An Empirical Study of the Effect of File Editing Patterns on Software Quality

Feng Zhang[1], Foutse Khomh[2], Ying Zou[2], and Ahmed E. Hassan[1]
[1]*School of Computing, Queen's University, Canada*
{*feng, ahmed*}*@cs.queensu.ca*
[2]*Department of Electrical and Computer Engineering, Queen's University, Canada*
{*foutse.khomh, ying.zou*}*@queensu.ca*

***Abstract*—While some developers like to work on multiple code change requests, others might prefer to handle one change request at a time. This juggling of change requests and the large number of developers working in parallel often lead to files being edited as part of different change requests by one or several developers. Existing research has warned the community about the potential negative impacts of some file editing patterns on software quality. For example, when several developers concurrently edit a file as part of different change requests, they are likely to introduce bugs due to limited awareness of other changes. However, very few studies have provided quantitative evidence to support these claims. In this paper, we identify four file editing patterns. We perform an empirical study on three open source software systems to investigate the individual and the combined impact of the four patterns on software quality. We find that: (1) files that are edited concurrently by many developers have on average 2.46 times more future bugs than files that are not concurrently edited; (2) files edited in parallel with other files by the same developer have on average 1.67 times more future bugs than files individually edited; (3) files edited over an extended period (*i.e.*, above the third quartile) of time have 2.28 times more future bugs than other files; and (4) files edited with long interruptions (*i.e.*, above the third quartile) have 2.1 times more future bugs than other files. When more than one editing patterns are followed by one or many developers during the editing of a file, we observe that the number of future bugs in the file can be as high as 1.6 times the average number of future bugs in files edited following a single editing pattern. These results can be used by software development teams to warn developers about risky file editing patterns.**

***Keywords*-file editing pattern; change request; bug; software quality; empirical software engineering; mylyn.**

I. Introduction

Bugs are generally introduced inadvertently by developers when performing source code change requests. It is estimated that 80% of software development costs are spent on correcting bugs [1]. To implement a change request, a developer must change one or many files. File changes are done through editing each involved file one or several times. As developers work concurrently on several changes in parallel, several file editing patterns emerge. For example, a developer might edit multiple files simultaneously. Another developer might prefer to edit files one by one. Some others may follow both editing patterns. Developers often follow the editing pattern that best suits their personal skills, schedule constraints, and programming experience. However, we conjecture that some file editing patterns are likely riskier than others, and that it is important to raise the awareness of development teams about the editing patterns followed by developers.

Despite the large body of work on awareness tools for software development [2], [3], [4], there are very few studies that empirically investigated the risks posed by a 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 perform such a study, one needs detailed information about file editing activities occurring in developers' workspaces. A tool such as Mylyn [5] which records and monitors developer's programming activities, like the selection and the editing of files, provides the opportunity for such a study.

In this paper, we analyze developers' integrated development environment (IDE) interaction logs (*i.e.*, logs recording developers' selection and editing of files) from three open source software systems, Mylyn[1], Eclipse Platform[2] and Eclipse Plug-in Development Environment (PDE)[3]. We identify four file editing patterns and analyze the relations between these patterns and the occurrences of bugs. We briefly summarize our findings:

- **Concurrent editing:** several developers edit the same file concurrently.
  On average, files that are edited concurrently by many developers have 2.46 times more future bugs than files that are not involved in any concurrent editing.
- **Parallel editing:** multiple files are edited in parallel by the same developer.
  On average, files edited in parallel with other files by the same developer are 1.67 times more buggier in the future than files edited individually.
- **Extended editing:** developers spend longer time editing a file, *e.g.*, duration of editing periods above the third quartile of the editing times of all files.
  On average, files edited over a period of time greater than the third quartile have 2.28 times more future bugs than other files.

[1]http://www.eclipse.org/mylyn/
[2]http://www.eclipse.org/platform/
[3]http://www.eclipse.org/pde/

- **Interrupted editing:** developers observe long idle times during the editing of a file, *e.g.*, duration of idle periods above the third quartile of all the idle periods observed by developers.
  On average, files edited with interruption time greater than the third quartile have 2.1 times more future bugs than other files.

When more than one editing patterns are followed by one or many developers during the editing of a file, the risk of future bugs in the file increases further.

- **Interactions between the patterns:** the likelihood of future bugs in a file edited following more than one editing pattern is higher than the likelihood of future bugs in a file edited following a single editing pattern. The number of future bugs in a file edited by developers following more than one editing patterns can be as high as 1.6 times the average number of future bugs in files edited following a single editing pattern.

