The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM)
ISSN: 2602-3199

The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), 2021

Volume 16, Pages 182-190

IConTES 2021: International Conference on Technology, Engineering and Science

Do API-Migration Changes Introduce New Bugs?

Omar MEQDADI
Jordan University

Shadi ALJAWARNEH
Jordan University

Muneer BANI YASSEIN
Jordan University

Abstract: Software quality is broadly dependent on the use of dependent platforms, compilers, and APIs. This research reports a case study exploring the risk of API-migration activities in the regard of bug-introducing changes and software maintenance quality. The study involves screening tens of thousands of commits for six large C++ open source systems to identify bug-introducing commits caused by undertaking adaptive maintenance tasks through using traditional heuristic approaches. The obtained results show that 14.5% to 22.2% of examined adaptive commits enclose buggy code changes and so developers have to consider the potential risk of introducing new bugs after undertaking API-migration practices. Moreover, from investigating the bug fixing activities made by API-migration tasks, we provide a demonstration that typically these fixing activities do not cause further bugs and hence are safe undertakings. We feel that this work has developed a data set that will be used for constructing approaches to identify, characterize, and minimize potential adaptive maintenance practices that introduce bugs into a software system.

Keywords: Bug-introducing commits, Adaptive maintenance, API migration, Cchange commits

Introduction

When developers evolve software systems, they frequently perform *adaptive maintenance* activities as a process to cope with changes of underlying APIs, compilers, operating systems, and dependent platforms (Swanson,1976; Schach et. al., 2003). Examples of adaptive maintenance comprise of migrating a software system to work on a new version of the underlying API (Meqdadi et. al., 2013; Meqdadi et. al., 2019).

Normally, during the evolution activities, developers perform changes that are clean (e.g., containing no bugs or defects). Unfortunately, developers sometimes unintended make improper code statements that introduce bugs into software systems (Kim et. al., 2008; Śliwerski et. al., 2005). These improper changes are termed *bug-introducing changes*. A bug in the source code negatively impacts the quality of software systems through leading the execution of systems to an undesired external behavior (Kim et. al., 2008). Sometime later, the developer who realize this behavior records the bug in a bug report that be submitted to bug tracking systems.

Recently, a rich number of studies have been undertaken to better understanding the phenomenon of bug-introducing changes by mining history repositories of software systems (Hassan et. al., 2005; Eyolfson et. al., 2011; Posnett et. al., 2013). Kim et.al (Kim et. al., 2008) used repository commits to develop a machine learning classifier that determine whether a new software change is buggy (e.g., cause a new bug) or clean change. Shedding the light on the impact of change-proneness APIs in fault-proneness of software systems have been made previously. After studying 7,097 Android applications, Linares-Vásquez et.al (Linares-Vásquez et. al., 2013) Found that making use of a change-proneness API would negatively affect the quality of a software system. Zibran et. al. (Zibran et. al., 2011) manually examined 1,513 bug-reports extracted from bug tracking systems of different open-source projects. The study shows that 37.14% of the examined reports were relevant to the issues of API usability. However, there is no general analysis previously made on how potentially bug prone is a system after undertaking necessary adaptive maintenance changes. Unfortunately, there is no attempt in the literature to figure out whether API-migration changes are clean or buggy.

In this regard, we explore whether API-migration changes introduce new bugs into open source systems. The motivation of this work is to gain the attention of developers to take steps to focus on the possibility of introducing further bugs because of API-migration practices. The contribution of our study is to develop a data set that can be used for further investigations that interest in exploring methods to recognize, characterize, and minimize potential bug-introducing changes that might be caused by practicing adaptive maintenance activities. For instance, this study draws observations that directly guide developers involved in quality management and regression testing processes relevant to adaptive maintenance.

Therefore, we performed a case study of six C++ open-source systems that have a long evolutionary history comprises of successful API migration activities. We analyzed the history repositories and bug tracking systems used by our subject systems to determine adaptive commits that introduced bugs after accomplishing API-migration tasks. Next, we followed up our previous work in (Meqdadi et. al., 2019) and discovered whether bug-fixing activities undertaken through API-migration tasks are clean changes or lead to new bugs.

