRODUCTION

A software bug is a defect that causes software to behave in unintended ways or to produce incorrect or unexpected results. It is estimated that 80% of software development costs are spent on bug fixings [1]. Bugs are generally introduced inadvertently by developers when performing source code changes. During the implementation of a change request, a developer might change one or many files. File changes are done through editing each involved file one or several times. As developers might work concurrently on several change requests in parallel, several file editing patterns emerge. For example, one developer might

*Correspondence to: Feng Zhang, School of Computing, Queen's University, Kingston, ON, Canada.
†E-mail: feng@cs.queensu.ca

edit multiple files simultaneously; another developer might edit files one by one; and some others may follow both editing patterns. Developers often follow the editing pattern that best suits their personal skills, schedule constraints, and programming experiences. Understanding how file editing patterns impact software quality is of significant interest for software organizations. It is important to raise the awareness of development teams about the risky editing patterns followed by developers. There has been a large body of work on awareness tools for software development [2–6]. For example, Codebook [4] and Crystal [5] can warn developers about potential file editing conflicts. However, very few studies empirically investigated the risks posed by lack of developers' awareness about file editing patterns of fellow team members. The relationship between file editing patterns and bugs has yet to be studied in details.

To examine the effect of file editing patterns on software quality, one needs detailed information about file editing activities occurring in developers' workspaces. A tool such as Mylyn [7] that records and monitors developer's programming activities, like the selection and the editing of files, provides the opportunity for such a study. Mylyn is an Eclipse plug-in that records a developer's interactions with the integrated development environment (IDE). The interactions include the selection and editing of files. Based on Mylyn logs, we propose four metrics to identify four file editing patterns: concurrent editing, parallel editing, extended editing, and interrupted editing. Concurrent editing pattern occurs when several developers edit the same file concurrently. Parallel editing pattern occurs when multiple files are edited in parallel by the same developer. Extended editing pattern occurs when developers spend longer time editing a file, for example, the duration of editing periods longer than 123 h in our data set. Interrupted editing pattern occurs when developers observe long idle time during editing of a file, for example, the duration of idle periods longer than 816 h in our data set.

We perform an empirical study to analyze the relations between these patterns and the occurrences of bugs. Because of the availability of Mylyn logs, we choose three Eclipse projects: Mylyn,[§] Eclipse Platform,[¶] and Eclipse Plug-in Development Environment (PDE).[‖] We collect the Mylyn logs for a period of 2 years (i.e. from January 1, 2009 to December 31, 2010) to identify file editing patterns and extract the commit logs of 6 months (i.e. from January 1, 2011 to June 30, 2011) to mine future bugs (i.e. bugs reported after developers' changes). We briefly summarize our findings as follows:

- **Concurrent editing**: On average, files that are edited concurrently by many developers have 1.2 times more future bugs than files that are not involved in any concurrent editing.
- **Parallel editing**: On average, files edited in parallel with too many other files (i.e. more than 14.33 files, the third quartile in our data set) by the same developer are 2.9 times more likely to experience bugs in the future than files edited individually.
- **Extended editing**: On average, files edited over a period of time greater than the third quartile (i.e. 123 h in our data set) are 1.9 more likely to experience future bugs than other files.
- **Interrupted editing**: On average, files edited with interruption time greater than the third quartile (i.e. 816 h in our data set) are 2.0 more likely to exhibit future bugs than the files with short interruptions (i.e. below or at the median).

When more than one editing pattern is followed by one or many developers during the editing of a file, the risk of future bugs in the file increases further. For example, files edited following concurrent, extended and interrupted patterns together are 3.9 times more likely to experience future bugs than files that are never edited following any of the four patterns.

Because the four file editing patterns impact software quality negatively, it is necessary to understand factors that are related to their occurrence. The factors are selected along three dimensions: the ownership of files, the types of change requests in which the files were involved, and the initial code quality of the files. We further investigate the joint effect of different factors and file editing patterns on the likelihood of future bugs. We obtained the following results:

- **Concurrent editing**: Files with a major owner are 0.6 times less likely to exhibit the concurrent editing pattern than files without a major owner. Files without a major owner are 2.4 times more

[§]http://www.eclipse.org/mylyn.
[¶]http://www.eclipse.org/platform.
[‖]rlhttp://www.eclipse.org/pde.

DOI: 10.1002/smr

likely to experience future bugs if they are edited following the concurrent editing pattern, compared with files without a major owner and not involved in the concurrent editing pattern. The concurrent editing pattern further increases the risk for bugs in files with bad quality (i.e. high McCabe's complexity, high coupling between objects, and lack of cohesion in methods).

- **Parallel editing**: Files with a major owner are 0.7 times less likely to exhibit the parallel editing pattern than files without a major owner. However, regarding the ownership of files and the type of change requests, the risk of files edited following parallel editing pattern does not increase significantly. But when the quality of a file is poor, the risk for future bugs significantly increases when the file is edited following the parallel editing pattern.
- **Extended editing**: Files with a major owner are 0.5 times less likely to exhibit the extended editing pattern than files without a major owner. Files without a major owner but involved in extended editing are 2.1 times more likely to experience future bugs than files without a major owner and not involved in an extended editing. The risk for bugs in files with bad quality significantly increases when edited following the extended editing pattern.
- **Interrupted editing**: Files with a major owner are 0.5 times less likely to exhibit the interrupted editing pattern than files without a major owner. Moreover, when a file edited following the interrupted editing pattern does not have a major owner, the risk for future bugs is 2.2 times higher than when there is a major owner for the file. When the quality of a file is poor, the risk for future bugs significantly increases if the file is edited following the interrupted editing pattern.

This paper extends our previous work [8] that was published in the proceedings of the 19th Working Conference on Reverse Engineering (WCRE) in the following ways:

- Our earlier paper examines the effect of file editing patterns on software quality. In this paper, we further investigate the factors that impact the occurrence of each of the four file editing patterns. We investigate the ownership of files, the types of change requests, and the initial code quality of the files.
- The earlier work controlled only the confounding effect of size and the number of changes. In this paper, we control for more factors, such as the ownership of files, the type of change requests, and the code metrics. We investigate the joint effect of aforementioned factors and file editing patterns on the occurrence of future bugs.

The remainder of this paper is structured as follows. We describe the four file editing patterns in Section 2. Section 3 provides some background on the task and application life cycle management framework Mylyn. Section 4 introduces the setup of our case study and describes our analysis approach. Section 5 presents the results of our study. Section 6 discusses threats to the validity of our study. Section 7 relates our study with previous work. Finally, Section 8 summarizes our findings and outlines some avenues for future work.

2. FILE EDITING PATTERNS

This section introduces the four file editing patterns of our study.

2.1. Concurrent editing pattern

During the development and maintenance activities, developers are sometimes assigned inter-dependant change requests. As illustrated in Figure 1, this situation can result in some file (i.e. *File #1*) being edited concurrently by different developers (i.e. *Developer #1* and *Developer #2*) at the same time. We refer to this phenomenon as the concurrent editing pattern. An example of file edited following the concurrent editing pattern is the file *BugzillaTaskEditorPage.java* of the Mylyn project which was modified concurrently by three developers named *Frank*, *Steffen Pingel*, and *David Green*. The concurrent editing pattern poses the risk of one developer overriding changes from another developer or introducing a bug because of some unnoticed changes in the file because each developer is performing his or her edits independently in his or her own working space, before all the changes are merged together eventually.

DOI: 10.1002/smr

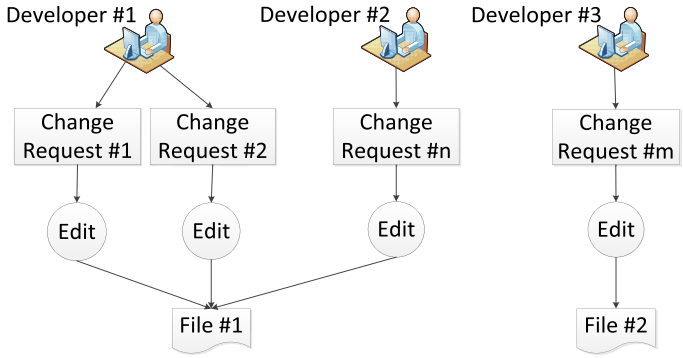

Figure 1. An illustrative example of the concurrent editing pattern. The file *File #1* is edited concurrently by two developers *Developer #1* and *Developer #2*, and the file *File #2* is edited solely by the developer *Developer #3*.

2.2. *Parallel editing pattern*

Developers sometimes edit a number of files in parallel when performing a change request. As illustrated in Figure 2, some files (i.e., *File #1*, *File #2*, *File #3*, …, and *File #n*) can be edited in parallel by a single developer (i.e. Developer #1). We refer to this phenomenon as the parallel editing pattern. An example of parallel editing occurred in the Mylyn project among the files *PlanningPerspectiveFactory.java*, *AbstractTaskEditorPage.java*, and *TasksUiPlugin.java*, which were changed simultaneously by a developer named tFrank. With the parallel editing pattern, a developer has a higher chance to become distracted because of frequent switches between files.

2.3. *Extended editing pattern*

When changing a file as part of a change request, a developer might end up performing several edits on the file. As illustrated in Figure 3, these edits might be done over a short period of time (i.e. for files *File #1* and *File #3*) or might be done over an extended period of time (i.e. for file *File #2*). We refer to the second scenario as the extended editing pattern. An example of extended editing occurred in the Mylyn project. The maximum editing time of *DiscoveryViewer.java* is 54 h, which is 35 times the median (i.e. 1.53 h) of the maximum editing times of all files in the Mylyn project. We conjecture that extended edits are possibly risky because developers might be distracted and forget what they have done since the last edit. The extended edits might also be a sign of a complex change that requires the developer to spend several editing sessions on the file.

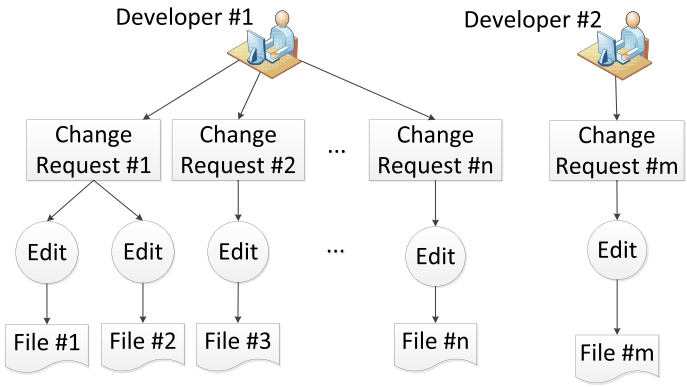

Figure 2. An illustrative example of the parallel editing pattern. The developer *Developer #1* edits files *File #1*, *File #2*, *File #3*, …, and *File #n* in parallel, and the developer *Developer #2* only edits the file *File #m*.

DOI: 10.1002/smr

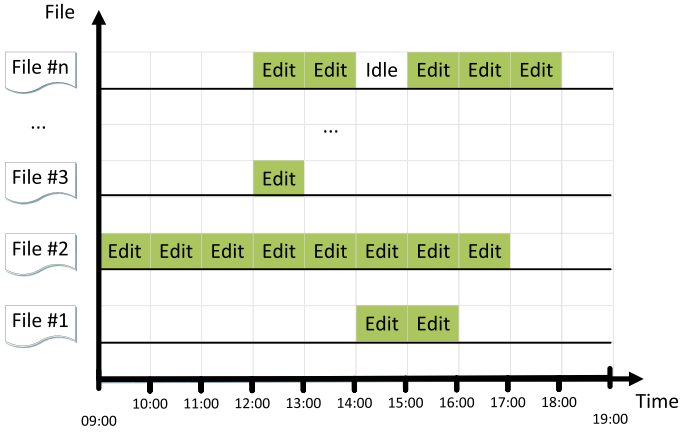

Figure 3. An illustrative example of the extended editing pattern. The file *File #2* is edited extensively than the files *File #1* and *File #3*.

2.4. *Interrupted editing pattern*

During development activities, developers are often interrupted by email alerts, meetings, or other duties. As illustrated in Figure 4, they might also simply take a break which may last for a few minutes (i.e. *File #1* and *File# n*) or longer (i.e. *File #2*). We refer to this phenomenon as the interrupted editing pattern. An example of interrupted editing occurred in the Mylyn project. The maximum interruption time of *TaskCompareDialog.java* is 3723 h, which is 965 times the median (i.e. 3.86 h) of the maximum interrupted durations of all files in the Mylyn project. The interrupted editing pattern poses the risk of developers introducing bugs because of a failure to recall some previous changes.

3. BACKGROUND OF MYLYN

3.1. *Mylyn*

Mylyn is an Eclipse plug-in for task management. A developer can create a Mylyn task to track the code changes when handling a change request. The developer's programming activities are monitored by Mylyn to create a 'task context' to predict relevant artifacts of the task. The programming activities include selection and editing of files. We use such activities to detect file editing patterns. In Mylyn, each activity is recorded as an interaction event between a developer and the IDE. There are eight types

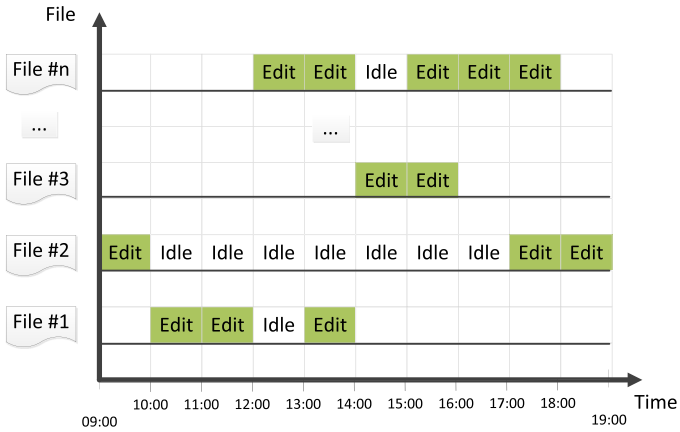

Figure 4. An illustrative example of the interrupted editing pattern. The edit of the file *File #2* is interrupted for a longer time than the edits of the files *File #1* and *File #n*.

DOI: 10.1002/smr

Table I. Event types in Mylyn logs.