The reminder of this paper is structured as follows. We describe the four studied file editing patterns in Section II. Section III provides some background on the task and application lifecycle management framework Mylyn. Section IV introduces the setup of our case study and describes our analysis approach. Section V presents the results of our study. Section VI discusses threats to the validity of our study. Section VII relates our study with previous work. Finally, Section VIII summarizes our findings and outlines some avenues for future work.

II. File Editing Patterns

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

A. Concurrent Editing Pattern

During the development and maintenance activities, developers are sometimes assigned inter-dependant change requests. This situation results in some files being edited concurrently by different developers 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 since each developer is performing his or her edits independently in his or her own working space, before all the changes are merged together eventually.

B. Parallel Editing Pattern

Developers sometimes edit a number of files in parallel when performing a change request. In this situation, some files are edited in parallel. 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 *Frank*. With the parallel editing pattern, a developer has a higher chance to become distracted due to frequent switches between files.

C. Extended Editing Pattern

When changing a file as part of a change request, a developer might end up performing several edits on the file. These edits might be done over a short span of time or might be done over an extended span of time. We refer to the second scenario as the extended editing pattern. An example of extended editing occurred in the Mylyn project. The average editing time of *DiscoveryViewer.java* is 4.5 hours, which is 3.6 times the median (*i.e.*, 1.25 hours) of the editing times of all files in the Mylyn project. We conjecture that extended edits are possibly risky since 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.

D. Interrupted Editing Pattern

During development activities, developers are often interrupted by email alerts, meetings, or other duties. They might also simply take a break which may last for a few minutes or longer. We refer to this phenomenon as the interrupted editing pattern. An example of interrupted editing occurred in the Mylyn project. The average interruptions time of *TaskCompareDialog.java* is 338.0 hours, which is 164 times the median (*i.e.*, 2.06 hours) of the editing times 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.

III. Background on the Framework Mylyn

Mylyn is an Eclipse plug-in that monitors developer's programming activities, such as selection and editing of files. In Mylyn, each developer's activity is an *event*. There are eight types of *events* in Mylyn: *Attention*, *Command*, *Edit*, *Manipulation*, *Prediction*, *Preference*, *Propagation*, and *Selection* [5]. Three of the eight *events* are triggered by a developer, *i.e.*, *Command*, *Edit* and *Selection* events. A Mylyn log records a list of events triggered by developers during programming activities, such as bug fixing or feature enhancement. Mylyn logs are stored in an XML format. Each Mylyn log is identified by a unique *Id* (*i.e.*, the task identifier) and contains descriptions of events (*i.e.*, *InteractionEvent*) recorded by Mylyn. The description of each event includes: a starting date (*i.e.*, *StartDate*), an end date (*i.e.*, *EndDate*), a type (*i.e.*, *Kind*), the identifier of the UI affordance that tracks the event (*i.e.*, *OriginId*), and the

Table I: The three subject systems.

System	Description	# of change request reports	# of logs
Mylyn	Task and application lifecycle management framework.	2,722	3,883
Platform	Core frameworks, services and runtime provider of Eclipse.	606	793
PDE	Eclipse plug-in development environment.	524	638

names of the files involved in the event (*i.e.*, *StructureHandle*). Mylyn logs are compressed, encoded under the Base64 format, attached to change request reports, and stored in change request tracking systems.

IV. Case Study Setup

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

1) Are there different file editing patterns?
2) Do file editing patterns lead to more bugs?
3) Do interactions among file editing patterns lead to more bugs?

A. Data Collection

In this study, we use Mylyn interaction logs to identify file editing patterns. As an Eclipse project, Mylyn is frequently used in other Eclipse projects. We choose three Eclipse projects with the highest number of change request reports containing Mylyn logs. Table I shows the descriptive statistics of the subject systems. In total, we examine 2,140 files that have been modified by 119 developers working on the subject systems.

B. Data Processing

Figure 1 shows an overview of our approach. First, we extract revision history data from the source code repositories (*i.e.*, CVS). We also extract Mylyn interaction logs from change request repositories (*i.e.*, Bugzilla). We compute several metrics to identify file editing patterns. We then statistically compare the proportion of buggy files (and the number of bugs in files) edited following the patterns.