Research Questions

In our previous study undertaken in (Meqdadi et. al., 2019), we discussed the beneficial of undertaking API-migration tasks with respect to bug fixing activities. On the contrast, this study explores the risk of introducing new bugs into software systems after accomplishing demanded adaptive maintenance changes. Our hypothesis is that API-migration tasks can play an important role in the quality of software systems positively through bug fixing and negatively through introducing further bugs. Specifically, we address two main questions concerning bug-introducing changes.

RQ1) - Is there a new bug injected into system as a result of accomplishing an API-migration task?

To answer this question, we need to identify the set of bug-introducing commits of each system. Then, we have to cross intersection the list of bug-introducing commits and the list of recognized adaptive commits. The resultant intersection list with non-zero value of size means that API-migration changes might introduce bugs into systems. We view this question as an important investigation since it can be used to identify potential risks of adaptive maintenance practices. Obtained dataset from this question would serve as a baseline that is of great interest for future work focused on further studying the problematic aspects (e.g., bug-introduction) of API-migrations.

RQ2) - Is a bug fixing change through an API-migration task a clean change?

Clearly, the collection of adaptive commits that are simultaneously identified as a bug fixing and a bug-introducing commit is used to answer this question. Given this trend of adaptive commits means that a specific API-migration change might fix a bug and in conjunction introduce another into a software system. We consider this question as an important demand in order to help developers to realize whether fixing current bugs through API-migration practices is a safe process.

Study Setup

To undertake our case study, we selected diverse open source systems written in C++. The main principle behind electing our subject systems is that they all have active API-migration tasks that were undertaken successfully. In this study, for each subject system, we have identified bug-fixing commits along with relevant bug-introducing changes. Below, we present details behind our study approaches.

Studied Systems

Previously, we have identified and examined the adaptive maintenance tasks in the context of API-migration changes for several open source systems (Meqdadi et. al., 2013; Meqdadi et. al., 2019). In this research, we have chosen our dataset from six open source systems namely: four KDE packages: *KOffice*, *Extragear/graphics,* the KDE editor *Kate,* and the visual database applications creator *Kexi*. Also, we studied the 3D graphics toolkit *OpenSceneGraph (OSG)* and the full text search tool *Recoll*.

The studied systems were elected because of their active evolutionary history that include effective adaptive maintenance tasks in the context of API migrations changes. As well, we have previously inspected the undertaken adaptive maintenance changes of these selected systems in our prior research (Meqdadi et. al., 2013; Meqdadi et. al., 2019; Meqdadi et. al., 2020). For instance, the API-migration tasks of the KOffice, Extragear/graphics, and OSG systems were analyzed in (Meqdadi et. al., 2013), where their adaptive commits had been allocated through the manually reading of messages in the log files. If a message of a commit encloses key terms of API-migration tasks (i.e., APIs interfaces or language features), then the commit is considered as an adaptive commit. Similarly, the adaptive commits of the remaining systems were allocated and examined in (Meqdadi et. al., 2019) and (Meqdadi et. al., 2020), where the study proposed a machine learning classifier to label the commits of an evolution history into either one of two classes, namely adaptive and non-adaptive commits.

Next, to develop our dataset, we have used the *GNU UNIX DIFF* utility to recover the added, modified, and removed source code lines in every adaptive commit of examined projects. Alike, we determined change hunks relevant to each adaptive commit. A change hunk is defined as a continuous set of code lines that are affected by a commit along with contextual unchanged lines (Alali et. al., 2008).

Table 1 has the main information regarding our dataset. For instance, the table demonstrates the undertaken API-migration task, the examination period of time, total number of commits along with the number of adaptive commits, and the number of files and hunks touched by the captured adaptive commits of each subject system.