Event type	Description	Developer initiated?
Command	Click buttons, menus, and type in keyboard shortcuts	Yes
Edit	Select any text in an editor	Yes
Selection	Select a file in the explorer	Yes
Attention	Update the meta-context of a task activity	No
Manipulation	Directly manipulate the degree of interest (DOI) value through Mylyn' user interface	No
Prediction	Predict relevant files based on search results	No
Preference	Change workbench preferences	No
Propagation	Predict relevant files based on structural relationships (e.g. the parent chain in a containment hierarchy)	No

```
<?xml version="1.0" encoding="UTF-8"?>
<InteractionHistory
    Id="https://bugs.eclipse.org/bugs-311966"
    Version="1">
    <InteractionEvent
        StartDate="2010-06-25 11:27:23.935 EDT"
        EndDate="2010-06-25 11:27:27.777 EDT"
        Kind="edit"
        OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
        StructureHandle="/org.eclipse.mylyn.bugzilla.ui/src/org/eclipse/mylyn/
                         internal/bugzilla/ui/tasklist/BugzillaConnectorUi.java"
        StructureKind="resource"
        Interest="2.0"
        Delta="null"
        Navigation="null"
    />
    ...
    <InteractionEvent
    ...
    />
</InteractionHistory>
```

Figure 5. Structure of the Mylyn log of bug #311966.

of interaction events in Mylyn, as described in Table I. Three types of interaction events are triggered by a developer, that is, *Command*, *Edit* and *Selection* events. In this study, we use *Edit* events to detect file editing patterns.

3.2. *Mylyn log format*

Each Mylyn log has a task identifier, which often contains the change request ID. A Mylyn log is stored in an XML format. Its basic element is *InteractionEvent* that describes the event. The descriptions include the following: a starting date (i.e. *StartDate*), an end date (i.e. *EndDate*), an event type (i.e. *Kind*), the identifier of the UI affordance that tracks the event (i.e. *OriginId*), and the names of the files involved in the event (i.e. *StructureHandle*). Figure 5 presents an example of *InteractionEvent* that was recorded during the implementation of the bug #311966**).

4. CASE STUDY SETUP

This section presents the design of our case study, which aims to address the following research questions:

(1) Do file editing patterns lead to more bugs?
(2) Which factors contribute to the occurrence of file editing patterns?

**https://bugs.eclipse.org/bugs/show_bug.cgi?id=311966.

DOI: 10.1002/smr

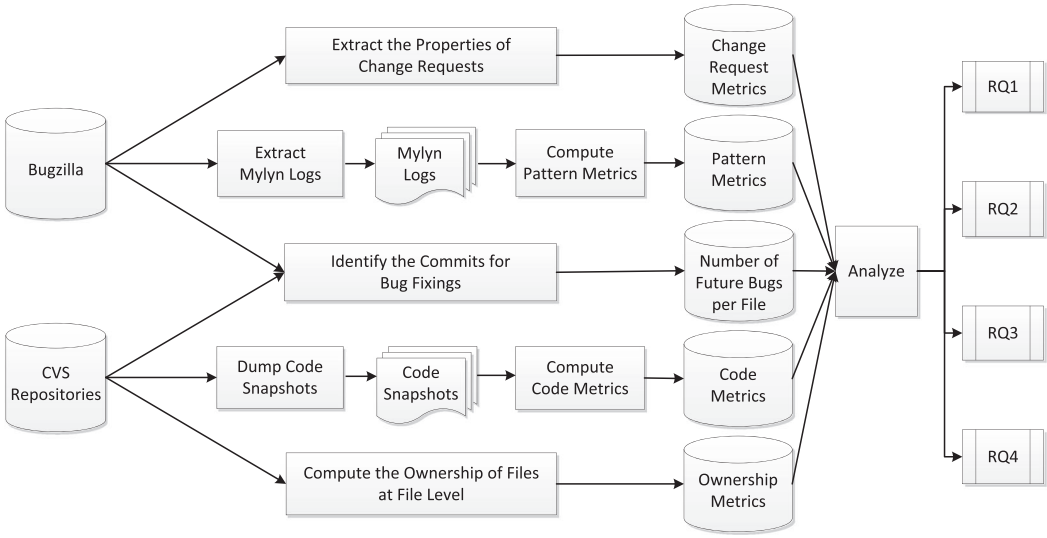

Figure 6. Overview of our approach to analyze the effect of file editing patterns on software quality.

(3) Which factors affect the bug proneness of files edited following the patterns?
(4) Do interactions among file editing patterns lead to more bugs?

4.1. Approach

The overview of our approach is presented in Figure 6. There are two types of data sources: Eclipse Bugzilla[††] that stores the property and the corresponding Mylyn logs of each change request; and Concurrent Versions System (CVS) repositories that store the source code and code change history. Change request metrics are computed from the properties of change requests. From Mylyn logs, four pattern metrics are computed to identify the four file editing patterns: concurrent editing, parallel editing, extended editing, and interrupted editing. We mine the CVS commits that contain change request IDs. With a change request ID, we retrieve the properties of the change request and determine whether a commit was for a bug fixing or a feature enhancement. From the CVS repositories, we further dump source code snapshots to compute code metrics and examine code change history to compute the ownership of files at file level.

To study the effect of file editing patterns on software quality, we statistically compare the proportion of buggy files (and the number of bugs in files) edited following the patterns. We investigate if the ownership of files, the type of a change request (e.g. a bug fix or feature enhancement), and the initial code quality (i.e. complexity, coupling, cohesion, abstraction, encapsulation, and documentation measured by static metrics) play a role in the occurrence of editing patterns. We further investigate if the ownership of files, the type of change requests, and the initial code quality affect the risk of experiencing future bugs in the files edited following certain patterns.

The remainder of this section elaborates on the details of the steps.

4.2. Subject projects

This study uses Mylyn interaction logs to identify file editing patterns. Hence, the subject systems are chosen based on the number of Mylyn logs. Mylyn is frequently used in Eclipse projects. We search all Eclipse projects and sort them by the number of change request reports with Mylyn logs. Mylyn logs are compressed and attached in a change request report. The attachment is always named as 'mylyn-context.zip'. We use the 'Advanced Search' tool on Eclipse Bugzilla webpage. In the 'Custom Search' section, we choose the field 'Attachment filename', select the condition 'is equal to', and type the query

[††]https://bugs.eclipse.org.

DOI: 10.1002/smr

Table II. List of subject projects.

Name	Description	Date of creation	Number of files	Total lines of code
Mylyn	Task and application lifecycle management framework	2005-06-17	1989	198 785
Platform	Core frameworks, services, and runtime provider for Eclipse	2001-04-28	61 366	2 284 564
PDE	A comprehensive tool set for developing Eclipse plug-ins	2001-06-05	320	78 864

The date of creations are determined by the first commit made in each project. The descriptions are obtained using all java files in the snapshot of 2010-12-31.

Table III. The descriptive statistics of the subject projects.

Name	Number of change request reports	Number of Mylyn logs	Number of files	Total lines of code	Number of developers
Mylyn	2722	3883	1177	143 164	72
Platform	606	793	738	142 674	22
PDE	524	638	225	46 530	39

The statistics are obtained only using the java files involved in editing activities that were recorded in Mylyn logs.

string 'mylyn-context.zip'. In the result page, the Eclipse Bugzilla shows a limited list of change request reports. After clicking the 'See all search results for this query' link, the Eclipse Bugzilla shows the full list of change request reports containing Mylyn logs. We choose three projects having most change request reports with Mylyn logs. The three projects are: Mylyn, Eclipse Platform, and Eclipse PDE. Table II briefly describes the three subject projects. The descriptive statistics of the files that were recorded in Mylyn logs are described in Table III.

4.3. Mining future bugs

In our case study, software quality is measured by the occurrence and the number of future bugs (i.e. bugs reported after developers' changes) in files. The bug fixing change information for each file is mined from the version control system. The three subject projects use CVS to manage source code changes. The whole history of revisions to the source code can be extracted from CVS repositories. We downloaded the CVS repositories of our three subject projects from Eclipse Archives[‡‡] on October 20, 2011. To separate the pattern analysis period from the period for counting future bugs, we selected the date of January 1, 2011 as our split date. A 6-month period is often used to count the future bugs in the studies on bug prediction, for example, the widely cited work by Zimmerman *et al.* [9]. Lee *et al.* [10] apply a 8:2 time split to separate the period for computing Mylyn metrics from the period for counting the future bugs. This paper complies the two rules. That is, we chose to use the 6-month period (20%) after the split date (i.e., January 1, 2011) as the future bug counting period and the 2-year period (80%) before the split date as the pattern analysis period.

To mine future bugs, we first extracted the change logs of all commits performed during the future bug counting period (i.e. from January 1, 2011 to June 30, 2011 in this study). During this period, the developers might implement enhancements or fix bugs. Moreover, the fixed bugs might be reported before and after January 1, 2011. It is necessary to check the corresponding type (i.e. enhancement or bug) and report date for each bugID. We manually inspected the change logs and found that the bug ID is often contained in the change logs. The bug IDs in Eclipse projects are integers. We observed that the bug IDs monotonically increase along the report date. The last bug reported on December 31, 2010

[‡‡]http://archive.eclipse.org/arch.

DOI: 10.1002/smr

is #333371, and the first bug reported on July 1, 2011 is #350890. Hence, this study automatically extracted integers greater than 333 371 and less than 350 890 from the change logs as bug IDs. To determine the type for each bug ID, we downloaded the corresponding change request report from Bugzilla. All enhancements were filtered out. In total, we obtained 104 future bugs from the 2140 files in the three projects. The number of future bugs of Mylyn, Platform, and PDE are 48, 39, and 17, respectively. We further calculate the density of future bugs by dividing the number of future bugs of a file by the size of the file. The density of future bugs of each project is presented in Figure 7. Similar to [10, 11], we combined data from the three projects because of their small sizes.

4.4. Recovering file edit history

To collect the information of our file editing patterns, we analyzed Mylyn logs recorded during the pattern analysis period (i.e. from January 1, 2009 to December 31, 2010 in this study). Mylyn logs are compressed, encoded under the Base64 format, and attached to the change request reports. We downloaded the change request reports and extracted the properties of the reports, such as reporting date, reporter's name, change request type (i.e. bug fix or feature enhancement), project and module names, comments, Mylyn attachments, and attachers' names. We decoded the Mylyn attachments from the Base64 format, and unzipped them to extract Mylyn logs. We parsed each Mylyn log to extract the *Edit* events. This study relies on the *Edit* events to track developers' accesses to files and compute the duration of developers' file editing periods. As mentioned in Section 3.1, the *Edit* events are issued when a developer selects the content (i.e. the text) of a file in the Eclipse IDE. For each *Edit* event, we extract the start date, the end date, and the names of the files concerned by the event. There is no explicit ownership information stored in Mylyn logs. Hence, we considered the *attacher* of a Mylyn log as the developer whose activities were recorded in the the Mylyn log.

4.5. Identifying file editing patterns

This section describes how we detect editing patterns followed by developers during code changes. We propose a set of metrics based on the information of the *Edit* events collected from Mylyn logs. In the following subsections, we discuss the detection of each editing pattern in details.

4.5.1. Concurrent editing pattern. For each file, we identify all edits involving the file using the Mylyn logs; for each edit, we track concurrent edits involving the file. We compute the number of concurrent edits for each file and the number of developers involved in the edits. The number of changes made to a file is known to be related to the number of future bugs in the file [12]. We control for that by dividing the number of concurrent edits and the number of developers by the number of changes, following respectively Eqns. (1) and (2). We obtain the average number of concurrent edits per change ($N_{ConcurrentEdits}$) and the average number of developers editing the file concurrently during a change ($N_{ConcurrentDevs}$).

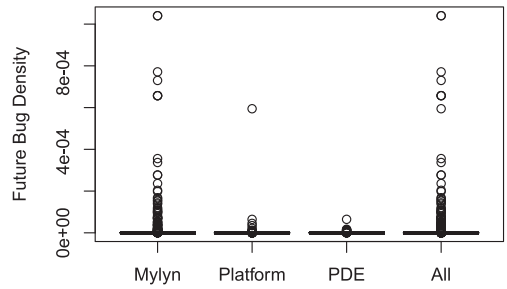

Figure 7. Box plot of the density of the future bugs in each individual project (i.e. Mylyn, Platform, and PDE) and the combined data set of the three projects.

DOI: 10.1002/smr

$$N_{ConcurrentEdits} = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\left(Overlap_{CE}\left(Edit_i, Edit_j\right)\right) \quad (1)$$

$$N_{ConcurrentDevs} = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\left(Overlap_{CD}\left(Edit_i, Edit_j\right)\right) \quad (2)$$

where $Edit_i$ represents the i^{th} Edit on the file and N is the total number of changes in the history of the file.

$Overlap_{CE}(Edit_i, Edit_j)$ equals to 1 when there is an overlap between the time windows of $Edit_i$ and $Edit_j$; otherwise, it is equal to 0. A time window of an event is the interval starting from its first timestamp to its last timestamp.

$Overlap_{CD}(Edit_i, Edit_j)$ equals to 1 when $Edit_i$ and $Edit_j$ are edited by different developers, and there is an overlap between the time windows of $Edit_i$ and $Edit_j$. In other cases, $Overlap_{CD}(Edit_i, Edit_j)$ equals to 0.

For example, given a file F involved in three edits $Edit_1$, $Edit_2$, $Edit_3$. If ($Edit_1$, $Edit_2$) and ($Edit_2$, $Edit_3$) have overlapping time windows, the number of concurrent edits for $Edit_1$, $Edit_2$, and $Edit_3$ are 1, 2, and 1, respectively. The average number of concurrent edits per change of F is $N_{ConcurrentEdits} = \frac{1}{3}(1+2+1) = 1.33$. If $Edit_1$ was performed by developer d_1, while $Edit_2$ and $Edit_3$ by developer d_2, then $Edit_1$ and $Edit_2$ were edited concurrently by d_1 and d_2; $Edit_2$ and $Edit_3$ were edited solely by d_2. The average number of developers involved in edits $Edit_1$, $Edit_2$, and $Edit_3$ are $\frac{2}{1} = 2$, $\frac{2}{2} = 1$ and $\frac{1}{1} = 1$. The average number of developers involved in concurrent edits in F is $N_{ConcurrentDevs} = \frac{1}{3}(2+1+1) = 1.33$.

We compute the $N_{ConcurrentDevs}$ value for each file from our studied projects. A file was edited following the *concurrent editing* pattern, if and only if its $N_{ConcurrentDevs}$ is greater than 0.

4.5.2. Parallel editing pattern. We compute the number of parallel editing files of an edit i ($n_{ParallelEdits}(i)$) using Eqn. (3). For each file *File*, we sum the $n_{ParallelEdits}(i)$ values of all edits i in the history of the file *File*. In order to control for the confounding effect of the number of changes made to the file, we divide the sum of $n_{ParallelEdits}(i)$ by the number of changes and obtain the average number of files in a parallel edit ($N_{ParallelEdits}$) of *File*, following Eqn. (4).

$$n_{ParallelEdits}(i) = \sum_{j=1}^{M} Overlap_{PE}\left(File, File_j\right) \quad (3)$$

where $File_j$ represents the j^{th} file in the Edit. M is the total number of files in the Edit.