1) Recovering File Change History: In our case study, we measure software quality using the number of bugs in files. For each file changed by developers, we extract bug fixing change information. The three subject systems use CVS to track source code changes. CVS change logs contain the whole history of revisions to the source code. We downloaded the CVS repositories of our three subject systems on October 20, 2011. We select the date of January 1, 2011 to separate the pattern analysis period from the period for counting future bugs (*i.e.*, bugs reported after developers' changes). We refer to the period after the split date as the future bug counting period. A similar decision is made in a study by Lee *et al.* [6]. Bugs extracted from commit logs between January 1, 2011 and October 20, 2011 are considered to be future bugs. We use Mylyn logs from January 1, 2009 to December, 31, 2010 to collect information of file editing patterns.

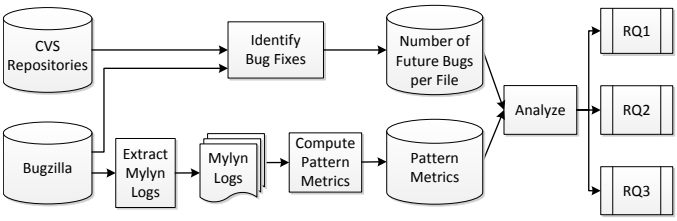


Figure 1: Overview of our approach to analyze the effect of file editing patterns on code quality

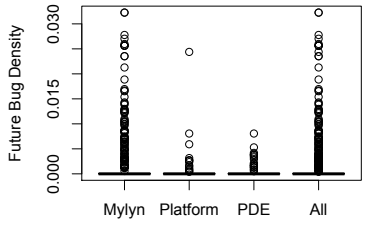


Figure 2: Box plot of the density of future bugs in Mylyn, Platform, and PDE.

We extracted the change logs of all commits performed during the future bug counting period. In total, we obtained 4,492 logs from the Eclipse repository (*i.e.*, PDE and Platform), and 578 logs from the Mylyn repository (*i.e.*, Mylyn). We manually analyzed each log to identify bug fixing change logs. We also extracted bug IDs from the obtained bug fixing change logs. For each bug ID, we downloaded the corresponding bug report from Bugzilla and extracted the bug opening date. We compared the bug opening date with the split date. We filtered out all the bugs opened before the split date and all enhancements. In total, we obtained 98 future bugs from 2,140 files modified by 119 developers from the subject systems. For each file, we calculate the density of future bugs by dividing the number of future bugs of the file by the size of the file. Figure 2 shows the box plot of the density of future bugs in our studied systems. Similar to [6], [7], we combined data from the three subject systems because of their small sizes.

2) Recovering File Edit History: In a change request report, an attachment containing a Mylyn log is named "mylyn/context/zip". We search Eclipse Bugzilla to generate a list of change request reports with attachments of that name. We download the change request reports in the list. We parse the reports to extract information, such as reporting date, reporter's name, project and module names, comments, Mylyn attachments, and attachers' names. We decoded and unzipped Mylyn attachments to extract Mylyn logs. We consider the *attacher* of a Mylyn log to be its owner, since there is no explicit ownership information in Mylyn logs.

We parse each Mylyn log to extract *Edit* and *Selection* events. We rely on *Edit* and *Selection* events to track developers' accesses to files and compute the duration of developers' file editing periods. *Edit* events are issued when a developer selects the content (*i.e.*, the text) of a file in the Eclipse IDE and *Selection* events are triggered when a developer selects a file. For each *Edit* or *Selection* event, we extract the start date, the end date, and the names of the files concerned by the event.

3) Identifying File Editing Patterns: Using *Edit* event information collected from Mylyn logs, we propose a set of metrics for detecting editing patterns followed by developers during the development/maintenance of the systems. In the following subsections, we discuss the detection of each editing pattern in details.

a) 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 [8]. We control for that by dividing the number of concurrent edits and the number of developers by the number of changes, following respectively Equation (1) and Equation (2). We obtain the average number of concurrent edits per change ($N_{ConEdits}$) and the average number of developers editing the file concurrently during a change ($N_{ConDevs}$).

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

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

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.
$Overlap_{CD}(Edit_i, Edit_j)$ 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 average number of concurrent edits per change of F is $N_{ConEdits} = \frac{(1+2+1)}{3} = 1.33$. If $Edit_1$ was performed by developer d_1, while $Edit_2$ and $Edit_3$ by developers 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 concurrent edits in F is $N_{ConDevs} = \frac{(2+1+1)}{3} = 1.33$.