Identification of Bug Fixing Commits

Similar to our previous work (Meqdadi et. al., 2019), to identify whether a given change commit has fixed a bug, we employed the traditional heuristics approach (Fischer et. al., 2003; Bird et. al., 2009) that successfully and widely used for recognizing the set of bug-fixing commits from a history repository (Kim et. al., 2006). This traditional method is based on searching for valid bug reference IDs and key words (i.e., fixed, crash, or bug) within the messages of the change log file. Next, for each extracted bug's ID, we retrieve the associated bug report from the bug tracking system of the examined system. Although a commit could fix more than one bug, in this study we assume that each bug is only being fixed by one fixing commit. This aforementioned assumption represents one of the threats to validity of this work.

Identification of Bug-Introducing Commits

Our research study aims at understanding whether adaptive changes might introduce new bugs in open source systems. Therefore, we basically need to recognize the set of the adaptive commits that have introduced bugs, and then figure whether these problematic commits also have fixed earlier bugs.

The bug tracking systems are valuable resources regarding bugs. However, the major problem is that these tracking systems have no insight on when and why a bug was introduced into the code, who injected it, and where it occurred (Kim et. al., 2006; Kim et. al., 2008). That is, there is not enough information to identify bug-introducing changes immediately from bug tracking systems. Based on that, we need an approach or a tool that would help us to recognize bug-introducing changes.

Consequently, we have surveyed several valuable prior studies in the literature that have focused on examining bug-introducing commits. We have leveraged the knowledge regarding numerous known recovering methods of such problematic commits. Particularly, for our investigating experiments, we have used the approach that was proposed and effectively evaluated in (Kim et. al., 2006; Kim et. al., 2008). Initially, the approach determines bug-introducing changes from the bug-fixing commits that are recovered using the approach discussed in the prior section. Next, for each recovered fixing commit R, the approach runs the *DIFF* tool to detect all hunks in the preceding commit of R (e.g., R-1) that have been changed to fix the bug.

That is, the approach follows the assumption that deleted or modified statements in each changed hunk represent the candidate source code of relevant bug. After that, the approach tracks down the origins of source code lines in the potential buggy hunks by using the built-in *annotate* command of the version history. This feature returns the most recent commit in which a line was changed, which would be considered as the candidate bug-introducing commit. For accurateness, the approach skips all source code lines that have been changed after the bug has been reported. Furthermore, the approach removes false positives and false negatives by using annotation graphs and also by ignoring non-semantic source code changes and outlier fixes. More details regarding this approach can be regained in (Kim et. al., 2006; Kim et. al., 2008).

Table 1. Subject open-source systems of our study

	KOffice	Extragear / graphics	OSG	Kexi	Kate	QuantLib
Adaptive Maintenance Task	Migrating to Qt4	Migrating to Qt4	Migrating to OpenGL 4.x	Migrating to Qt5.x	Migrating to Qt5.x	Migration to Visual C++ 2017
Time Period	1/1/2006 – 12/31/2010	1/1/2006 – 12/31/2010	1/1/2014 - 1/1/2017	7/7/2014 – 1/1/2018	1/1/2015 – 1/1/2018	1/1/2017– 1/1/2018
# Commits in the Log File	38980	26336	1984	3283	922	628
# Adaptive Commits	131 (0.3%)	219 (0.8%)	126 (6.35%)	161 (4.90%)	54 (5.9%)	59 (9.4%)
# Adaptive Change Files	858	910	491	682	186	206
# Adaptive Change Hunks	1138	1893	1521	2104	571	550

Results and Discussion

Figure 1 illustrates the entire workflow that we performed to achieve our results for each subject system. As shown, we must first collect the set of the adaptive commits that are considered as bug-fixing commits. Then, the set of all bug-fixing commits of each subject system are collected using the identification approach discusses previously. But, this approach retrieves all bug-fixing commits regardless their type (e.g., adaptive or non-adaptive). Therefore, we have categorized retrieved fixing commits as adaptive or non-adaptive using the results of our previous studies (Meqdadi et. al., 2013; Meqdadi et. al., 2019; Meqdadi et. al., 2020). Basically, the outcome of this phase is the extraction of commits that fix bugs by performing API-migration tasks We refer to these interested adaptive commits as "*Adaptive-Bug-Fixing*" ones.