$Overlap_{PE}(File, File_j)$ equals to 1 when there is an overlap between the time windows of *File* and $File_j$; otherwise, it is equal to 0.

$$N_{ParallelEdits} = \frac{1}{N}\sum_{i=1}^{N} n_{ParallelEdits}(i) \quad (4)$$

where $n_{ParallelEdits}(i)$ represents the number of parallel editing files of the i^{th} Edit of the file and N is the total number of Edits in the history of the file.

For example, given a file F involved in three edits $Edit_1$, $Edit_2$ and $Edit_3$. In $Edit_1$, F was modified in parallel with five other files; in $Edit_2$, F was modified in parallel with nine other files; in $Edit_3$, F was modified solely. The number of parallel editing files of the three edits are the following: $n_{ParallelEdits}(1) = 5$, $n_{ParallelEdits}(2) = 9$, $n_{ParallelEdits}(3) = 1$. The average number of parallel editing files of F is $N_{ParallelEdits} = \frac{1}{3}(5+9+1) = 5$.

For each file from our studied projects, we compute the $N_{ParallelEdits}$ value. We conclude that a file was modified following the *parallel editing* pattern, if and only if its $N_{ParallelEdits}$ is greater than 0.

4.5.3. Extended editing pattern. For each file, we identify all edits involving the file and compute the time span of each edit i of the file ($editTime(i)$) using Eqn. (5). We compute the $editTime(i)$ values of each edit in the history of the file and find the maximum value as *EditTime* to measure the editing time. The *EditTime* is calculated following Eqn. (6).

DOI: 10.1002/smr

$$editTime(i) = \sum_{j=1}^{M} \left(EndTime_j - StartTime_j\right) \tag{5}$$

where $StartTime_j$ and $EndTime_j$ represent the starting and ending time of the j^{th} edit event involving the file and M is the total number of Edit events in the i^{th} edit in the history of the file.

$$EditTime = maximum(editTime(i)), \forall i \in [1, N] \tag{6}$$

where $editTime(i)$ represents the time span of the i^{th} edit of the file and N is the total number of edits in the history of the file.

For example, given a file F involved in two edits $Edit_1$ and $Edit_2$, with the time spans of the two edits being respectively, $editTime(1) = 1$ h and $editTime(2) = 2$ h. The maximum editing time of F is $EditTime = maximum(1, 2) = 2$.

For each file from our studied systems, we compute the *EditTime* value. We conclude that a file was modified following the *extended editing* pattern, if and only if its *EditTime* is greater than the third quartile of all *EditTime* values.

4.5.4. Interrupted editing pattern. For each file, we identify all changes involving the file and compute the idle time of each change i ($idleTime(i)$) using Eqn. (7). We compute the $idleTime(i)$ values of each change in the history of the file and find the maximum value as *IdleTime* to measure the interruption time. The *IdleTime* is calculated following Eqn. (8).

$$idleTime(i) = \sum_{j=2}^{M} \left(StartTime_j - EndTime_{j-1}\right) \tag{7}$$

where $StartTime_j$ and $EndTime_j$ represent the starting and ending time of the j^{th} edit event changing the file and M is the total number of Edit events in the i^{th} edit.

$$IdleTime = maximum(idleTime(i)) \forall i \in [1, N] \tag{8}$$

where $idleTime(i)$ represents the idle time of the i^{th} edit on the file and N is the total number of changes in the history of the file.

For example, given a file F involved in two edits $Edit_1$ and $Edit_2$, with the interruption time of F in the two edits being: $idleTime(1) = 2$ h and $idletime(2) = 16$ h. The maximum interruption time of F is $IdleTime = maximum(2, 16) = 16$.

For each file from our studied systems, we compute the *IdleTime* value. We conclude that a file was modified following the *interrupted editing* pattern, if and only if its *IdleTime* is greater than the third quartile of all *IdleTime* values.

4.6. Determining the ownership of files

We investigate two aspects of the ownership of files: the level of the ownership and the presence of a major owner. Bird *et al*. [13] propose a metric to measure the proportion of the ownership of a developer for a particular component (i.e. a compiled binary). Their metric is the ratio of the number of commits by the developer relative to the total number of commits for a component. However, it is likely that different commits involve different numbers of lines. To obtain more precise ownership, this study weights each commit by its size as Bachmann *et al*. [14]. The size of a commit is described by the sum of the number of added lines, the number of deleted lines, and the number of modified lines.

The ownership of each file is computed using Eqn. (9).

$$Ownership\left(File_i, Developer_j\right) = \frac{\sum_{n=1}^{N_{ij}} \left(A_{ijn} + D_{ijn} + M_{ijn}\right)}{\sum_{m=1}^{D_i} \sum_{n=1}^{N_{im}} \left(A_{imn} + D_{imn} + M_{imn}\right)} \tag{9}$$

where $File_i$ is the i^{th} file and $Developer_j$ is the j^{th} developer of the file $File_i$. D_i is the number of developers of the file $File_i$, and N_{ij} is the number of commits made by the developer $Developer_j$ to

DOI: 10.1002/smr

the file $File_i$. A_{ijn}, D_{ijn}, M_{ijn} are, respectively, the number of added, deleted, and modified lines of the n^{th} commit made by the developer $Developer_j$ to the file $File_i$.

We compute the ownership of files using the pattern analysis period (i.e. from January 1, 2009 to December 31, 2010). The number of added, deleted, and modified lines of each file are calculated from all pairs of consecutive revisions made during that period. We investigate the impact of the level of the ownership of files on the occurrence of file editing patterns. In our data set, the median of the proportion of ownership of all developers for all the files is 0.9633. We further examine the impact of the presence of a major owner. Whether a file has a major owner is determined by the Eqn. (10).

$$hasMajorOwner(File_i) = \begin{cases} true & \text{if } max(Ownership(File_i)) \geq 0.9633 \\ false & \text{if } max(Ownership(File_i)) < 0.9633 \end{cases} \quad (10)$$

where $File_i$ is the i^{th} file and $max(Ownership(File_i))$ is the maximum of $Ownership(File_i, Developer_j)$ values for all $Developer_j$ that edited $File_i$.

4.7. Identifying the type of change requests

We chose to investigate the type of change requests because previous study by Ying *et al*. [15] has reported a relation between the type of change requests (i.e. an enhancement or a bug fix) and the editing style of developers. We identify the type of the change requests that were addressed by developers using the *severity* of these change requests, extracted from Bugzilla reports (refer to Section 4.4). The *severities* used in Eclipse Bugzilla are the following: *trivial*, *minor*, *normal*, *major*, *critical*, *blocker*, and *enhancement*.

However, the severity field is imprecise [15] because it is usually set either by default (i.e. *normal*) or manually by developers. Similar as Ying *et al*. [15], we group the seven severities into three coarser ones: enhancement, minor, and major and define the following three metrics:

- ENHANCEMENT—the number of enhancements implemented on a particular file
- MINORBUG—the number of minor bugs (i.e. trivial, minor, and normal) fixed on a particular file
- MAJORBUG—the number of major bugs (i.e. major, critical, and blocker) fixed on a particular file

For each file, we compute the three aforementioned metrics using the pattern analysis period (i.e. from January 1, 2009 to December 31, 2010).

4.8. Measuring the initial code quality

Files with a certain code quality (e.g. high or less complexity) might be more likely to exhibit certain file editing patterns. For example, files with high complexity may be more likely to exhibit the extended file editing pattern. Therefore, it is interesting to study whether the initial code quality of a file affects the occurrence of file editing patterns.

In our case study, we select 14 metrics commonly used to measure software maintainability. The 14 metrics are further classified into six categories (i.e. complexity, coupling, cohesion, abstraction, encapsulation, and documentation) in our previous work [19]. Table IV presents the name and the category of the 14 metrics. For the metrics at class or method level, we aggregate them to the file level by three statistics, that is, average, maximum, and sum of the metric values. In this study, we refer to the code quality of the snapshot just before the pattern analysis period as the initial code quality. The snapshot is dumped on the date of December 31, 2008.

The metrics are computed using a commercial tool named *Understand*.[§§] The mapping between the aforementioned metric names and the metric names used by the tool *Understand* is presented in Appendix A.

[§§]http://www.scitools.com.

DOI: 10.1002/smr

Table IV. List of source code metrics.

Category	Metric name	Description	Metric level	File level
Complexity	CLOC	Class lines of code	File	Value
	NIM [16]	Number of instance methods	Class	Avg, Max, Sum
	NIV [16]	Number of instance variables	Class	Avg, Max, Sum
	WMC [17]	Weighted methods per class	Class	Avg, Max, Sum
	CC [18]	McCabe cyclomatic complexity	Method	Avg, Max, Sum
Couping	CBO [17]	Coupling between objects	Class	Avg, Max, Sum
	RFC [17]	Response for a class	Class	Avg, Max, Sum
Cohesion	LCOM [17]	Lack of cohesion in methods	Class	Avg, Max, Sum
Abstraction	DIT [17]	Depth of inheritance tree	Class	Avg, Max, Sum
	IFANIN [16]	Number of immediate base classes	Class	Avg, Max, Sum
	NOC [17]	Number of immediate subclasses	Class	Avg, Max, Sum
Encapsulation	RPM	Ratio of public methods	Class	Avg, Max, Sum
Documentation	CL	comment of lines	File	Value
	RCC [16]	Ratio comments to codes	File	Value

4.9. Analysis method

We study if bugs in files are related to file editing patterns. To understand the file editing patterns, we further examine if the occurrences of file editing patterns are impacted by the ownership of files, the type of change requests in which the files were involved, and the initial code quality of the files. The joint effect of the three aforementioned aspects, and file editing patterns on bug proneness is also investigated.

4.9.1. Fisher's exact test and odds ratio (OR). The Fisher's exact test [20] is applied to determine if there are non-random associations between a particular file editing pattern and the occurrence of future bugs. If there are non-random associations, we further compute the odds ratio (OR) [20] which indicates the likelihood of an event to occur (i.e. a future bug). OR is defined as the ratio of the odds p of an event occurring in one sample, for example, the set of files edited following a specific editing pattern (experimental group), to the odds q of it occurring in the other sample, e.g. the set of files edited but not following the pattern (control group): $\text{OR} = \frac{p/(1-p)}{q/(1-q)}$. An OR of 1 indicates that the event (i.e. a future bug) is equally likely in both samples. $\text{OR} > 1$ indicates that the event is more likely in the first sample (i.e. the experimental group of files edited following the editing pattern). An $\text{OR} < 1$ indicates the opposite (i.e. the control group of files edited not following the pattern).

The Fisher's exact test [20] is also used to determine if there are non-random associations between the occurrence of a particular file editing pattern and the ownership of files, the types of change requests, and the initial code quality of files. To control the family-wise error rate of multiple comparisons, we apply the Holm–Bonferroni method [21] to correct the threshold of P-values. If there are non-random associations, we further compute the OR to quantify the impact of the aforementioned three aspects on the occurrence of file editing patterns.

Moreover, we apply the Fisher's exact test [20] to determine if there are non-random associations between the occurrence of future bugs and the presence of a particular file editing pattern together with the ownership of files, the types of change requests, and the initial code quality of files. As we conduct multiple tests, we apply the Holm–Bonferroni method [21] to correct the threshold of P-values. If there are non-random associations, we further compute the OR to investigate how much the ownership of files, the types of change requests, and the initial code quality affect the bug proneness of files edited following a particular pattern.

4.9.2. Wilcoxon and Kruskal–Wallis rank sum test. We split the full set of files into two groups: (1) the files edited following a particular pattern; and (2) the files that were edited without following the pattern. We apply the Wilcoxon rank sum test [20] to determine whether the density of future bugs of the two groups are significantly different. The Wilcoxon rank sum test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distributions of assessed variables. In

DOI: 10.1002/smr

cases of comparisons among more than two groups of files, we apply the Kruskal–Wallis rank sum test [20] that is an extension of the Wilcoxon rank sum test to more than two groups.

5. CASE STUDY RESULTS

This section presents and discusses the results of our four research questions.

Prior to answer our research questions, it is important to determine if all four editing patterns are followed by developers during code changes and are therefore worth investigating individually. We propose the following exploratory question.

5.1. Are there different file editing patterns?

This question aims to provide the quantitative data on the number of files edited by developers following the four editing patterns and determine the existence of interactions between the editing patterns, for example, if a file can be edited concurrently by different developers, over an extended period of time.

To answer this question, we identify the editing pattern(s) of a file using the metrics described in Section 4.5. The distribution of the metrics values is presented in Figure 8. We classify the files of our subject projects using the patterns followed by developers during file editing.

Table V summarizes the number of files that were edited following each pattern or combination of patterns. As shown in Table V, only 205 files in our systems were edited following none of the four patterns under investigation. The most frequent editing pattern followed by developers is the parallel editing pattern, which exists in 90% (i.e. 1922 out of 2140) of files. Moreover, 44% (i.e. 945 out of 2140) of files from our systems were edited following more than one editing pattern.

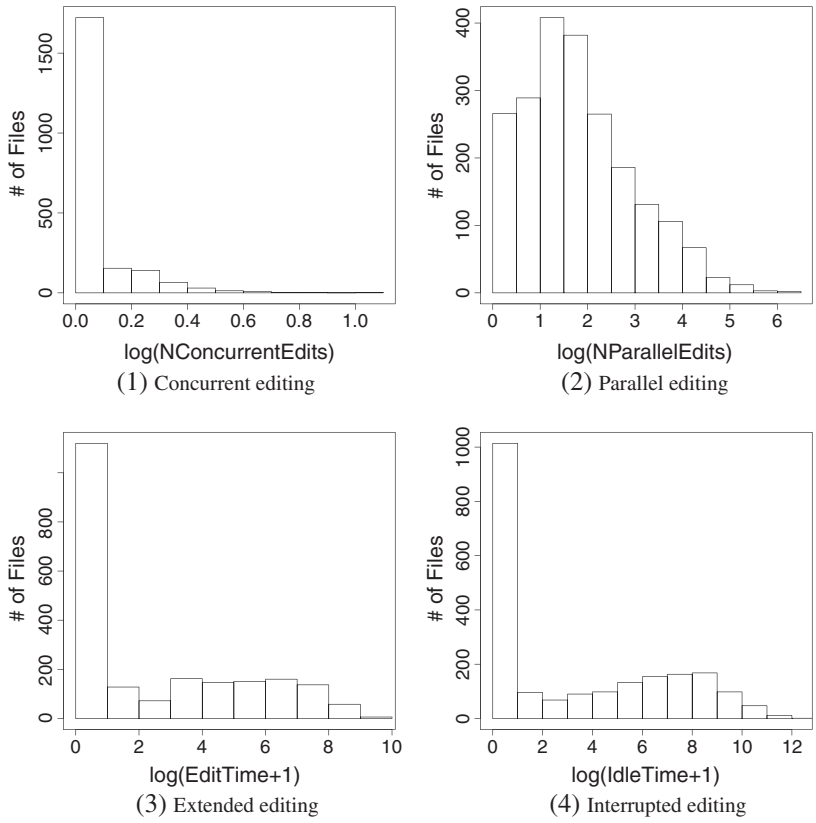

Figure 8. Distribution of metric values for the file editing patterns.