For each file, we compute the $N_{ConDevs}$ value. We conclude that a file was modified following the *concurrent editing* pattern, if and only if its $N_{ConDevs}$ is greater than 0.

b) Parallel Editing Pattern. We compute the number of parallel editing files of an edit i ($n_{ParallelEdits}(i)$) using Equation (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 Equation (4).

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

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 5 other files; in $Edit_2$, F was modified in parallel with 9 other files; in $Edit_3$, F was modified solely. The number of parallel editing files of the three edits are: $n_{ParallelEdits}(1) = 5$, $n_{ParallelEdits}(2) = 9$, $n_{ParallelEdits}(3) = 1$. The average number of parallel editing files of F is $N_{ParallelEdits} = \frac{(5+9+1)}{3} = 5$.

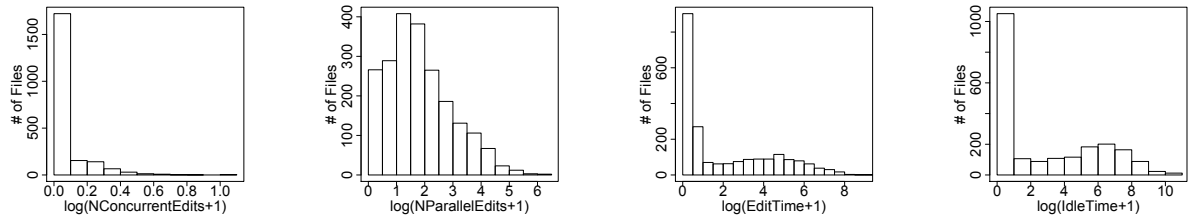
For each file, we compute the $N_{ParallelEdits}$ value. We conclude that a file was modified following the *parallel editing* pattern, if and only if its $N_{ParallelEdits}$ is greater than 0.

c) 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 Equation (5). We sum the $editTime(i)$ values of all edits in the history of the file. To control for the confounding effect of the number of changes made to the file, we divide the sum of $editTime(i)$ by the number of changes and obtain the average editing time ($EditTime$), following Equation (6).

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

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.*

$$EditTime = \frac{1}{N}\sum_{i=1}^{N} editTime(i) \quad (6)$$



(1) Concurrent editing (2) Parallel editing (3) Extended editing (4) Interrupted editing

Figure 3: Distribution of metric values for the file editing patterns

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, $editTime(1) = 1$ hours and $editTime(2) = 2$ hours. The average editing time of F is $EditTime = \frac{1}{2}(1+2) = 1.5$.

For each file, 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.

d) 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 Equation (7). We sum the $idleTime(i)$ values of all changes in the history of the file. To control for the confounding effect of the number of changes made to the file, we divide the sum of $idleTime(i)$ by the number of changes and obtain the average interruption time ($IdleTime$), following Equation (8).

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

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.*

$$IdleTime = \frac{1}{N} \sum_{i=1}^{N} idleTime(i) \quad (8)$$

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: $idleTime(1) = 2$ hours and $idletime(2) = 16$ hours. The average interruption time of F is $IdleTime = \frac{1}{2}(2+16) = 9$.

For each file, 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.

C. Analysis Method

We study if bugs in files are related to file editing patterns.

1) Analyzing the relationship between a file editing pattern and the probability of future bugs: We use the Fisher's exact test [9] to determine if there are non-random associations between a particular file editing pattern and the occurrence of future bugs. We also compute the *odds ratio* (OR) [9] indicating the likelihood of an event to occur (*i.e.*, a bug). OR is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the set of files edited following a specific editing pattern (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the set of files edited but not following the pattern (control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event (*i.e.*, a bug) is equally likely in both samples. 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 OR < 1 indicates the opposite (*i.e.*, the control group of files edited but not following the pattern).

2) Analyzing the relationship between a file editing pattern and the number of future bugs: We use the Wilcoxon rank sum test [9] to compare the number of future bugs of files edited following a particular pattern and other files that were edited but not following the pattern. 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 cases of comparisons among more than two groups of files, we apply the Kruskal-Wallis rank sum test [9] which is an extension of the Wilcoxon rank sum test to more than two groups.

V. Case Study Results

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

RQ1: Are there different file editing patterns?