As the next step, we need to know commits that were probably induced bugs into each examined system. We have applied the identification approach of bug-introducing commits that presented previously, which depends on the annotation feature of history repositories. Again, this produces, for each subject system, a set of commits that are likely introduce bugs irrespective their type of maintenance (e.g., adaptive or non-adaptive). Also, it labels, for each bug-introducing commit, the potential buggy hunks that comprise bug-introducing code statements. To be precise, we manually examined all retrieved buggy code changes and we found they were all true candidates. For instance, no returned change was a comment line change, or a cosmetic change.

Essentially, we restricted our attention to buggy changes made by API-migration tasks. Therefore, we reused the categorization of adaptive commits performed in (Meqdadi et. al., 2013; Meqdadi et. al., 2019; Meqdadi et. al., 2020) to mark adaptive commits that would be described as bug-introducing commits. We will call these problematic commits as "*Adaptive-Bug-Introducing*" commits.

For instance, Figure 2 shows the commit #1177459 that was marked as a bug-fixing commit. From the developer message, we would observe that the adaptive maintenance task that ported the file *KoCsvImportDialog.cpp* from Qt3 to Qt4 would be considered as a bug-introducing change. After tracking down the origins of source code lines in the changed hunks of the commit #1177459 using the built-in *annotate* command, we found that the bug was originally injected because of some API incompatibilities resultant from using the modified version of class *QString* in Qt4(e.g., incompatibilities with item-based table view in the case of a default model). This using of modified *QString* was made by the adaptive commit #1132380. Thus, this adaptive commit is considered as a bug-introducing commit. The relevant buggy hunk made by this commit is shown in Figure 3.

Table 2 shows the results achieved after applying the identification approach of bug-introducing commits for each subject system. Also, the table reports the percentages of adaptive commits and change hunks that were recognized as bug-introducing. For instance, the percentages given in the second row were calculated with respect to the total number of adaptive commits while the percentages given in the third row were calculated with respect to the total number of adaptive change hunks given in Table 1.

Thereby, we could realize that *adaptive maintenance tasks (in the context of API-migration changes) might unawares introduce new bugs into open source systems*. For instance, for the OSG system, our observation is that nearly 19.0% of undertaken adaptive commits were buggy commits and caused injection of various bugs into the system. Furthermore, 14.9% of adaptively changed hunks of the OSG system involved buggy code statements.