DOI: 10.1002/smr

Table V. Occurrences of file editing patterns and their interactions.

List of patterns or combination of patterns	Number and percentage of files							
	Mylyn		Platform		PDE		All	
No patterns	90	8%	74	10%	41	18%	205	10%
<Con>	437	37%	42	6%	18	8%	497	23%
<Parl>	1083	92%	655	89%	184	82%	1922	90%
<Ext>	352	30%	163	22%	20	9%	535	25%
<Int>	271	23%	224	30%	40	18%	535	25%
<Con, Parl>	434	37%	42	6%	18	8%	494	23%
<Con, Ext>	252	21%	13	2%	8	4%	273	13%
<Con, Int>	189	16%	21	3%	9	4%	219	10%
<Parl, Ext>	351	30%	158	21%	20	9%	529	25%
<Parl, Int>	270	23%	220	30%	40	18%	530	25%
<Ext, Int>	209	18%	118	16%	13	6%	340	16%
<Con, Parl, Ext>	252	21%	13	2%	8	4%	273	13%
<Con, Parl, Int>	189	16%	21	3%	9	4%	219	10%
<Con, Ext, Int>	160	14%	12	2%	6	3%	178	8%
<Parl, Ext, Int>	208	18%	118	16%	13	6%	339	16%
<Con, Parl, Ext, Int>	160	14%	12	2%	6	3%	178	8%

Con, concurrent; Parl, parallel; Ext, extended; Int, interrupted editing patterns

Overall, we conclude that developers follow the four file editing patterns during development and maintenance activities.

In the following research questions, we examine the patterns (and their interactions) in more detail to understand factors contributing to their occurrence and determine if some file editing patterns (and interaction among patterns) are more risky than others.

5.2. *RQ1: Do file editing patterns lead to more bugs?*

Motivation

In Section 5.1, we found that very frequently, developers follow one of the four file editing patterns under investigation in this study. However, following these patterns is likely to be risky. For example, during a parallel editing, a developer might become distracted because of the frequent switches between files and inadvertently introduce an error into the system. In this research question, we investigate the relation between each file editing pattern and the occurrence of bugs. Understanding the risks posed by each file editing pattern is important to raise the awareness of developers about the potential risk of their working style. Managers can use the knowledge of these patterns to decide on the acquisition of awareness tools that can assist developers during development and maintenance activities.

Approach

We classify the files based on the patterns followed by developers during file editing. Two groups will be created for each pattern P_i: (1) the group GP_i containing files that were edited by developers following P_i; and (2) the other group NGP_i containing files that were edited by developers not following P_i. We also calculate the density other than the number of future bugs of each file because previous studies (e.g. [9, 22]) have found size to be related to the number of bugs in a file. The density is computed by dividing the number of future bugs of each file by the size of the file.

For each pattern P_i, we test the two following null hypotheses:

H_{01}^1: The proportion of files exhibiting at least one future bug does not differ between the groups GP_i (of files edited by developers following P_i) and NGP_i (of files edited by developers not following P_i).
H_{01}^2: There is no difference between the density of future bugs of files from groups GP_i and NGP_i.

Hypothesis H_{01}^1 (respectively H_{01}^2) is about the probability of bugs (respectively the density of future bugs) in files edited following the pattern P_i. H_{01}^1 and H_{01}^2 are two-tailed because they

DOI: 10.1002/smr

investigate whether the file editing pattern P_i is related to a higher or a lower risk of bug. We use the Fisher's exact test and compute the OR to test H_{01}^1. We perform a Wilcoxon rank sum test for H_{01}^2.

The files in our data set do not have the same level of involvement in the patterns. For example, some files are edited concurrently by five developers, while others are edited concurrently by only two developers. Because the file edited concurrently by more developers is more at risk for conflicting changes, we believe that the level of involvement of a file in a pattern is likely to impact the risk of bugs in the file. Therefore, for each pattern, we further analyze the relation between the level of involvement in the pattern and the occurrence of bugs. The level of involvement of a file f in the:

- Concurrent editing pattern: is the average number of developers involved in a concurrent editing of f (i.e. $N_{ConDevs}$).
- Parallel editing pattern: is the average number of files edited in parallel with f (i.e. $N_{ParallelEdits}$).
- Extended editing pattern: is the average editing time of f (i.e. *EditTime*).
- Interrupted editing pattern: is the average interruption time of f (i.e. *IdleTime*).

For a concurrent or parallel (respectively an extended or interrupted) editing pattern P_i, we use the third quartile (respectively median) of all the level values of files that were involved in P_i, to split the group GP_i of files edited following P_i, into two groups GP_i^1 and GP_i^2. GP_i^1 contains files with a level of involvement lower than the third quartile (respectively median) of all the level values of files involved in P_i. GP_i^2 contains files with level values greater than the third quartile (respectively median) of all the level values of files involved in P_i. For the extended editing pattern and the interrupted editing patterns, we use the median instead of the third quartile because these patterns were defined based on the third quartile. For each pattern P_i, we test following null hypotheses:

H_{01}^3: The proportion of files exhibiting at least one bug is the same for NGP_i, GP_i^1, and GP_i^2.
H_{01}^4: There is no difference between the density of future bugs of files from groups NGP_i, GP_i^1, and GP_i^2.

Hypothesis H_{01}^3 is about the probability of bugs in files. We apply the Fisher's exact test and compute the OR to test H_{01}^3. Hypothesis H_{01}^4 is about the density of future bugs in files; we perform the Kruskal–Wallis rank sum test for H_{01}^4.

All the tests are performed using the 95% confidence level (i.e. P-value < 0.05).

Findings

(1) Concurrent editing pattern. As shown in Figure 9, the concurrent editing pattern is the most risky pattern among the four patterns. The likelihood of bugs in a file edited following the concurrent editing pattern is higher compared with files edited following one of the other three patterns.

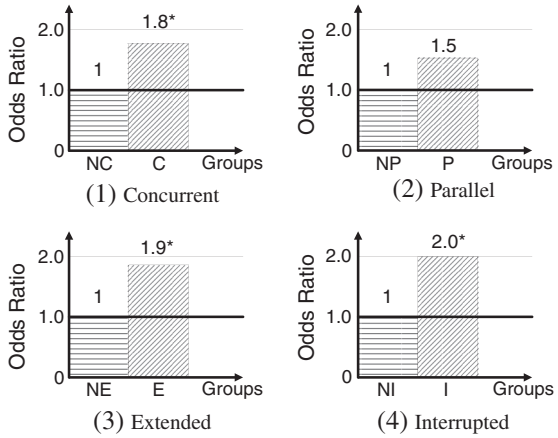

Figure 9. Odds ratio between files that are not involved in file patterns and files that are involved in file editing patterns. (* indicates that the test was statistically significant, i.e. P-value < 0.05.)

DOI: 10.1002/smr

The Fisher's exact test was statistically significant (P-value = $2.03e-03$), therefore we reject H_{01}^{1} for the concurrent editing pattern. A file with concurrent edits is 1.8 times more likely to experience a future bug than a file that was never involved in a concurrent edit (Figure 9 (1)). We observed a statistically significant difference between the density of future bugs of files edited concurrently and others. The P-value of Wilcoxon rank sum test is $1.58e-03$; therefore, we reject H_{01}^{2} for the concurrent editing pattern. The density of future bugs of files involved in concurrent edits is 1.2 times greater than files that were never edited concurrently.The results of the impact of different levels of involvement in concurrent editing pattern are shown in Table VI. We could not reject either H_{01}^{3} or H_{01}^{4} for the concurrent editing pattern. The results from Table VI suggest that a low level of involvement in concurrent editing pattern is risky, while a high level of involvement is not risky. This is contrary to what we had hypothesized, the risk for bug is decreased when the number of developers involved in a concurrent editing is very high (i.e. above the third quartile). Nevertheless, this result is in line with Linus's law that "given enough eyeballs, all bugs are shallow." [23] Development teams can monitor only files that were edited concurrently by a small number of developers.

(2) Parallel editing pattern. Results from Table VI show that when developers follow the parallel editing pattern, the risk for bugs is not necessary increased. The P-values of the Fisher's exact test and the Wilcoxon rank sum test are respectively $2.15e-01$ and $1.97e-01$ when the number of files edited in parallel (i.e. the level) is below the third quartile (i.e. 14.33 files). Therefore, we cannot reject H_{01}^{1} and H_{01}^{2} for the parallel editing pattern in general. We attribute this result to the experience of the developers (of our studied projects) with parallel editing, as it is a common practice that developers change more than one file at the same time (e.g. [24]). In fact, Table V shows that 90% of files were involved in a parallel editing pattern in all the three studied projects. Because of such high frequency of the parallel editing pattern, the developers of these projects are probably used to parallel editing and likely more cautious. Therefore, the parallel editing pattern might be not enough by itself to measure the risk of introducing bugs.

However, from Table VI, it appears that the level of involvement of a file in a parallel editing plays a significant role in increasing the risk for bugs. The risk for bug and the density of future bugs is increased significantly when the number of files edited in parallel is above the third quartile (i.e. 14.33 files). We conclude that although developers can manage to edit a certain number of files at the same time, editing a large number of files in parallel is still risky. Therefore, we suggest that development teams pay more attention to files that are edited in parallel with too many other files. Quality assurance teams should advise developers against editing too many files in parallel.

(3) Extended editing pattern. The extended editing pattern increases the risk of bugs in files. Indeed, files with edit time spanning greater than the third quartile (i.e. 123 h) are 1.9 times more likely to experience a future bug than other files, as illustrated in Figure 9 (3). The Fisher's exact test was statistically significant (P-value = $6.63e-04$). We reject H_{01}^{1} for the extended editing pattern. The Wilcoxon rank sum test was also statistically significant for H_{01}^{2} (P-value = $5.18e-04$); therefore, we reject H_{01}^{2} for the extended editing pattern. Files edited following the extended editing pattern have on average 1.2 times greater density of future bugs than files that were never

Table VI. Relation between the level of involvement in a pattern, the risk of bugs, and the corresponding P-value.

	Level ≤ third quartile		Level > third quartile	
Pattern	*OR*	Average bug density	*OR*	Average bug density
Concurrent	2.0* (1.07e − 03)	**1.4*** (6.41e − 04)	1.4 (3.07e − 01)	0.8 (3.47e − 01)
Parallel	1.1 (8.74e − 01)	1.2 (7.88e − 01)	**2.9*** (8.73e − 04)	**2.1*** (1.27e − 03)
Extended	1.0 (8.31e − 01)	0.9 (8.40e − 01)	**1.9*** (1.55e − 03)	**1.1*** (1.36e − 03)
Interrupted	0.9 (6.53e − 01)	0.6 (5.65e − 01)	**2.0*** (4.95e − 04)	**0.9*** (5.64e − 04)

*indicates that the test was statistically significant, i.e. P-value < 0.05.

DOI: 10.1002/smr

involved in an extended editing. Although files edited following the extended editing pattern are at risk for bugs in general, all extended editings are not equally risky. Results from Table VI suggest that only files edited for a long time (i.e. above the third quartile) are very likely to exhibit a future bug. Extended editings might be the fact of complex tasks or change requests that are not clearly described. Developers should investigate the root cause of long editings that occur on files involved in extended editing patterns.

(4) Interrupted editing pattern. The occurrence of interrupted editing patterns does not necessarily increase the risk of bugs in files. Indeed, the *P*-value of the Fisher's exact test is statistically significant (i.e. $6.73e-05$) only when the interruption time (i.e. the level) is above the third quartile (i.e. 816 h). Therefore, we reject H_{01}^{1} for the interrupted editing pattern where the level of involvement of the file is above the third quartile. One possible explanation for this result is that it might become more difficult for developers to recall part of the code changes that were performed over 1 month ago (816 h are approximately 34 days). We also reject H_{01}^{2} for the interrupted editing pattern because the Wilcoxon rank sum test was statistically significant (*P*-value = $6.12e-05$). The files that were edited with interruption time greater than the third quartile have 1.1 times higher density of future bugs than other files that were not involved in an interrupted editing. The results for different levels of involvement in the interrupted editing pattern are shown in Table VI. Managers should consider reassigning change requests to other developers quickly, if the developers in charge of these change requests stop working on them or pause for a long time.

5.3. RQ2: Which factors contribute to the occurrence of file editing patterns?

Motivation

In Section 5.1, we observed that 90% (i.e. 1935 out of 2140) of files in our data set were edited following at least one of the four file editing patterns. However, we also observed that the four file editing patterns were not followed with the same frequency. For example, 90% (i.e. 1922 out of 2140) of files were edited in parallel with other files while only 23% (i.e. 497 out of 2140) of files were edited concurrently by multiple developers. Because the risks of bugs of files edited following different editing patterns are not equal, it is important to understand the factors contributing to the occurrence of a file editing pattern, so that development teams can act to monitor some file editing patterns. This research question aims to examine if the occurrences of file editing patterns are impacted by the following three factors: (1) the ownership of files; (2) the type of change requests; and (3) the initial code quality of the files. We describe our approaches and findings for each factor as follows.

(1) The ownership of files

Approach

We use the two metrics *Ownership* and *hasMajorOwner* that are defined in Section 4.6 to measure the ownership of a particular file. First, we investigate whether the level of the highest ownership of each file impacts the occurrence of file editing patterns. The quartiles of the highest *Ownership* of files in our data set are the following: 0 (0%), 0.89 (25%), 1.00 (50%), 1.00 (75%), and 1.00 (100%). Therefore, we divide files into two groups using the second quartile (i.e. 0.89). We also investigate whether files with a major owner are more or less likely to exhibit the file editing patterns. We divide files into two groups:

(1) G_{noneMO} containing files without major owners (i.e. *hasMajorOwner* equals to *false*); and
(2) G_{hasMO} containing files with major owners (i.e. *hasMajorOwner* equals to *true*).

For each file editing pattern P_i, we test the following null hypothesis:

H_{02}^{1}: The proportion of files exhibiting file editing pattern P_i does not differ between the groups divided by the metric *Ownership*.

H_{02}^{2}: The proportion of files exhibiting file editing pattern P_i does not differ between the groups G_{noneMO} and G_{hasMO}.

DOI: 10.1002/smr

Hypothesis H_{02}^1 is about the probability of exhibiting a file editing pattern when considering the level of highest ownership of files. H_{02}^2 is about the probability of exhibiting file editing patterns for files without or with major owners.