Motivation. This question is preliminary to the other questions. It provides the quantitative data on the number of files edited by developers following the four editing patterns. In this research question, we determine if all four editing patterns are followed by developers when working on change requests, and are therefore worth investigating individually. We also determine the existence of interactions between the editing patterns, *e.g.*, if a file can be edited concurrently by

Table II: Occurrences of file editing patterns and their interactions

ID	List of patterns or combination of patterns	# Files.
(0)	No patterns	201
(1)	<Concurrent>	497
(2)	<Parallel>	1922
(3)	<Extended>	535
(4)	<Interrupted>	535
(1, 2)	<Concurrent, Parallel>	494
(1, 3)	<Concurrent, Extended>	236
(1, 4)	<Concurrent, Interrupted>	190
(2, 3)	<Parallel, Extended>	525
(2, 4)	<Parallel, Interrupted>	528
(3, 4)	<Extended, Interrupted>	311
(1, 2, 3)	<Concurrent, Parallel, Extended>	236
(1, 2, 4)	<Concurrent, Parallel, Interrupted>	190
(1, 3, 4)	<Concurrent, Extended, Interrupted>	133
(2, 3, 4)	<Parallel, Extended, Interrupted>	308
(1, 2, 3, 4)	<Concurrent,Parallel,Extended,Interrupted>	133

different developers, over an extended period of time.

Approach. We answer this research question by classifying the files of our subject systems using the patterns followed by developers during file editing. We identify the editing pattern(s) of a file using the metrics described in Section IV-B3. Figure 3 shows the distribution of the metrics values. For each pattern (or combination of patterns), we report the number of files edited following the pattern (or the combination of patterns).

Findings. Table II summarizes the number of files that were edited following each pattern or combination of patterns. As shown in Table II, only 201 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 (1,922 files). 949 files from our systems were edited following more than one editing pattern.

Overall, we conclude that developers follow the four file editing patterns during development and maintenance activities. In the next two research questions we examine the patterns (and their interactions) in more detail to determine if some file editing patterns (and interaction between patterns) are more risky than others.

RQ2: Do file editing patterns lead to more bugs?

Motivation. In **RQ1**, 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. Similarly to **RQ1**, we identify the editing pattern of a file using the metrics described in Section IV-B3. We classify the files based on the patterns followed by developers during file editing. For each pattern P_i, we create two groups: a group GP_i containing files that were edited by developers following P_i and another group NGP_i containing files that were edited by developers not following P_i. We also compute the number of future bugs of each file. Because previous studies (*e.g.*, [10], [11]) have found size to be related to the number of bugs in a file. We divide the number of future bugs of each file by the size of the file to control for the confounding effect of size. We obtain the density of future bugs of each file.

For each pattern P_i, we test the two following null hypothesis (there is no H_{01} because **RQ1** is exploratory):

H^1_{02}: *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).*

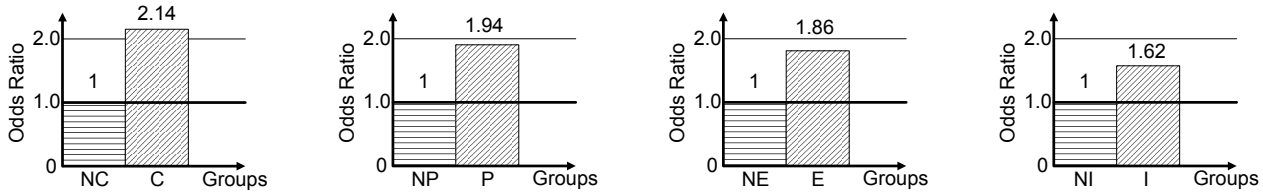
H^2_{02}: *there is no difference between the density of future bugs of files from groups GP_i and NGP_i.*

Hypothesis H^1_{02} (respectively H^2_{02}) is about the probability of bugs (respectively the density of future bugs) in files edited following the pattern P_i. H^1_{02} and H^2_{02} are two-tailed since they 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 *odds ratio* to test H^1_{02}. We perform a Wilcoxon rank sum test for H^2_{02}.

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 five developers is more at risk for conflicting changes than the file edited by two developers, 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^1_i and GP^2_i. GP^1_i contains



(1) Concurrent (2) Parallel (3) Extended (4) Interrupted

Figure 4: Odds ratio between files that are not involved in patterns and files that are involved in editing patterns.