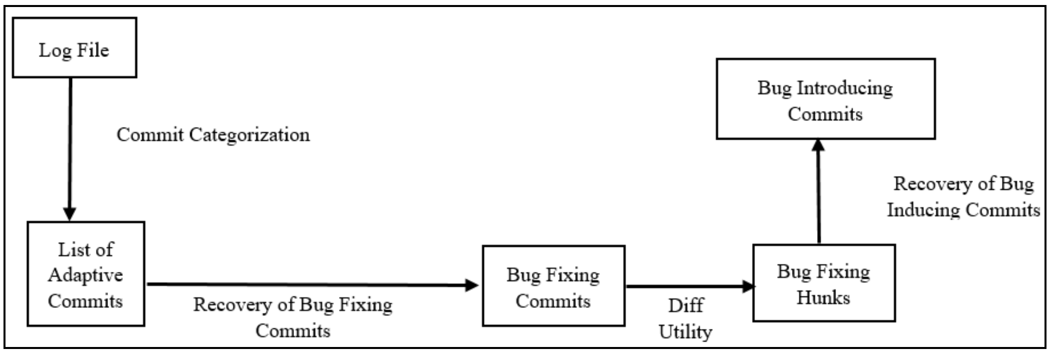
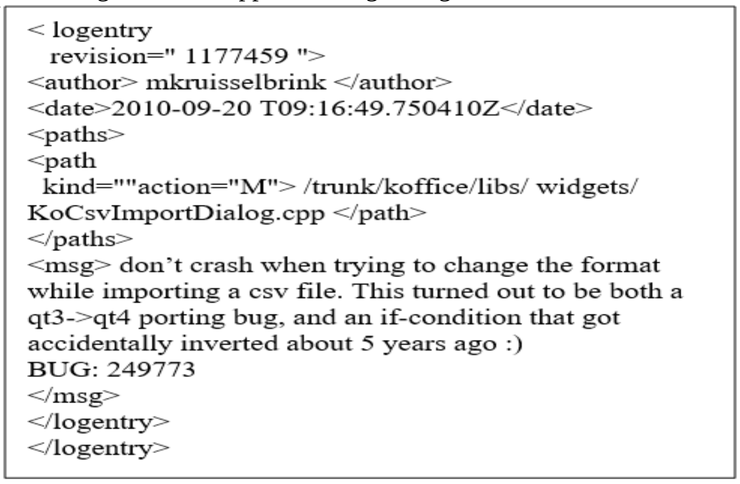
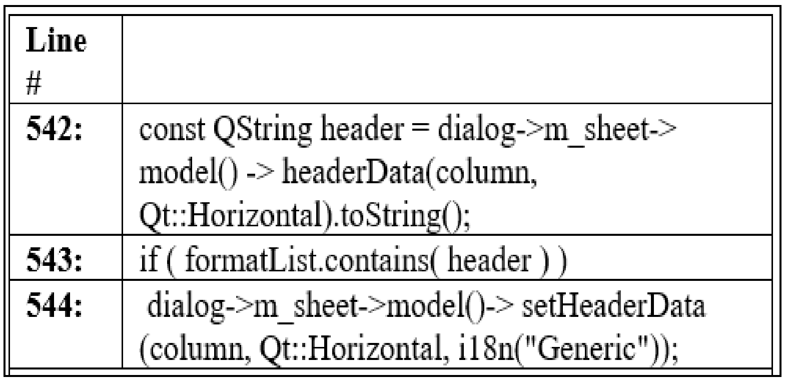


Figure 1. Workflow for study approach

Figure 2. A Snippet of a bug-fixing commit for Koffice


```
< logentry
  revision=" 1177459 ">
<author> mkruisselbrink </author>
<date>2010-09-20 T09:16:49.750410Z</date>
<paths>
<path
 kind=""action="M"> /trunk/koffice/libs/ widgets/
KoCsvImportDialog.cpp </path>
</paths>
<msg> don't crash when trying to change the format
while importing a csv file. This turned out to be both a
qt3->qt4 porting bug, and an if-condition that got
accidentally inverted about 5 years ago :)
BUG: 249773
</msg>
</logentry>
</logentry>
```



Line #	
542:	const QString header = dialog->m_sheet-> model() -> headerData(column, Qt::Horizontal).toString();
543:	if (formatList.contains(header))
544:	dialog->m_sheet->model()-> setHeaderData (column, Qt::Horizontal, i18n("Generic"));

Figure 3. A Snippet of a buggy adaptive code hunk

Table 2. Statistics from identification of bug-introducing commits

	KOffice	Extragear / graphics	OSG	Kexi	Kate	QuantLib
# of Bug-Introducing Commits	6626	4161	409	575	147	130
# of Adaptive- Bug-Introducing Commits	19 (14.5%)	39 (17.8%)	24 (19.0%)	23 (14.3%)	12 (22.2%)	18 (20.3%)
#of Buggy Adaptive Change Hunks	221 (19.4%)	414 (21.9%)	226 (14.9%)	349 (16.6%)	118 (20.7%)	111 (20.2%)

The study performed in (Meqdadi et. al., 2013) uncovered the result that adaptive commits are usually system wide and large. On the other hand, obtained results of (Śliwerski et. al., 2005) show that typically large commits are being bug-introducing commits more than other commits. Given these two studies, we can explain why adaptive commits might introduce bugs into systems. It is hardly surprising that adaptive commits that involved of large changes have a possibility for inducing later fixes.