We use the Fisher's exact test to accept or reject the hypothesis. If there is a statistically significant difference, we reject the hypothesis and further compute ORs to determine how much the ownership of files affects the occurrence of file editing patterns.

Findings

The ORs and corresponding *P*-values are presented in Table VII. Four Fisher's exact tests were performed to test hypothesis H_{02}^1 and H_{02}^2, respectively. We apply the Holm–Bonferroni method to correct the threshold *P*-value to 0.05 and 0.05, respectively. The results show that files with strong ownership are less likely to exhibit concurrent, extended, and interrupted editing patterns. Files with a major owner are less likely to experience all four editing patterns. One possible reason is that change request tasks related to a particular file are often assigned to the major owner of the file. For instance, in our pattern analysis period, the file *AbstractRepositoryQueryPage.java* of Mylyn project is modified to address ten change requests. Six of them are assigned to the major owner (i.e. Steffen Pingel) of the file. Also, the major owner is involved in addressing three other change requests, and he opens the remaining one. As a summary, this result suggests that quality assurance teams should pay attention to files with weaker ownership (i.e. the highest ownership value is below 0.89, the second quartile) and files without a major owner.

(2) The type of change requests

Approach

As described in Section 4.7, we have three types of change requests: enhancements, minor bug, and major bug. We use the three metrics ENHANCEMENT, MINORBUG, and MAJORBUG to measure the involvement of files in different types of change requests. We first investigate whether file editing patterns are more likely to occur in files involved in more enhancement implementations or bug fixes. We categorize files into two groups:

(1) $G_{enhancement}$ containing files with more enhancement implementations than bug fixes; and
(2) G_{bug} containing files implementing more bug fixes than enhancement implementations.

We also examine whether file editing patterns are more likely to occur in files involved in more minor bug fixes or in more major bug fixes. We split files into two groups:

(1) $G_{minorBug}$ containing files with more minor bugs fixes than major bugs fixes; and
(2) $G_{majorBug}$ containing files with more major bugs fixes than minor bugs fixes.

For each file editing pattern P_i, we test the following null hypothesis:

H_{02}^3: The proportion of files exhibiting the file editing pattern P_i does not differ between the groups $G_{enhancement}$ and G_{bug}.
H_{02}^4: The proportion of files exhibiting the file editing pattern P_i does not differ between the groups $G_{minorBug}$ and $G_{majorBug}$.

Hypothesis H_{02}^3 is about the probability of exhibiting file editing patterns for files edited more for feature enhancements or more for fixing bugs. H_{02}^4 is about the probability of exhibiting file editing patterns for files edited mainly to fix minor or major bugs.

Table VII. Odds ratios obtained when examining the impact of the ownership of files on the probability of the occurrences of file editing patterns and the corresponding *P*-value of the Fisher's exact tests.

Ownership of files	Concurrent	Parallel	Extended	Interrupted
Highest *Ownership*	0.5* (6.86e − 08)	0.8 (2.85e − 01)	**0.5*** (2.79e − 10)	**0.6*** (2.96e − 06)
hasMajorOwner	**0.6*** (4.06e − 08)	**0.7*** (1.92e − 02)	**0.5*** (6.91e − 11)	**0.5*** (2.09e − 09)

*indicates that the test was statistically significant, i.e. *P*-value < 0.05.

DOI: 10.1002/smr

Table VIII. Odds ratios obtained when examining the impact of the type of change requests on the probability of the occurrences of file editing patterns and the corresponding *P*-value of Fisher's exact tests.

Type of change requests	Concurrent	Parallel	Extended	Interrupted
Bug versus enhancement	**2.6*** (1.08e − 06)	**2.0*** (9.71e − 04)	**2.0*** (2.93e − 04)	**1.5*** (1.46e − 02)
Minor versus major bug	**4.8*** (3.53e − 04)	1.7 (1.59e − 01)	1.1 (1.00e + 00)	1.1 (1.00e + 00)

*indicates that the test was statistically significant.

We apply the Fisher's exact test to examine the two hypotheses. If there is a statistically significant difference, we further compute ORs to determine how much the type of change requests and the level of severity of bugs affect the occurrence of file editing patterns.

Findings

The ORs and the corresponding *P*-values are presented in Table VIII. Four Fisher's exact tests were performed to test hypothesis H_{02}^3 and H_{02}^4, respectively. The threshold *P*-values are corrected to 0.05, and $1.67e-02$, respectively.

Results show that files with more bug fixes than enhancement implementations are more likely to exhibit the four file editing patterns. This might be because developers are more cautious when solving complicated problems. In the three subject projects, the median (i.e. 4) of the number of files involved in implementing enhancements is twice the median (i.e. 2) of the number of files involved in fixing bugs, indicating that the enhancement implementions are often more complicated than bug fixings. For example, seven files were edited when implementing the enhancement #300078 of Mylyn project. Five files were modified when fixing the bug #300229 of Eclipse Platform project.

The results also show that files with more fixes for minor bugs than major bugs are more likely to exhibit concurrent pattern. One possible explanation is that developers might be more cautious when fixing bugs with higher severity than lower severity.

(3) The initial code quality of files

Approach

We use the metrics described in Section 4.8 to measure the initial code quality of files. The files are divided into two groups based on each metric, separately:

(1) files with the metric value below or at the median; and
(2) files with the metric value above the median.

For each file editing pattern P_i, we test the following null hypothesis:

H_{02}^5: The proportion of files exhibiting file editing pattern P_i does not differ between the groups divided by each of the aforementioned metrics.

Hypothesis H_{02}^5 is about the probability of exhibiting file editing patterns in files with smaller or larger metric values (on the initial snapshot just before the period of identification of the file editing patterns). The hypothesis is two-tailed because it investigates whether the initial code quality of a file can increase or decrease the likelihood of a file editing pattern occurring.

We apply the Fisher's exact test to test the hypothesis. If there is a statistically significant difference, we reject the hypothesis and further compute ORs to determine how much the initial code quality of files affects the occurrence of file editing patterns.

Findings

The ORs and the corresponding *P*-values are presented in Table IX. In total, we perform 140 (i.e. 35×4) Fisher's exact tests. We apply the Holm–Bonferroni method to correct the threshold of *P*-value to $6.33e-04$. The results show that the extended and interrupted editing patterns are more highly impacted by the initial code quality in general. The detailed impacts

DOI: 10.1002/smr

Table IX. Odds ratios obtained when examining the impact of the initial code quality on the probability of the occurrences of file editing patterns and corresponding *P*-value of Fisher's exact tests.

Category	Code metric	Concurrent	Parallel	Extended	Interrupted
Complexity	CLOC	1.3 (2.26e − 02)	1.5 (2.16e − 02)	**2.5*** (1.2e − 13)	**3.6*** (1.30e − 24)
	avgNIM	1.1 (4.36e − 01)	1.2 (3.78e − 01)	1.2 (1.37e − 01)	1.4 (7.40e − 03)
	maxNIM	1.1 (2.88e − 01)	1.5 (2.67e − 02)	**1.9*** (4.65e − 08)	**2.9*** (3.27e − 18)
	sumNIM	1.0 (8.67e − 01)	1.4 (9.42e − 02)	**1.8*** (5.50e − 07)	**3.1*** (1.06e − 19)
	avgNIV	1.1 (2.35e − 01)	1.2 (4.21e − 01)	1.4 (4.50e − 03)	**1.9*** (1.39e − 07)
	maxNIV	1.1 (4.99e − 01)	1.3 (1.29e − 01)	**1.7*** (1.06e − 05)	**2.5*** (5.40e − 14)
	sumNIV	1.1 (5.38e − 01)	1.4 (7.63e − 02)	**1.7*** (2.69e − 05)	**2.5*** (1.20e − 13)
	avgWMC	1.1 (6.17e − 01)	1.2 (2.19e − 01)	1.2 (1.54e − 01)	1.4 (4.30e − 03)
	maxWMC	1.1 (3.44e − 01)	1.6 (8.12e − 03)	**2.0*** (1.43e − 08)	**3.1*** (5.91e − 20)
	sumWMC	1.0 (7.81e − 01)	1.5 (4.23e − 02)	**1.8*** (1.38e − 06)	**3.3*** (1.19e − 21)
	avgCC	1.3 (4.56e − 03)	0.9 (4.29e − 01)	**2.1*** (5.28e − 10)	**1.9*** (1.54e − 07)
	maxCC	**1.5*** (2.73e − 04)	1.1 (7.91e − 01)	**3.0*** (2.22e − 19)	**2.6*** (2.85e − 15)
	sumCC	1.2 (1.48e − 01)	1.5 (3.40e − 02)	**2.4*** (1.55e − 12)	**3.4*** (1.34e − 23)
Coupling	avgCBO	**2.4*** (2.85e − 15)	1.7 (3.54e − 03)	**1.8*** (7.36e − 07)	1.3 (2.02e − 02)
	maxCBO	**2.3*** (6.21e − 14)	**2.5*** (6.14e − 07)	**2.5*** (4.08e − 14)	**2.1*** (1.62e − 09)
	sumCBO	**2.0*** (2.67e − 10)	**2.8*** (7.77e − 08)	**2.5*** (4.17e − 14)	**2.6*** (6.26e − 15)
	avgRFC	1.3 (2.61e − 02)	1.7 (2.65e − 03)	1.2 (6.53e − 02)	**1.8*** (2.50e − 06)
	maxRFC	1.2 (5.13e − 02)	1.7 (6.05e − 03)	**1.6*** (6.53e − 05)	**2.9*** (4.78e − 18)
	sumRFC	1.0 (7.38e − 01)	1.6 (6.25e − 03)	**1.6*** (3.03e − 05)	**3.0*** (9.54e − 19)
Cohesion	avgLCOM	1.3 (1.67e − 02)	1.0 (1.00e + 00)	1.146 (2.59e − 01)	0.9 (5.52e − 01)
	maxLCOM	**1.6*** (2.32e − 05)	1.3 (1.34e − 01)	**2.4*** (6.69e − 13)	**2.5*** (4.35e − 14)
	sumLCOM	**1.5*** (1.23e − 04)	1.3 (1.12e − 01)	**2.2*** (4.94e − 11)	**2.4*** (2.89e − 13)
Abstraction	avgDIT	1.2 (1.79e − 01)	1.2 (2.49e − 01)	**0.6*** (1.23e − 04)	0.8 (1.51e − 01)
	maxDIT	1.4 (2.84e − 02)	2.0 (9.95e − 03)	1.0 (9.40e − 01)	**1.9*** (1.52e − 05)
	sumDIT	0.9 (5.04e − 01)	1.7 (4.59e − 03)	1.4 (1.24e − 02)	**2.2*** (2.94e − 11)
	avgIFANIN	1.2 (1.33e − 01)	1.1 (7.25e − 01)	**1.7*** (3.06e − 05)	**2.3*** (9.46e − 12)
	maxIFANIN	0.7 (2.51e − 01)	0.9 (7.14e − 01)	1.5 (8.43e − 02)	2.1 (7.90e − 04)
	sumIFANIN	1.0 (8.21e − 01)	1.3 (1.52e − 01)	**1.6*** (8.31e − 05)	**2.6*** (1.35e − 15)
Encapsulation	avgNOC	1.4 (1.75e − 02)	1.8 (5.14e − 02)	1.7 (1.35e − 03)	**2.2*** (4.53e − 07)
	maxNOC	1.4 (1.75e − 02)	1.8 (5.14e − 02)	1.7 (1.35e − 03)	**2.2*** (4.53e − 07)
	sumNOC	1.4 (1.75e − 02)	1.8 (5.14e − 02)	1.7 (1.35e − 03)	**2.2*** (4.53e − 07)
	avgRPM	**1.5*** (4.53e − 04)	1.2 (3.32e − 01)	1.1 (5.93e − 01)	1.0 (7.21e − 01)
	sumRPM	0.9 (3.66e − 01)	1.4 (8.88e − 02)	1.5 (1.37e − 03)	**2.4*** (4.60e − 13)
Documentation	CL	0.9 (4.36e − 01)	1.0 (9.30e − 01)	**2.1*** (2.27e − 09)	**2.5*** (1.09e − 13)
	RCC	**0.5*** (8.72e − 08)	**0.4*** (1.48e − 06)	**0.5*** (3.94e − 08)	**0.5*** (9.78e − 09)

*indicates that the test was statistically significant, i.e. $P\text{-value} < 6.33e-04$.

of each category of metrics are described as follows. Higher complexity (e.g. greater values in CLOC, maxNIM, maxWMC, and CC) increases the likelihood of files to experience extended and interrupted editing patterns. In addition, the metric maxCC has a relationship with the occurrence of concurrent editing pattern. Stronger coupling between objects (i.e. greater values in maxCBO and sumCBO) contributes significantly to the occurrence of all the four file editing patterns. The coupling metric RFC only impacts the occurrence of the interrupted editing pattern. A weak cohesion (i.e. high values in maxLCOM and sumLCOM) increases the likelihood of concurrent, extended, and interrupted editing patterns. The abstraction mainly affects the occurrence of the interrupted editing pattern. The metrics avgRPM and sumRPM have a relationship with the occurrence of concurrent and interrupted editing patterns, respectively. Increasing the ratio of comments to code (i.e. RCC) can significantly reduce the likelihood of files to exhibit all the four file editing patterns. There are six metrics playing a significant role in the occurrence of at least three file editing patterns. These six metrics are the following: maxCC, maxCBO, sumCBO, maxLCOM, sumLCOM, and RCC. Among these metrics, the metric RCC is the easiest one to control. Developers should add more comments to help reduce the likelihood of experiencing the four file editing patterns in the future. Moreover, we suggest that development teams monitor and control the aforementioned six quality indicators investigated in this work, in order to reduce the occurrence of the editing patterns and hence the risk that they may induce.

DOI: 10.1002/smr

Overall, we observe that the ownership of files and the severity of bug fix change requests affect the occurrence of the four editing patterns investigated in this work. The initial code quality of files, in particular the aspects of quality measured by maxCC, maxCBO, sumCBO, maxLCOM, sumLCOM, and RCC, contribute significantly to the occurrence of at least three out of the four file editing patterns.

5.4. RQ3: Which factors affect the bug proneness of files edited following the patterns?

Motivation

In Section 5.2, we observe that all four file editing patterns increase the likelihood of future bugs. In Section 5.3, we observe that there exists a relationship between the occurrence of file editing patterns and the ownership of files, the type of change requests, and the initial code quality. Hence, it is interesting to investigate the combined effect that these three factors (i.e. ownership of files, the type of change requests, and the initial code quality) and the file editing patterns have on the likelihood of future bugs. This research question aims to examine such combined effects. In particular, we want to understand if the ownership of a file impacts the risk for future bugs in this file when it is edited following a certain editing pattern. We also want to examine whether the types of the change requests in which a file was involved do impact the risk for future bugs in the file when it is edited following a certain editing pattern. For the initial code quality, we only investigate the aforementioned six metrics (i.e. maxCC, maxCBO, sumCBO, maxLCOM, sumLCOM, and RCC) that play a significant role in the occurrence of at least three file editing patterns, individually.