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_{02}^3: *the proportion of files exhibiting at least one bug is the same for* NGP_i, GP_i^1 *and* GP_i^2.
H_{02}^4: *there is no difference between the density of future bugs of files from groups* NGP_i, GP_i^1 *and* GP_i^2.

Similar to H_{02}^1, H_{02}^3 is about the probability of bugs in files. Hence, we use the Fisher's exact test and compute the *odds ratio* to test H_{02}^3. H_{02}^4 like H_{02}^2 is about the density of future bugs in files; we perform the Kruskal-Wallis rank sum test for H_{02}^4. All the tests are performed using the 5% level (*i.e.*, p-value < 0.05).

Findings. Among the four patterns, the concurrent editing pattern is the most risky. The likelihood of bugs in a file edited following the concurrent editing pattern is higher compared to files edited following one of the other three patterns. Figure 4 shows ORs values for the four patterns. A file with concurrent edits is 2.14 times more likely to experience a future bug than a file that was never involved in a concurrent edit (Figure 4 (1)). The OR value is 2.14 and the Fisher's exact test was statistically significant. Therefore, we reject H_{02}^1 for the concurrent editing pattern. Files involved in concurrent edits have on average 2.46 times more future bugs than files that were never edited concurrently. The Wilcoxon rank sum test was statistically significant. Hence, we also reject H_{02}^2. We could not reject either H_{02}^3 or H_{02}^4 for the concurrent editing pattern. However, results from Table III suggests that 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".

Overall, a file edited in parallel with other files by the same developer is 1.94 **times (shown in Figure 4 (2)) more likely to experience a bug than a file that was always edited individually throughout its revision history. The number of files involved in the parallel edits plays a significant role in increasing the risk for bugs (see Table III).** We obtained an OR value of 1.94 for the parallel editing pattern and the Fisher's exact test was statistically significant for H_{02}^1. The Wilcoxon rank sum test was also statistically significant for H_{02}^2. Files edited in parallel have on average 1.67 times more future bugs than files that were always edited individually. However, as shown in Table III, 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 only when the number of files edited in parallel (*i.e.*, the *level*) is above the third quartile. We conclude that although the parallel editing of files is risky in general, all parallel editings are not equally risky. Development teams can chose to monitor only files that were edited in parallel with too many other files. Quality assurance teams should advice developers against editing too many files in parallel.

Table III: Relation between the level of involvement in a pattern and the risk of bugs (* indicates that the test was statistically significant, *i.e.*, $p\text{-}value < 0.01$)

Pattern	level $\leq$ third quartile		level $>$ third quartile	
	OR	Average bug density	OR	Average bug density
Concurrent	2.393*	1.417*	1.603	0.824
Parallel	1.343	1.446	3.857*	2.606*
Extended	1.033	0.660	1.877*	1.135*
Interrupted	1.244	0.815	1.752*	0.740*

The extended editing pattern increases the risk of bugs in files. Indeed, files with edit time spans on average greater than the third quartile are 1.86 **times more likely to experience a future bug than other files, as illustrated in Figure 4 (3).** The Fisher's exact test was statistically significant. We reject H_{02}^1 for the extended editing pattern. The Wilcoxon rank sum test was also statistically significant for H_{02}^2. We then reject H_{02}^2 for the extended editing pattern. Files edited following the extended editing pattern have on average 2.28 times more future bugs than files that were never involved in an extended editing. When the edit time span of a file is on average lower than the third quartile, Table III shows that the risk of bug is not significantly different from those of files with edit time spans lower than the median.

The interrupted editing pattern increases the risk of

bugs in files. In fact, files edited with interruption time on average greater than the third quartile are 1.62 **times more likely to experience a future bug than other files, as illustrated in Figure 4 (3).** The Fisher's exact test was statistically significant. Therefore, we reject H_{02}^1 for the interrupted editing pattern. The Wilcoxon rank sum test was also statistically significant for H_{02}^2. We then reject H_{02}^2 for the interrupted editing pattern. Files that were edited following the interrupted editing pattern have on average 2.1 times more future bugs than files that were not involved in an interrupted editing. When the interruption time of a file is on average lower than the third quartile, Table III shows that the risk for bugs in the file is not significantly different from those of files with interruption times lower than the median. Managers should consider taking measures to avoid the frequent interruption of developers.

RQ3: Do interactions among file editing patterns lead to more bugs?