Although most adaptive commits being clean, this type of maintenance should receive certain attention throughout testing and validation tasks. Importantly, our study uncovered the main result that *it is particularly worthy to spend efforts in quality assurance after accomplishing specific adaptive maintenance task*. For instance, developers have to confirm that adaptive commits are carefully revised and tested since these commits have a potential for introducing later bugs. Essentially, our study is a valuable source of data for further examining of bugs being introduced by adaptive changes. It is of great interest for future work to get a more detailed picture regarding buggy adaptive changes to develop automated methods to characterize, allocate, and hence avoid potential risks from undertaking demanded adaptive maintenance practices.

Our previous work presented in (Meqdadi et. al., 2019) demonstrated that undertaking API-migration tasks are supportive for fixing a set of existing bugs in open source systems. In contrast to the previous study, after answering RQ1, it will be a paramount investigating to figure out whether the adaptive maintenance could be recommended as a safe approach to fix current bugs of a software system. That is, a fair question to ask now is, is a bug fixing change through an API-migration task a clean change? In order to answer this question, we further examined the sets of extracted *Adaptive-Bug-Introducing* and *Adaptive-Bug-Fixing* commits. We have defined a new classification of adaptive commits. The new category involves adaptive commits that had fixed a bug and simultaneously introduced another into a software system. We refer to this category as "*Adaptive-Bug-Fixing-Inducing*".

Certainly, the group of Adaptive-Bug-Fixing-Inducing commits represent the cross intersection between Adaptive-Bug-Fixing and Adaptive-Bug-Introducing sets. Table 3 shows the percentages of adaptive commits that are classified as Adaptive-Bug-Fixing-Inducing for each system. These percentages were calculated with respect to the total number of Adaptive-Bug-Introducing commits given in Table 2. The results are surprising. Although adaptive changes might cause new bugs, the majority of bug fixing activities that undertaken throughout API-migration tasks did not inject new bugs into systems. That is, *fixing bugs through an API-migration task is typically a clean and safe with no risks of bugs*.

Table 3. Adaptive Bug-Fixing-Inducing commits

	KOffice	Extragear/ Graphics	OSG	Kexi	Kate	QuantLib
# of Adaptive Bug-Fixing-Inducing commits	1 (5.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (5.6%)

Threats to Validity

A first threat to the validity of our work is that our examined systems may not be representative for all software systems. For instance, our dataset was extracted from open source systems written in C++. Thus, we cannot assume all obtained results have same weight or can be generalized for other systems written in other than C++ or these results are valid for commercial systems.

A construct threat to validity of our study is that there might be adaptive commits that we did not rely on in our case study. The group of adaptive commits that were examined in our study are the outcomes from the manual identification of adaptive commits performed in (Meqdadi et. al., 2013; Meqdadi et. al., 2019; Meqdadi et. al., 2020) that may comprise false positives and false negatives. Moreover, the presented case study here is based on the assumption that all adaptive commits being used in the undertaken experiments embraces only adaptive changes. However, developers might group both adaptive and non-adaptive changes into one version history commit.

We followed the traditional heuristics approach to identify bug fixing commits. However, we do not argue that this approach is without faults and consequently might affect the accuracy and correctness of our findings. Moreover, there is no standard how developers commit changes. Hence, some bug-fixing tasks might be accomplished through several commits. Another internal threat to validity is that we have depend on the approach proposed in (Kim et. al., 2006; Kim et. al., 2008) to obtain the set of bug-introducing commits. This approach has several limitations and so we believe that the using another accurate approach to find missing bug-introducing commits is essential. This is a one of our future plans.

Related Works

Similar to our study, lots of studies have been proposed to investigate the impact of software maintenance tasks on bug fixing activities. Kim et al. (Kim et. al., 2011) examined the role of API-level refactoring in fixing bugs for several JAVA projects such as Eclipse, JEdit, and Columba systems. They found that there are more bug fixes after finishing API-level refactoring activities. Also their results show that developers need less time to fix a bug after submitting API-level refactoring. The impact of adding new features and other code improvements with respect to bug proneness was reviewed by Posnett et al. (Posnett et. al., 2011). Their results show that code improvements might positively affect bug-fixing activities for some symptoms but could negatively influence fixing activities for other systems.