Approach

In the approach of RQ2 (refer to Section 5.3), files are split into two groups using each metric defined in Section 4.6, Section 4.7, and Section 4.8. To address this research question, we further divide the two groups into four subgroups by considering whether a particular file editing pattern had occurred or not. The first group is always selected as the control group, and the other three groups are experimental groups. We test the following hypothesis for each pattern P_i:

H_{03}^1: The proportion of files exhibiting at least one future bug does not differ between the control group and experimental groups.

Hypothesis H_{03}^1 is about the probability of bugs in files with different values of the corresponding metric and edited following the pattern P_i. H_{03}^1 is two-tailed because it investigates whether the combined effect of file editing pattern P_i and the corresponding metric is related to a higher or a lower risk for bug. We use the Fisher's exact test and compute the OR to test H_{03}^1. We describe our detailed groups and findings for each factor as follows.

(1) The ownership of files

Groups

For each pattern P_i, the files are divided by the proportion of the highest ownership as follows.

- G_{ow1}: files not exhibiting the pattern P_i and where the value of the highest *Ownership* is below or at the second quartile;
- G_{ow2}: files not exhibiting the pattern P_i and where the value of the highest *Ownership* is above the second quartile;
- G_{ow3}: files exhibiting the pattern P_i and where the value of the highest *Ownership* is below or at the second quartile;
- G_{ow4}: files exhibiting the pattern P_i and where the value of the highest *Ownership* is above the second quartile.

For each pattern P_i, the files are divided by the presence of major owners as follows.

- G_{mo1}: files not exhibiting pattern P_i and without major owners;
- G_{mo2}: files not exhibiting pattern P_i but with major owners;
- G_{mo3}: files exhibiting pattern P_i and without major owners;

DOI: 10.1002/smr

- G_{mo4}: files exhibiting pattern P_i and with major owners.

Findings

We present ORs and the corresponding *P*-values in Table X. When testing hypothesis H_{03}^1 for metric *Ownership* (respectively *hasMajorOwner*), we perform 12 (respectively 12) Fisher's exact tests. We apply Holm–Bonferroni method to correct the threshold of *P*-value to $6.25e-03$ (respectively 0.01). The results show that files with stronger ownership are less likely to experience future bugs when edited following concurrent, extended, and interrupted editing patterns. A strong ownership on a file can alleviate the impact of risky editing patterns, in particular the concurrent editing pattern. When developers edit files with weak ownership (i.e. without major owners) concurrently, the likelihood of future bugs in the files significantly increases to 2.2 times the amount of bugs in files with strong ownership and edited without following the concurrent editing pattern. The results also show that files without a major owner are more likely to experience a future bug if they are edited following concurrent and extended editing patterns. However, the presence of major owners can significantly reduce the risk posed by concurrent, extended, and interrupted editing patterns. We suggest that development teams organize their tasks in a way that favors the owning of files by developers; hence, the risk incurred by file editing patterns can be reduced.

(2) The type of change requests

Groups

For each pattern P_i, the files are divided by the type of change requests as follows.

- G_{ct1}: files not exhibiting the pattern P_i and having more enhancements implemented than bugs fixed;
- G_{ct2}: files not exhibiting the pattern P_i and implementing more bug fixes than enhancements;
- G_{ct3}: files exhibiting the pattern P_i and having more enhancements implemented than bugs fixed;
- G_{ct4}: files exhibiting the pattern P_i and implementing more bug fixes than enhancements.

For each pattern P_i, the files are divided by the level of severity of change requests as follows.

- G_{sl1}: files not exhibiting the pattern P_i and having more major bugs fixed than minor bugs;
- G_{sl2}: files not exhibiting the pattern P_i and having more minor bugs fixed than major bugs;
- G_{sl3}: files exhibiting the pattern P_i and having more major bugs fixed than minor bugs;
- G_{sl4}: files exhibiting the pattern P_i and having more minor bugs fixed than major bugs.

Findings

We present ORs and the corresponding *P*-values in Table XI. When testing hypothesis H_{03}^1 for the type of change requests (respectively the level of severity of bugs), we perform 12 (respectively 12) Fisher's exact tests. We apply Holm–Bonferroni method to correct the threshold *P*-value to $5.56e-03$ (respectively $4.17e-03$). The results show that files with more bug fixes (compared with enhancement implementations) in their history and edited following the concurrent editing pattern are 3.9 times more

Table X. Odds ratios obtained when examining the combined impact of file editing patterns and ownership of files on the probability of future bugs and *P*-value of Fisher's exact tests.

Ownership of files	Group	Concurrent	Parallel	Extended	Interrupted
Ownership	G_{ow1}		Control group		
	G_{ow2}	**0.5*** (4.39e – 03)	0.3 (6.73e – 02)	**0.5*** (2.01e – 03)	**0.4*** (4.96e – 04)
	G_{ow3}	**2.2*** (3.19e – 03)	1.3 (8.20e – 01)	2.0 (9.74e – 03)	2.0 (8.58e – 03)
	G_{ow4}	0.6 (8.77e – 02)	0.5 (1.92e – 01)	0.7 (1.67e – 01)	0.8 (3.60e – 01)
hasMajorOwner	G_{mo1}		Control group		
	G_{mo2}	**0.5*** (3.67e – 03)	0.3 (7.18e – 02)	**0.5*** (8.33e – 04)	**0.5*** (4.57e – 04)
	G_{mo3}	**2.4*** (1.78e – 04)	1.2 (8.34e – 01)	**2.1*** (1.53e – 03)	**2.2*** (9.26e – 04)
	G_{mo4}	**0.4*** (5.13e – 03)	0.4 (1.17e – 01)	0.5 (2.46e – 02)	0.6 (9.96e – 02)

G_{mo1} and G_{ow1} are the control groups.
*indicates that the test was statistically significant.

DOI: 10.1002/smr

Table XI. Odds ratios obtained when examining the combined impact of file editing patterns and the type of change requests on the probability of future bugs and p -value of Fisher's exact tests.

Type of change requests	Group	Concurrent	Parallel	Extended	Interrupted
Bug versus enhancement	G_{ct1}	Control group			
	G_{ct2}	2.3 (5.45e − 02)	2.0 (6.90e − 01)	3.3 (1.30e − 02)	2.3 (6.14e − 02)
	G_{ct3}	4.2 (7.36e − 02)	1.6 (1.00e + 00)	7.7 (5.72e − 03)	3.6 (7.23e − 02)
	G_{ct4}	**3.9*** (5.72e − 04)	3.0 (3.57e − 01)	**5.5*** (1.70e − 04)	**4.7*** (1.93e − 04)
Minor bug versus major bug	G_{sl1}	Control group			
	G_{sl2}	0.9 (7.79e − 01)	Inf (1.00e + 00)	1.5 (1.00e + 00)	1.4 (1.00e + 00)
	G_{sl3}	0.0 (1.00e + 00)	Inf (1.00e + 00)	3.2 (2.73e − 01)	3.2 (2.73e − 01)
	G_{sl4}	1.6 (4.93e − 01)	Inf (1.00e + 00)	2.6 (2.97e − 01)	2.8 (2.08e − 01)

G_{ct1} and G_{sl1} are the control groups.
*indicates that the test was statistically significant.

likely to experience future bugs than files with more enhancement implementations (compared with bug fixes) in their history and edited without following the concurrent editing pattern. For the files edited following the extended (respectively interrupted) editing pattern, files with more bug fixes (compared with enhancement implementations) in their history are 5.5 (respectively 4.7) times more likely to experience future bugs than other files with more enhancement implementations (compared with bug fixes) in their history. This finding is in line with previous work from the literature (e.g. [25]) that reported that files with past bugs are likely to experience future bugs. The level of severity of the fixed bugs has no impact on the occurrence of future bugs. Developers should be cautious when modifying files with more bug fixes than enhancement implementations, following concurrent, extended, or interrupted editing patterns.

(3) The initial code quality

Groups
For each pattern P_i, we divide the files using maxCC metric values as follows.

- G_{ccq1m}: files not exhibiting the pattern P_i and having a value of maxCC below or at the median of all maxCC values;
- G_{ccq2m}: files not exhibiting the pattern P_i and having a value of maxCC above the median of all maxCC values;
- G_{ccq3m}: files exhibiting the pattern P_i and having a value of maxCC below or at the median of all maxCC values;
- G_{ccq4m}: files exhibiting the pattern P_i and having a value of maxCC above the median of all maxCC values.

Similarly, we obtain the groups G_{cboq1m}, G_{cboq2m}, G_{cboq3m}, and G_{cboq4m} for the metric maxCBO, the groups G_{cboq1s}, G_{cboq2s}, G_{cboq3s}, and G_{cboq4s} for the metric sumCBO, the groups $G_{lcomq1m}$, $G_{lcomq2m}$, $G_{lcomq3m}$, and $G_{lcomq4m}$ for the metric maxLCOM, the groups $G_{lcomq1s}$, $G_{lcomq2s}$, $G_{lcomq3s}$, and $G_{lcomq4s}$ for the metric sumLCOM, and the groups G_{rccq1}, G_{rccq2}, G_{rccq3}, and G_{rccq4} for the metric RCC.

Findings
We present ORs and the corresponding P-values in Table XII. When testing hypothesis H_{03}^{1}, we perform 60 Fisher's exact tests in total. We apply Holm–Bonferroni method to correct the threshold of P-value to 1.43e−03. The findings on metrics maxCC, maxCBO, sumCBO, maxLCOM, and sumLCOM are very similar. That is, higher values of these metrics significantly increase the likelihood of the files to exhibit a bug in the future. In addition, when files are edited following the four file editing patterns, the risk for future bugs increases. For example, when a file with a high complexity (i.e. maxCC value above the median) is edited following the concurrent editing pattern, its likelihood to exhibit a bug increases from 2.7 to 5.8 times the likelihood of bugs for files that are edited without following the concurrent editing pattern. The findings for the RCC metric indicate that

DOI: 10.1002/smr

Table XII. Odds ratios obtained when examining the combined impact of file editing patterns and the initial code quality on the probability of future bugs and *P*-value of Fisher's exact tests.

Code metric	Group	Concurrent	Parallel	Extended	Interrupted
maxCC	G_{ccq1m}		Control group		
	G_{ccq2m}	**3.6*** (3.54e − 09)	4.1 (5.45e − 02)	**4.9*** (1.92e − 12)	**4.1*** (5.96e − 10)
	G_{ccq3m}	1.2 (6.58e − 01)	1.4 (7.89e − 01)	2.7 (8.99e − 03)	2.0 (1.07e − 01)
	G_{ccq4m}	**5.8*** (2.06e − 12)	**5.6*** (3.60e − 04)	**6.1*** (3.54e − 12)	**6.4*** (7.23e − 14)
maxCBO	G_{cboq1m}		Control group		
	G_{cboq2m}	**2.2*** (1.69e − 04)	3.0 (8.91e − 02)	**2.6*** (1.04e − 05)	**2.3*** (8.52e − 05)
	G_{cboq3m}	0.6 (3.19e − 01)	1.2 (8.24e − 01)	1.1 (6.95e − 01)	1.0 (1.00e + 00)
	G_{cboq4m}	**4.3*** (1.96e − 10)	**3.7*** (1.27e − 03)	**4.3*** (4.04e − 10)	**4.6*** (1.64e − 11)
sumCBO	G_{cboq1s}		Control group		
	G_{cboq2s}	**2.8*** (1.38e − 06)	4.2 (3.83e − 02)	**3.4*** (1.73e − 08)	**2.8*** (2.33e − 06)
	G_{cboq3s}	0.8 (8.34e − 01)	1.4 (8.11e − 01)	1.7 (1.52e − 01)	1.0 (1.00e + 00)
	G_{cboq4s}	**5.0*** (1.97e − 11)	**4.7*** (3.93e − 04)	**5.2*** (1.94e − 11)	**5.1*** (2.33e − 12)
maxLCOM	$G_{lcomq1m}$		Control group		
	$G_{lcomq2m}$	**2.6*** (4.87e − 06)	3.9 (5.65e − 02)	**3.5*** (1.09e − 08)	**2.8*** (4.77e − 06)
	$G_{lcomq3m}$	1.2 (6.73e − 01)	1.7 (6.17e − 01)	2.7 (5.01e − 03)	1.7 (1.91e − 01)
	$G_{lcomq4m}$	**4.3*** (2.90e − 09)	**5.0*** (1.25e − 03)	**4.5*** (3.74e − 09)	**4.5*** (1.37e − 10)
sumLCOM	$G_{lcomq1s}$		Control group		
	$G_{lcomq2s}$	**2.3*** (8.03e − 05)	4.0 (5.54e − 02)	**3.0*** (2.60e − 07)	**2.3*** (1.01e − 04)
	$G_{lcomq3s}$	1.1 (8.41e − 01)	1.8 (4.70e − 01)	2.4 (1.24e − 02)	1.3 (4.07e − 01)
	$G_{lcomq4s}$	**4.0*** (6.02e − 09)	**4.9*** (1.24e − 03)	**4.1*** (1.98e − 08)	**4.1*** (5.25e − 10)
RCC	G_{rccq1}		Control group		
	G_{rccq2}	1.0 (9.19e − 01)	0.6 (5.18e − 01)	0.8 (2.09e − 01)	0.8 (3.95e − 01)
	G_{rccq3}	**2.7*** (1.12e − 05)	1.4 (6.72e − 01)	2.0 (1.62e − 03)	**2.4*** (1.04e − 04)
	G_{rccq4}	0.5 (1.16e − 01)	0.9 (7.96e − 01)	1.0 (1.00e + 00)	1.0 (1.00e + 00)

*indicates that the test was statistically significant. i.e. $P\text{-value} < 1.43 - 03$.

higher ratio of comments to code can alleviate the risk incurred by the concurrent editing pattern. The aforementioned finding reinforces our recommendation that development teams should monitor and control for quality indicators such as size, complexity, and coupling, and developers should add more comments to their code.

> Overall, we find that a strong ownership of files can reduce the negative impact of the four file editing patterns. In the presence of concurrent, extended, or interrupted editing patterns, the likelihood of future bugs in a file that underwent more bug fixes than enhancement implementations is increased. In the presence of any of the four editing patterns, the likelihood of future bugs in a file with high complexity, strong coupling between objects, or less cohesion is increased.