Motivation. In **RQ1**, we found that a large number of files from our systems (*i.e.*, 949 files) were 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 **RQ2**, 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.
Approach. For each file, we use the metrics described in Section IV-B3 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 II, 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_{03}^1: *the proportion of files exhibiting at least one future bug does not differ between the groups* GP_i, $PInteract_i$, *and* $GNoP$.
H_{03}^2: *there is no difference between the density of future bugs of files from groups* GP_i, $PInteract_i$, *and* $GNoP$.

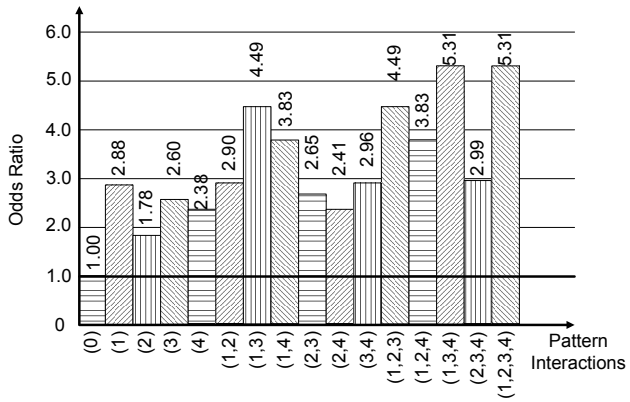


Figure 5: Odds ratios of future bugs in files from the 16 groups listed in Table II

Similar to **RQ2**, hypothesis H_{03}^1 (respectively H_{03}^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 *odds ratio* to test H_{03}^1. We perform a Kruskal-Wallis rank sum test for H_{03}^2. We test H_{03}^1 and H_{03}^2 using the 5% level (*i.e.*, p-value < 0.05).
Findings. 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. In fact, when the concurrent editing pattern, the extended editing pattern, and the interrupted editing pattern are followed all together during modifications of a file, the OR value is the highest, as illustrated in Figure 5. Also, whenever 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 increased (*i.e.*, the OR is increased). The Fisher's exact test was statistically significant. Therefore, we reject H_{03}^1. The Kruskal-Wallis rank sum test for H_{03}^2 was also statistically significant. We also reject H_{03}^2.

For all pattern interactions but one, the density of future bugs is increased when a file is edited following more than one editing pattern. The only exception is when a file is edited following both the Parallel and the Interrupted editing patterns. We found that files edited following both the Parallel and the Interrupted editing patterns have less future bugs than files edited following only either the Parallel or the Interrupted editing pattern. However, the difference was not statistically significant. The number of future bugs in a file edited by developers following more than one editing patterns can go as high as 1.6 times the average number of future bugs in files edited following a single editing pattern.

VI. Threats to validity

We now discuss the threats to validity of our study following common guidelines provided in [12].

Construct validity threats concern the relation between

theory and observation. Our construct validity threats are mainly due to 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 rely on the judgement of one of the authors during the manual identification of bug fixing change logs. Therefore, because of the subjective nature of the task, some change logs might have been counted wrongly. However, we compared our bug data with bug data from the study of Lee *et al.* [6], which are publicly available. We found the two data sets to be consistent.

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 process of the three software systems which 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.

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 for the potential confounding effect of size and code churns.

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 available online[4].

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.

VII. 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.

A. Analysis of File editing patterns

To the best of our knowledge, this study is the first attempt to empirically quantify the impact of concurrent, parallel, extended, and interrupted file editing patterns on software bug-proneness.

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 [2] and Crystal [3], have been proposed to warn developers about potential file editing conflicts. Treude and Storey [4], who investigated 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.* [13] investigated file editing patterns in a large telecommunication software system and found that about 50% of the files were modified consecutively by more than one developer in the period between two releases of the software. They did not study the concurrent editing of files, but nevertheless reported that files edited by multiple developers were at a higher risk for bugs. Staudenmayer *et al.* [14] studied concurrent changes at module level in another telecommunication software system and observed that although a concurrent modification of modules can shorten the development time of a software system, developers often experience conflicting changes. D'Ambros *et al.* [15] investigated the relation between change coupling and bugs through an analysis of files frequently committed together. They concluded 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 necessary 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 do not have committing rights. 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 are able to identify file edits that happened at the same time thanks to the rich developer's interaction logs collected by the development teams of the three Eclipse projects, using the Mylyn tool. Parnin and Rugaber [16] investigated developers' interruptions during software development tasks and reported on the strategies adopted by developers to successfully resume a task after an interruption. They did not assess the likelihood that a bug would be introduced because of frequent interruptions. Lee *et al.* [6], propose to use the time spent by developers on tasks to predict future bugs in the files involved in the tasks. In this work, we empirically quantify the likelihood of having bugs in files edited by developers following respectively the concurrent,

[4]http://tinyurl.com/fileeditingpatternstudy-zip

parallel, extended and interrupted editing patterns.