The influence of underlying API on software quality was investigated through numerous earlier studies. In (Linares-Vásquez et. al., 2013), the authors examined thousands of android applications to analyze the role of underlying API changes on software quality of a software system in terms of fault proneness. The obtained results illustrated that instability of underlying API negatively impact the quality of software projects. Similarly, Zibran et.al (Zibran et.al, 2011) found that a significant portion of reported bugs were originally related to the API usability and correctness. A case study of Apache projects was performed by Mileva et al. (Mileva et. al., 2009) to investigate the role of migration of code libraries on bug fixing efforts. The main finding of the study is that developers usually migrate back to earlier version of a library to ease the fixing efforts. Even though these studies have focused on the role of API correctness on the quality of software systems, the risk of API-migration in terms of bug-introducing changes was not considered in the literature.

There is a rich literature for examining the characteristics of bug-introducing changes using project histories. Kim et al. (Kim et. al., 2008) developed a machine learning classifier to classify whether an undertaking maintenance change is clean or buggy. Their classifier used several features of a committed change to determine if the change might introduce bugs in the future, such as added and deleted delta, author of the change, and the complexity of the modified files. The performance of the classifier was evaluated across 12 open source projects. Tufano et al. (Tufano et. al., 2017) performed an empirical study to characterize fix-inducing commits using several developers related factors. The studied factors are coherence of the commit, the experience level of developers, and the interfering changes made by other developers. The main finding of the study is that bug-introducing changes were made by more experienced developers. Also, the study shows that bug-introducing commits are usually less coherent and have more past interfering changes when compared to other commits. Rahman et. al. (Rahman et. al., 2011) studied version control systems to examine the impact of the developer experience and ownership on the possibility of introducing bugs after committing a maintenance change. The main observation is that specialized experience developers have made less defects in a software system when compared with general experience developers. The main characteristics of fix-inducing changes were investigated for ECLIPSE and MOZILLA projects in (Śliwerski et. al., 2005). The results show that that larger maintenance changes are more likely to induce future fixes. Furthermore, bug fix changes are likely to induce a future change than regular code enhancements. Despite, rather than general examination of fix-inducing changes, our study directly investigates these risky changes relevant to adaptive maintenance activities in the context of API-migration.

To the best of our knowledge, our work is the first study examines the risks of API-migration activities in terms of bug-introducing changes. There is no other work underwent previously had address this research area.

Conclusion

This work presents a case study of several C++ open source systems to determine whether adaptive maintenance changes (e.g., in the context of API-migration practices) lead to introduce new bugs into a software system. The study is based on the data obtained from software repositories of subject systems and through using traditional heuristic approaches to identify bug fixing and introducing code changes relevant to maintenance activities.

The study uncovered the main result that API-migration changes occasionally introduce new bugs into open source systems. For instance, our results show that 14.5% to 22.2% of studied adaptive commits were identified as bug-introducing commits. Thus, the observation that was obtained indicates that developers have to consider the potential risk of introducing new bugs after performing API-migration practices. On the other hand, a set of investigations was made focusing on whether bug fixing activities undertaken through API-migrations introduced new bugs. Our obtained results demonstrated that fixing current bugs by performing a specific API-migration practices is a safe process.

Our study developed a data set that will be used for further investigations of buggy changes relevant to adaptive maintenance. We expect that our future works will center on a deeply understanding of bugs that are being introduced by adaptive changes. We plan for a deep analysis of change features to identify common causality and patterns of adaptive changes that introduce bugs. We would explore the possibility of using machine learning algorithms to develop a model that automatically classify adaptive changes as buggy or clean immediately upon the accomplishment of an API-migration task.

Scientific Ethics Declaration

The authors declare that the scientific ethical and legal responsibility of this article published in EPSTEM journal belongs to the authors.

References

Alali, A., Kagdi, H., & Maletic, J. I. (2008, June). What's a typical commit? a characterization of open source software repositories. In *2008 16th IEEE international conference on program comprehension* (pp. 182-191). IEEE.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009, August). Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 121-130).