5.5. *RQ4: Do interactions among file editing patterns lead to more bugs?*

Motivation

In Section 5.1, we find that 44% of files (i.e., 945 out of 2140 files) from our subject projects are edited following more than one editing pattern. When multiple editing patterns are followed by developers during the modification of a file, the risk of introducing a bug can be increased. For example, if a developer editing multiple files simultaneously (i.e. the parallel editing pattern) is interrupted frequently (the interrupted editing pattern), the developer might become confused and cause errors in the files. In this research question, we investigate the interaction between the four file editing patterns. We want to understand if the risk of bugs in a file is increased when multiple editing patterns are followed by developers during the modifications of the file. Similar to RQ1 (refer to Section 5.2), developers and managers can use the knowledge of pattern interactions to decide on the acquisition of awareness tools that can warn developers about pattern interactions during development and maintenance activities.

DOI: 10.1002/smr

Approach

For each file, we use the metrics described in Section 4.5 to identify the editing patterns of the file. We classify the files based on the pattern(s) followed by developers during the modifications of the files. For each pattern P_i, we create a group GP_i containing files that were edited by developers following P_i. For each combination of pattern(s) $PInteract_i$ listed in Table V, we create a group $GPInteract_i$ containing files that were edited by developers following the patterns in $PInteract_i$. We also create a group *GNoP* containing files that were edited by developers following none of the four patterns. We compute the density of future bugs in each file and test the two following null hypothesis.

H_{04}^1: The proportion of files exhibiting at least one future bug does not differ between the groups GP_i, $GPInteract_i$, and *GNoP*.

H_{04}^2: There is no difference between the density of future bugs of files from groups GP_i, $GPInteract_i$, and *GNoP*.

Similar to RQ1, hypothesis H_{04}^1 (respectively H_{04}^2) is about the probability of bugs (respectively the density of future bugs). The two hypothesis are two-tailed. We use the Fisher's exact test and compute the OR to test H_{04}^1. We perform a Kruskal–Wallis rank sum test for H_{04}^2. We test H_{04}^1 and H_{04}^2 using the 95% confidence level (i.e. P-value <0.05).

Findings

The Fisher's exact test was statistically significant; therefore, we reject H_{04}^1. The Kruskal–Wallis rank sum test for H_{04}^2 was also statistically significant. We also reject H_{04}^2. The risk of future bugs in a file edited following more than one editing pattern is higher than the risk of future bugs in a file edited following a single editing pattern. As illustrated in Figure 10, whenever a file displayed the concurrent, extended, and interrupted editing patterns all together, the likelihood of future bugs becomes the highest (i.e. 3.9). The density of future bugs of these files is also the highest (i.e. 2.1 times greater than files edited but never following any of the four patterns). Moreover, when either the concurrent editing pattern or the extended editing pattern are used with other patterns during the modification of a file, the risk of future bugs in the file is often increased (i.e. the OR is increased). Developers should pay more attention to files edited following multiple editing patterns.

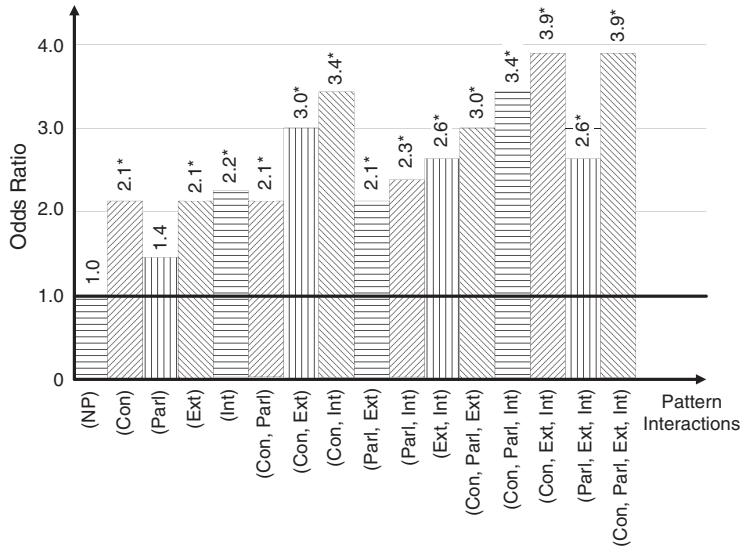

Figure 10. Odds ratios of future bugs in files from the 16 groups listed in Table V. The abbreviation NP stands for no file editing pattern. The abbreviations Con, Parl, Ext, Int denote concurrent, parallel, extended, and interrupted editing patterns, respectively.

DOI: 10.1002/smr

6. DISCUSSION

In this section, we discuss the impact that some of our design decisions can have on our results. We also discuss confounding factors.

6.1. Design decisions

This section discusses the design decisions that were made when performing the study presented in this paper.

6.1.1. Combination of data set. In this study, because of the small sizes of the data extracted from the three studied Eclipse projects, we made the decision to combine the data set of these three projects. To understand the impact of this decision on our results, we replicate the study on each project separately. The results of the replication are presented in Appendix B. The findings for the Mylyn project are consistent with those obtained using the combined data set. This result was kind of expected because 55% (i.e. 1177 out of 2140) of files in the combined data set belong to the Mylyn project. For the Eclipse Platform project, there are no significant relationships between the editing patterns and the occurrence of future bugs.

We attribute this result to the different degrees of the involvement of Mylyn tool among Mylyn, PDE, and Eclipse Platform projects. In fact, out of the 61 366 files in the Eclipse Platform project, only 738 (i.e. 1%) files were recorded in Mylyn logs. Mylyn logs captured only a small amount of developers' activities in the Eclipse Platform project. The ratios of recorded files for the Mylyn and the PDE project are respectively 59% and 70%. The extensive usages of Mylyn tools in these two projects resulted in more developers' activities being captured in Mylyn logs. Therefore, it is no surprise that the findings for the PDE project are also consistent with those of the combined data set, except for the relationship between the concurrent editing pattern and the occurrence of future bugs that is not statistically significant. One possible explanation is that only 8% of files are edited following the concurrent editing pattern in the PDE project (which is too small to achieve statistical significance), compared with 37% of files in the Mylyn project. In summary, the results of the replication show that our findings obtained using the combined data set are consistent with those obtained on projects where Mylyn tool was extensively used.

6.1.2. Strategy of grouping. In Section 5.2, we split the full set of files into two groups for each file editing pattern. The first group includes files edited without following pattern P_i, and the other group includes files edited following pattern P_i. The first group could contain files edited following the other three editing patterns. To examine whether this strategy of grouping impacts our findings, we divide the first group into two subgroups: (1) G_{NoneP} which contains files not exhibiting any of the four editing patterns; and (2) G_{NoneP_iOnly} which contains files not exhibiting pattern P_i but which were modified following some of the other remaining editing patterns. We obtained slightly different OR values but which are consistent with those from Section 5.2. Detailed results are presented in Appendix C. This result provides an extra validation of the conclusions reported in this paper.

6.1.3. Statistical method. Analysis of variance (ANOVA) can compare three or more groups for statistical significance; therefore, multiple factors can be compared together. However, one of the assumptions of ANOVA is that the variances are equal among different groups. We applied Levene's test [26] to assess the equality of variances. In general, we found that the *P*-values for most Levene's test are less than 0.05. There are significant differences between the variances of the different groups that are investigated in this study, making ANOVA unsuitable for our research questions. For these reasons, we chose to conduct Fisher's exact tests, ORs, and Wilcox rank sum tests to investigate the effect of the file editing patterns.

6.2. Confounding factors

Relative code churn

Nagappan and Ball [27] have found that the proportion of LOC changed in a file (i.e. relative code churn) also correlates highly with the density of future bugs. We investigate the potential confounding

DOI: 10.1002/smr

effect of relative code churns (RCHURN) on our findings by applying the similar approach as Section 5.4. We divide the files into two groups by the median of relative code churns. We compare the likelihood of future bugs between files with high relative code churn and files edited following any of the four file editing patterns and find that there is no significant difference. This might be because there are high correlations between files with high code churn and files involved in any of the four patterns. Hence, for each pattern P_i, we further divide the files into four non-overlapping groups: (1) files not exhibiting the pattern P_i with low relative code churn (i.e. less than or equal to the median); (2) files not exhibiting the pattern P_i with high relative code churn (i.e. above the median); (3) files exhibiting the pattern P_i with low relative code churn; and (4) files exhibiting the pattern P_i with high relative code churn. By comparing the likelihood of future bugs between the first two groups, we find that the relative code churn does increase the likelihood of future bugs (i.e., ORs are greater than 1), which is consistent with previous findings by Nagappan and Ball [27]. By comparing the likelihood of future bugs between the second and the fourth groups, we find that the likelihood of future bugs increases further (i.e. ORs are greater than 1), if files with high relative code churn experience any of concurrent, extended, or interrupted editing patterns. Detailed results are presented in Appendix D. Therefore, we can conclude that the relative code churn alone cannot explain the bug proneness of files that were edited following the four editing patterns.

7. THREATS TO VALIDITY

We now discuss the threats to validity of our study following common guidelines provided in [28].

Construct validity threats concern the relation between theory and observation. Our construct validity threats are mainly because of measurement errors. We rely on Mylyn logs to collect information about file editing patterns. Because some files may be edited without using Mylyn, our file editing information might be biased. Another potential source of bias is the computation of the numbers of future bugs. We automatically mined bug IDs from the bug fixing change logs and used the bug ID to link the bug fixing change logs to the change request reports. We manually sampled a number of change logs. Although we confirmed that bug ID always existed in the change logs for bug fixing or enhancement implementation (other than fixing style issue or configuration errors), it is possible that some future bugs were missed. However, we found that the bug data of Mylyn project of our study was consistent with the study of Lee *et al*. [10], which are publicly available.

Threats to internal validity concern our selection of subject systems and analysis methods. Although we study three software systems, some of the findings might still be specific to the development and maintenance processess of the three software systems that are Eclipse projects. In fact, the usage of Mylyn in the projects is likely to have affected the editing patterns of developers. Future studies should consider using a different tool to collect file editing data. The 14 metrics used in this study to measure code quality are commonly used in the literature to measure software maintainability [19], yet they might not be enough to measure the initial code quality of files. Moreover, the computation of the metrics using a different tool (i.e. other than the Understand tool) may yield different values. Future studies should consider investigating more metrics and using other tools. We apply pairwise comparison to understand the effect of file editing patterns on software quality. Pairwise comparison is suitable to investigate the impact of a single factor (e.g. the presence of a particular file editing pattern) but becomes difficult if analyzing many confounding factors together. Hence, we only controlled some important confounding factors (e.g. size and relative code churn). Future studies could apply ANOVA or other techniques to investigate more confounding factors.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models. We have used non-parametric tests that do not require making assumptions about the distribution of data sets.

We have controlled several important confounding factors (including the ownership of files, the type of change requests, the code quality, and the relative code churn) and found that the presence of file editing patterns provides additional explanatory power in explaining the bug proneness.

DOI: 10.1002/smr

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. Eclipse CVS and Bugzilla are publicly available to obtain the same data. All the data used in this study are also publicly available online.[¶¶]

Threats to external validity concern the possibility to generalize our results. We only analyzed three Eclipse projects, because of the limited adoption of Mylyn in open source projects. Further studies on different open and closed source systems are desirable to verify our findings.

8. RELATED WORK

The work presented in this paper relates to the analysis of file editing patterns and bug prediction. In the following subsections, we summarize the related research.

8.1. Analysis of file editing patterns

To the best of our knowledge, our earlier work [8] is the first attempt to empirically quantify the impact of concurrent, parallel, extended, and interrupted file editing patterns on software bug proneness. This paper extends the earlier work [8] by investigating the factors that impact the occurrence of file editing patterns and by examining joint effect of file editing patterns and several factors (e.g. the ownership of files, the type of change requests, and the code quality) on software quality.

A large body of research has been conducted on development activities; especially, many tools have been proposed to improve developers' awareness about project activities such as source code changes or development task creation. For example, the tools Codebook [4] and Crystal [5] have been proposed to warn developers about potential file editing conflicts. Treude and Storey [6], who investigate the usage of dashboards and feeds by development teams using data collected from the IBM Jazz development platform, reported on the need for better awareness tools that could provide both high-level awareness (e.g. about project team members, upcoming deadlines) and low-level awareness (e.g. about source code changes). Despite all research on developing better awareness tools, there are very few studies that empirically investigated the consequences of a lack of awareness of developers about the file editing patterns followed by team members.

Perry *et al*. [29] investigate file editing patterns in a large telecommunication software system and find that about 50% of the files are modified consecutively by more than one developer in the period between two releases of the software. They do not study the concurrent editing of files but nevertheless report that files edited by multiple developers were at a higher risk for bugs. One possible reason is that concurrent changes could introduce conflicts. This is observed by Staudenmayer *et al*. [30] in the study of concurrent changes at module level in another telecommunication software system.

In our study, we not only propose a metric to identify the concurrent editing pattern but also report that the concurrent editing pattern can increase the likelihood of future bugs to 1.8 times.

D'Ambros *et al*. [24] investigate the relation between change coupling and bugs through an analysis of files frequently committed together. They conclude that change coupling information can improve the performance of bug prediction models. However, relying on commit logs to identify files that are changed together is not very accurate. The fact that two files are submitted together into a software repository does not necessarily mean that the files are modified in parallel by one or many developers. Developers often edit multiple files (at different times) and commit the files all together. Moreover, in some systems, many developers editing files are not permitted to commit their changes to the code repository directly. In these systems, file modifications are validated by a review team prior to their submission into the software repository. The developer submitting the files is often one of the reviewers.

In this work, we precisely identify file edits that happened at the same time from the rich developers' interaction logs collected by the development teams of the three Eclipse projects, using the Mylyn tool.

[¶¶]https://bitbucket.org/serap/fileeditingpatternstudy.

DOI: 10.1002/smr

Lee *et al*. [10] propose to use the time spent by developers on tasks to predict future bugs in the files involved in the tasks. This study proposes to measure the maximum editing time and find that extended editing can increase the likelihood of future bugs by 1.9 times. Parnin and Rugaber [31] investigate developers' interruptions during software development tasks and report on the strategies adopted by developers to successfully resume a task after an interruption. However, they do not assess the relationship between the likelihood that a bug would be introduced and the length of interruptions. We propose to measure the maximum editing time and find that interrupted editing can increase the likelihood of future bugs by 2.0 times.

As a summary, this work proposes four metrics to identify the concurrent, parallel, extended and interrupted editing patterns, respectively. We further empirically quantify the likelihood of having bugs in files edited following the four file editing patterns.