B. Bug Prediction

Several studies have investigated the use of metrics to predict the location of future bugs in software systems. For example, Khoshgoftaar *et al.* [17] report good results from a combination of code metrics and knowledge from problem reporting databases for bug predictions. Moser *et al.* [18] 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.* [19] 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 defects. The resulting model can predict whether a source file has a defect with 99% accuracy. Nagappan and Ball [20] show that relative code churn metrics are good predictors of bug density in systems. Askari and Holt [21] provide a list of mathematical models to predict where the next bugs are likely to occur. In this work we investigate the possibility of using four file editing patterns to identify future location of bugs in systems. We propose metrics to identify the patterns. Although we do not build prediction models, we analyze the relation between the occurrence of our file editing patterns and future bugs. We also analyze the interaction between the editing patterns.

VIII. Conclusion

In this paper, we analyzed the developers' interaction logs of three open source software systems, Mylyn, Eclipse Platform, and Eclipse PDE, and identified four file editing patterns. We investigated the potential impact of the editing patterns on software quality.

Our results show that concurrent, parallel, extended, and interrupted file editing patterns are frequently followed by developers during development and maintenance activities. Whenever one of the four editing patterns is followed during the modifications of a file, the risk of future bugs in the file increases. Among the four patterns, the concurrent editing pattern is most risky. Developers and managers should also be cautious when one file is edited in parallel with too many other files. Also, development teams should avoid having developers spending too long time editing one file. They should also avoid interrupting their developers frequently.

We also observed that when more than one editing patterns are followed by one or many developers during the editing of a file, the risk of future bugs in the file increases further. The number of future bugs in a file edited by developers following more than one editing patterns can go as high as 1.6 times the average number of future bugs in files edited following a single editing pattern.

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 patterns occurrences and interactions.

References

[1] N. I. of Standards & Technology, "The economic impacts of inadequate infrastructure for software testing," May 2002, uS Dept of Commerce.

[2] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: discovering and exploiting relationships in software repositories," in *ACM/IEEE 32nd International Conference on Software Engineering*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 125–134.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178.

[4] C. Treude and M. Storey, "Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds," in *ACM/IEEE 32nd International Conference on Software Engineering*, ser. ICSE'10, vol. 1, may 2010, pp. 365 –374.

[5] Mylyn, "http://wiki.eclipse.org/mylyn_integrator_reference."

[6] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro interaction metrics for defect prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 311–321.

[7] F. Zhang, F. Khomh, Y. Zou, and A. Hassan, "An empirical study on factors impacting bug fixing time," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, oct. 2012.

[8] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan, "Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project," in *Proceedings of the 2010 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:10.

[9] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[10] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *International Workshop on Predictor Models in Software Engineering*, ser. PROMISE'07, may 2007, p. 9.

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

[12] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[13] D. Perry, H. Siy, and L. Votta, "Parallel changes in large scale software development: an observational case study," in *Proceedings of the 20th International Conference on Software Engineering*, ser. ICSE'98, apr 1998, pp. 251 –260.

[14] N. Staudenmayer, T. Graves, and D. Perry, "Adapting to a new environment: how a legacy software organization copes with volatility and change," in *5th International Product Development Conference*, ser. IPDC'98, 1998.

[15] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *16th Working Conference on Reverse Engineering*, ser. WCRE'09, oct. 2009, pp. 135 –144.

[16] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," in *IEEE 17th International Conference on Program Comprehension*, ser. ICPC'09, may 2009, pp. 80 –89.

[17] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl, "Data mining for predictors of software quality," *International Journal of Software Engineering and Knowledge Engineering*, vol. 9, no. 5, 1999.

[18] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 181–190.

[19] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *IWPSE '07: Ninth international workshop on Principles of software evolution*. NY, USA: ACM, 2007, pp. 11–18.

[20] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. NY, USA: ACM, 2005, pp. 284–292.

[21] M. Askari and R. Holt, "Information theoretic evaluation of change prediction models for large-scale software," in *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. NY, USA: ACM, 2006, pp. 126–132.