Eyolfson, J., Tan, L., & Lam, P. (2011, May). Do time of day and developer experience affect commit bugginess?. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (pp. 153-162).

Fischer, M., Pinzger, M., & Gall, H. (2003, September). Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (pp. 23-32). IEEE.

Hassan, A. E., & Holt, R. C. (2005, September). The top ten list: Dynamic fault prediction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (pp. 263-272). IEEE.

Kim, M., Cai, D., & Kim, S. (2011, May). An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 151-160).

Kim, S., Whitehead, E. J., & Zhang, Y. (2008). Classifying software changes: clean or buggy?. *IEEE Transactions on Software Engineering, 34*(2), 181-196.

Kim, S., Zimmermann, T., Pan, K., & James Jr, E. (2006, September). Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)* (pp. 81-90). IEEE.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., & Poshyvanyk, D. (2013, August). Api change and fault proneness: A threat to the success of android apps. In *Proceedings Of The 2013 9th Joint Meeting On Foundations of Software Engineering* (pp. 477-487).

Meqdadi, O., & Aljawarneh, S. (2020). A study of code change patterns for adaptive maintenance with AST analysis. *International Journal of Electrical and Computer Engineering, 10*(3), 2719-2733.

Meqdadi, O., & Aljawarneh, S. (2019, December). Bug types fixed by api-migration: a case study. In *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems* (pp. 1-7).

Meqdadi, O., Alhindawi, N., Alsakran, J., Saifan, A., & Migdadi, H. (2019). Mining software repositories for adaptive change commits using machine learning techniques. *Information and Software Technology*, *109*, 80-91.

Meqdadi, O., Alhindawi, N., Maletic, J.I., and Collard, M.L. (2013). Understanding large-scale adaptive changes from version histories: a case study. *In Proceedings of the 29th IEEE International Conference on Software Maintenance*, *ERA Track*, (pp. 22-28).

Mileva, Y. M., Dallmeier, V., Burger, M., & Zeller, A. (2009, August). Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops* (pp. 57-62).

Posnett, D., D'Souza, R., Devanbu, P., & Filkov, V. (2013, May). Dual ecological measures of focus in software development. In *2013 35th International Conference on Software Engineering (ICSE)* (pp. 452-461). IEEE.

Posnett, D., Hindle, A., & Devanbu, P. (2011, October). Got issues? do new features and code improvements affect defects?. In *2011 18th Working Conference on Reverse Engineering* (pp. 211-215). IEEE.

Rahman, F., & Devanbu, P. (2011, May). Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 491-500).

Schach, S. R., Jin, B. O., Yu, L., Heller, G. Z., & Offutt, J. (2003). Determining the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering*, *8*(4), 351-365.

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes?. *ACM sigsoft software engineering notes*, *30*(4), 1-5.

Swanson, E. B. (1976, October). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering* (pp. 492-497).

Tufano, M., Bavota, G., Poshyvanyk, D., Di Penta, M., Oliveto, R., & De Lucia, A. (2017). An empirical study on developer‐ related factors characterizing fix‐ inducing commits. *Journal of Software: Evolution and Process*, *29*(1), e1797.

Zibran, M. F., Eishita, F. Z., & Roy, C. K. (2011, October). Useful, but usable? factors affecting the usability of APIs. In *2011 18th Working Conference on Reverse Engineering* (pp. 151-155). IEEE.

Author Information

Omar MEQDADI
Jordan University of Science and Technology
Irbid, Jordan
Contact e-mail: *ommeqdadi@just.edu.jo*

Shadi ALJAWARNEH
Jordan University of Science and Technology
Irbid, Jordan

Muneer BANI YASSEIN
Jordan University of Science and Technology
Irbid, Jordan

To cite this article:

Meqdadi, O., Aljawarneh, S., & Bani Yassein, M. (2021). Do API-migration changes introduce new bugs?. *The Eurasia Proceedings of Science, Technology, Engineering & Mathematics (EPSTEM), 16*, 182-190.