8.2. Bug prediction

A large number of studies have investigated the use of metrics to predict the location of future bugs in software systems. For instance, Khoshgoftaar *et al*. [32] report that combining code metrics and knowledge from problem reporting databases can yield good results in bug predictions. Moser *et al*. [33] however show that process metrics outperform source code metrics as predictors of future bugs. Other researchers focus on using temporal information for bug prediction. Bernstein *et al*. [34] use temporal aspects of data (i.e. the number of revisions and corrections recorded in a given amount of time) to predict the location of bugs. The resulting model can predict whether a source file has a bug with 99% accuracy. Arisholm and Briand [35] propose a model based on code quality, class structure, changes in class structure metrics, and also measures of the history of class-level changes and bugs, to predict future bugs in classes. They perform a cost-effectiveness analysis and show that the estimated potential savings in verification effort of their model are about 29%. Nagappan and Ball [36] show that relative code churn metrics are good predictors of bug density in systems. Askari and Holt [37] provide a list of mathematical models to predict where the next bugs are likely to occur. Different from aforementioned prior work, the goal of this study is not to improve the performance of bug prediction models. We propose four metrics to identify four file editing patterns and aim to analyze the relation among the occurrence of our file editing patterns and future bugs. We also analyze the interaction among the editing patterns. Moreover, we examine the joint effect of file editing patterns and several factors (e.g. code quality) that are often used in bug prediction models.

9. CONCLUSION

In this paper, we analyzed the developers' interaction logs of three open source software projects, Mylyn, Eclipse Platform, and Eclipse PDE, and proposed metrics to identify four file editing patterns. We investigated the potential impact of the file editing patterns on software quality and factors impacting the occurrence of the file editing patterns. We also investigated if the ownership of files, the type of change requests, and the initial code quality affect the likelihood of future bugs in files edited following one of the four file editing patterns. During development and maintenance activities, 23% to 90% of files were edited following at least one of the concurrent, parallel, extended, and interrupted file editing patterns. The ownership of files and the severity of bug fix change requests affect the occurrence of the four editing patterns. The initial code quality of files, in particular the size, complexity, coupling, and the ratio of comments to code, also affects the occurrence of the file editing patterns.

Whenever concurrent or extended editing pattern is followed during the modifications of a file, the risk of future bugs in the file increases. Files edited following concurrent pattern are 1.8 times more likely to experience future bugs. Developers and managers should also be cautious when one file is edited in parallel with too many other files. If a developer is spending too much time (i.e. longer than 123 h) editing one file for a change request, development teams should consider inspecting the root cause of this delay and propose appropriate solutions (e.g. split the change request or refactor

DOI: 10.1002/smr

the file). Also, development teams should avoid having developers spending too long time editing one file. They should also avoid interrupting their developers frequently. One possible solution could be to decompose each large change request into a set of small change requests.

Our empirical study showed that there exists a relationship between the occurrence of the four file editing patterns and the following three factors: the ownership of files, the type of change requests, and the initial code quality. In the presence of concurrent, extended, and interrupted editing pattern, the likelihood of future bugs in a file that underwent more bug fixes than enhancement implementations increased 3.9, 5.5, and 4.7 times, respectively. Moreover, we observed that a strong ownership on files and good code quality can alleviate the negative impact of the four file editing patterns on software quality. We recommend that development teams monitor and control the following five metrics: maxCC, maxCBO, sumCBO, maxLCOM, and sumLCOM, and add more comments to their code (i.e. measured by RCC).

We also observed that when more than one editing patterns are followed by one or multiple developers during the editing of a file, the risk for future bugs in the file increases further. For instance, files edited following concurrent, extended, and interrupted patterns together are 3.9 times more likely to experience future bugs than files edited but never following any of the four patterns. In addition, the density of future bugs in files edited following more than one editing patterns is 2.1 times higher than in files edited but never following any of the four patterns. More attention should be paid to files edited following more than one editing patterns.

This work provides empirical evidence of the negative impact of concurrent, parallel, extended, and interrupted file editing patterns on software quality. Designers of awareness tools should consider integrating new features to track the four file editing patterns analyzed in this study, so that a developer working on a file can remain aware of the editing patterns of other developers working on related files. In future work, we plan to do that. We will propose an Eclipse plug-in to automatically identify our four editing patterns from collected Mylyn logs and inform developers about the pattern occurrences and interactions.

The authors would like to thank the anonymous reviewers of the 19th Working Conference on Reverse Engineering (WCRE) and the Journal of Software: Evolution and Process (JSEP) for their insightful comments, and Mrs. Quan Zheng and Nicolas Bettenburg from Queen's University for their help on data processing and analysis during the initial stage of this work.

APPENDIX A: Source code metrics

Appendix A1. List of source code metrics and the corresponding metric name used by the Understand tool.

Category	Metric	Description	Metric name in Understand
Complexity	CLOC	Class lines of code	CountLineCode
	NIM	Number of instance methods	CountDeclInstanceMethod
	NIV	Number of instance variables	CountDeclInstanceVariable
	WMC	Weighted methods per class	CountDeclMethod
	CC	McCabe cyclomatic complexity	Cyclomatic
Couping	CBO	Coupling between objects	CountClassCoupled
	RFC	Response for a class	CountDeclMethodAll
Cohesion	LCOM	Lack of cohesion in methods	PercentLackOfCohesion
Abstraction	DIT	Depth of inheritance tree	MaxInheritanceTree
	IFANIN	Number of immediate base classes	CountClassBase
	NOC	Number of immediate subclasses	CountClassDerived
Encapsulation	RPM	Ratio of public methods	$\frac{\text{CountDeclMethodPublic}}{\text{CountDeclMethod}}$
Documentation	CL	Comment of lines	CountLineComment
	RCC	Ratio comments to codes	RatioCommentToCode

DOI: 10.1002/smr

APPENDIX B: Results for separate projects

Appendix B1. Relation between the occurrence of patterns and the risk of bugs and the corresponding *P*-value.

Pattern	Mylyn	Platform	PDE	All
Concurrent	**1.6*** (1.85e − 02)	0.0 (2.49e − 01)	2.1 (2.28e − 01)	**1.8*** (2.03e − 03)
Parallel	2.2 (1.28e − 01)	0.6 (3.80e − 01)	2.2 (3.81e − 01)	1.5 (2.15e − 01)
Extended	**2.3*** (1.82e − 04)	1.0 (1.00e + 00)	**3.0*** (3.14e − 02)	**1.9*** (6.63e − 04)
Interrupted	**2.4*** (6.95e − 05)	0.9 (8.35e − 01)	**4.7*** (1.10e − 03)	**2.0*** (6.73e − 05)

*indicates that the test was statistically significant, i.e. *P*-value < 0.05

APPENDIX C: Results for different strategies on group split

Appendix **C1**. Relation between the occurrence of patterns and the risk of bugs, and the corresponding *P*-value.

Pattern	No any pattern	No oattern P_i but other patterns
Concurrent	**2.1*** (4.10e − 02)	**1.8*** (1.90e − 03)
Parallel	1.4 (3.22e − 01)	1.4 (3.90e − 01)
Extended	**2.1*** (3.20e − 02)	**1.8*** (9.66e − 04)
Interrupted	**2.2*** (1.71e − 02)	**2.0*** (1.40e − 04)

*indicates that the test was statistically significant, i.e. *P*-value < 0.05

APPENDIX D: Results for investigating relative code churn

Appendix D1. Relation between the occurrence of patterns and the risk of bugs, and the corresponding *P*-value.

	<No pattern P_i, high code churn> versus <no pattern P_i, low code churn>	<Pattern P_i, high code churn> versus <no pattern P_i, high code churn>
Concurrent	**1.6*** (4.02e − 02)	**2.4*** (3.59e − 04)
Parallel	2.0 (3.56e − 01)	1.4 (4.74e − 01)
Extended	**2.0*** (1.63e − 03)	**1.7*** (2.49e − 02)
Interrupted	**1.9*** (2.85e − 03)	**1.8*** (1.11e − 02)

*indicates that the test was statistically significant, i.e. *P*-value < 0.05

REFERENCES

1. Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing, May 2002. National Institute of Standards and Technology.
2. Biehl JT, Czerwinski M, Smith G, Robertson GG. Fastdash: a visual dashboard for fostering awareness in software teams. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '07, ACM: New York, NY, USA, 2007; 1313–1322.
3. Sarma A, Bortis G, van der Hoek A. Towards supporting awareness of indirect conflicts across software configuration management workspaces. Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. ASE '07, ACM: New York, NY, USA, 2007; 94–103.
4. Begel A, Khoo YP, Zimmermann T. Codebook: discovering and exploiting relationships in software repositories. Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering. ICSE '10, ACM: New York, NY, USA, 2010; 125–134.
5. Brun Y, Holmes R, Ernst MD, Notkin D. Proactive detection of collaboration conflicts. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11, ACM: New York, NY, USA, 2011; 168–178.
6. Treude C, Storey M. Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds. *Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering*, ICSE '10, 2010; 365–374.
7. Mylyn. Available from: http://wiki.eclipse.org/Mylyn/Integrator_Reference [July 3 2014].

DOI: 10.1002/smr

8. Zhang F, Khomh F, Zou Y, Hassan AE. An empirical study of the effect of file editing patterns on software quality. *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12, 2012; 456–465.
9. Zimmermann T, Premraj R, Zeller A. Predicting defects for eclipse. *Proceedings of the International Workshop on Predictor Models in Software Engineering*, PROMISE '07, 2007; 9.
10. Lee T, Nam J, Han D, Kim S, In HP. Micro interaction metrics for defect prediction. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11, ACM: New York, NY, USA, 2011; 311–321.
11. Zhang F, Khomh F, Zou Y, Hassan AE. An empirical study on factors impacting bug fixing time. *Proceedings of the 19th Working Conference on Reverse Engineering*, WCRE '12, 2012; 225–234.
12. Shihab E, Jiang ZM, Ibrahim WM, Adams B, Hassan AE. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. Proceedings of the 2010 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '10, ACM: New York, NY, USA, 2010; 4:1–4:10.
13. Bird C, Nagappan N, Murphy B, Gall H, Devanbu P. Don't touch my code!: examining the effects of ownership on software quality. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ESEC/FSE '11, ACM: New York, NY, USA, 2011; 4–14.
14. Bachmann A, Bird C, Rahman F, Devanbu P, Bernstein A. The missing links: Bugs and bug-fix commits. Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE '10, ACM: New York, NY, USA, 2010; 97–106.
15. Ying A, Robillard M. The influence of the task on programmer behaviour. *Proceedings of the IEEE 19th International Conference on Program Comprehension*, ICPC '11, 2011; 31–40.
16. Scitools. Metrics computed by understand. Available from: http://www.scitools.com/documents/metricsList.php [July 03, 2014].
17. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE) Jun* 1994; **20**(6):476–493.
18. McCabe T. A complexity measure. *IEEE Transactions on Software Engineering (TSE)* 1976; **SE-2**(4):308–320.
19. Zhang F, Mockus A, Zou Y, Khomh F, Hassan AE. How does context affect the distribution of software maintainability metrics? *Proceedings of the 29th IEEE International Conference on Software Maintainability*, ICSM '13, 2013; 350–359.
20. Sheskin DJ. Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition. Chapman & Hall/CRC: Boca Raton, 2007.
21. Holm S. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 1979; **6**(2):65–70.
22. Nagappan N, Ball T, Zeller A. Mining metrics to predict component failures. Proceedings of the 28th International Conference on Software Engineering. ICSE '06, ACM: New York, NY, USA, 2006; 452–461.
23. Raymond E. The cathedral and the bazaar. *Knowledge, Technology and Policy* 1999; **12**(3):23–49.
24. D'Ambros M, Lanza M, Robbes R. On the relationship between change coupling and software defects. *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE '09, 2009; 135–144.
25. Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A. Predicting faults from cached history. Proceedings of the 29th International Conference on Software Engineering. ICSE '07, IEEE Computer Society: Washington, DC, USA, 2007; 489–498.
26. Olkin I. Contributions to probability and statistics: essays in honor of Harold Hotelling. Stanford studies in mathematics and statistics, Stanford, Calif. Stanford University Press, 1960.
27. Nagappan N, Ball T. Static analysis tools as early indicators of pre-release defect density. *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, 2005; 580–586.
28. Yin RK. Case Study Research: Design and Methods - Third Edition. 3 edn., SAGE Publications: Thousand Oaks, CA, USA, 2003.
29. Perry D, Siy H, Votta L. Parallel changes in large scale software development: an observational case study. *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, 1998; 251–260.
30. Staudenmayer N, Graves T, Perry D. Adapting to a new environment: how a legacy software organization copes with volatility and change. *Proceedings of the 5th International Product Development Conference*, IPDC'98, 1998.
31. Parnin C, Rugaber S. Resumption strategies for interrupted programming tasks. *Proceedings of the IEEE 17th International Conference on Program Comprehension*, ICPC'09, 2009; 80–89.
32. Khoshgoftaar TM, Allen EB, Jones WD, Hudepohl JP. Data mining for predictors of software quality. *International Journal of Software Engineering and Knowledge Engineering (SEKE)* 1999; **9**(5):547–563.
33. Moser R, Pedrycz W, Succi G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. Proceedings of the 30th International Conference on Software Engineering. ICSE '08, ACM: New York, NY, USA, 2008; 181–190.
34. Bernstein A, Ekanayake J, Pinzger M. Improving defect prediction using temporal features and non linear models. Proceedings of the 9th International Workshop on Principles of Software Evolution. IWPSE '07, ACM: New York, NY, USA, 2007; 11–18.
35. Arisholm E, Briand LC. Predicting fault-prone components in a java legacy system. Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering. ISESE '06, ACM: New York, NY, USA, 2006; 8–17.
36. Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. Proceedings of the 27th International Conference on Software Engineering. ICSE '05, ACM: New York, NY, USA, 2005; 284–292.
37. Askari M, Holt R. Information theoretic evaluation of change prediction models for large-scale software. Proceedings of the 2006 International Workshop on Mining Software Repositories. MSR '06, ACM: New York, NY, USA, 2006; 126–132.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES

Feng Zhang received his B.S. degree in electronic information engineering from Nanjing University of Science and Technology (China) in 2004 and his M.S. degree in control science and engineering from Nanjing University of Science and Technology (China) in 2006. He is currently pursuing the Ph.D. degree in computer science from Queen's University in Canada. His research interests include empirical software engineering, software re-engineering, mining software repositories, and source code analysis.

Foutse Khomh is an assistant professor at the École Polytechnique de Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca). He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM, MSR, WCRE, ICWS, JSS, JSP, and EMSE. He has served on the program committees of several international conferences including ICSM, WCRE, MSR, ICPC, SCAM and has reviewed for top international journals such as SQJ, EMSE, TSE and TOSEM. He is program co-chair of the Workshops track at WCRE 2013, program chair of the Tool track at SCAM 2013, and program chair for Satellite Events at SANER 2015. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and guest editor for a special issue on Release Engineering in the IEEE Software magazine.

Ying Zou is a Canada Research Chair in Software Evolution. She is an associate professor in the Department of Electrical and Computer Engineering and cross-appointed to the School of Computing at Queen's University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.

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

DOI: 10.1002/smr