Towards characterizing bug fixes through dependency-level changes in Apache Java open source projects

Article *in* Science China Information Sciences · July 2022

DOI: 10.1007/s11432-020-3317-2

CITATIONS
8

READS
154

7 authors, including:

Di Cui
Xidian University
27 PUBLICATIONS **446** CITATIONS
SEE PROFILE

Yuanfang Cai
Drexel University
129 PUBLICATIONS **4,268** CITATIONS
SEE PROFILE

Qinghua Zheng
Xi'an Jiaotong University
571 PUBLICATIONS **11,332** CITATIONS
SEE PROFILE

Yang Liu
Nanyang Technological University
635 PUBLICATIONS **21,360** CITATIONS
SEE PROFILE

SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2022, Vol. 65 172101:1–172101:19
https://doi.org/10.1007/s11432-020-3317-2

Towards characterizing bug fixes through dependency-level changes in Apache Java open source projects

Di CUI[1], Lingling FAN[3], Sen CHEN[4], Yuanfang CAI[5], Qinghua ZHENG[1], Yang LIU[6] & Ting LIU[2*]

[1]*School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an* 710049*, China;*
[2]*School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an* 710049*, China;*
[3]*College of Cyber Science, Nankai University, Tianjin* 300350*, China;*
[4]*College of Intelligence and Computing, Tianjin University, Tianjin* 300350*, China;*
[5]*Department of Computer Science, Drexel University, Philadelphia* 19104*, USA;*
[6]*School of Computer Science and Engineering, Nanyang Technological University, Singapore* 639798*, Singapore*

Received 24 August 2020/Revised 7 May 2021/Accepted 12 July 2021/Published online 10 June 2022

Abstract The complexity and diversity of bug fixes require developers to understand bug fixes from multiple perspectives in addition to fine-grained code changes. The dependencies among files in a software system are an important dimension to inform software quality. Recent studies have revealed that most bug-prone files are always architecturally connected with dependencies, and as one of the best practices in the industry, changes in dependencies should be avoided or carefully made during bug fixing. Hence, in this paper, we take the first attempt to understand bug fixes from the dependencies perspective, which can complement existing code change perspectives. Based on this new perspective, we conducted a systematic and comprehensive study on bug fixes collected from 157 Apache open source projects, involving 140456 bug reports and 182621 bug fixes in total. Our study results show that a relatively high proportion of bug fixes (30%) introduce dependency-level changes when fixing the corresponding 33% bugs. The bugs, whose fixes introduce dependency-level changes, have a strong correlation with high priority, large fixing churn, long fixing time, frequent bug reopening, and bug inducing. More importantly, patched files with dependency-level changes in their fixes, consume much more maintenance costs compared with those without these changes. We further summarized three representative patch patterns to explain the reasons for the increasing costs. Our study unveils useful findings based on qualitative and quantitative analysis and also provides new insights that might benefit existing bug prediction techniques. We release a large set of benchmarks and also implement a prototype tool to automatically detect dependency-level changes from bug fixes, which can warn developers and remind them to design a better fix.

Keywords empirical software engineering, software maintenance, software evolution, software architecture, software design

Citation Cui D, Fan L L, Chen S, et al. Towards characterizing bug fixes through dependency-level changes in Apache Java open source projects. Sci China Inf Sci, 2022, 65(7): 172101, https://doi.org/10.1007/s11432-020-3317-2

1 Introduction

Bug fixing is one of the most frequent activities in the software lifecycle [1]. To help developers understand how a bug fix is designed and its change impact, researchers have conducted a lot of empirical studies [2–4] and proposed numerous easy-to-use tools [5, 6]. Most of these studies analyzed a bug fix by deriving its fine-grained changes on code elements/references and measuring its local impact within a single source file (e.g., code entropy [7]). While they are effective in characterizing small pieces of localized code changes, they cannot provide a systematic view about the impact of a bug fix on dependencies among files of the whole software system.

* Corresponding author (email: tingliu@mail.xjtu.edu.cn)

The dependencies among files are an important dimension to inform software quality. Recent studies [8, 9] have empirically revealed that most bug-prone files are architecturally connected with dependencies, which inevitably incurs significant maintenance costs over time if developers keep introducing changes on dependencies during bug fixing. Hence, for the best practice in the industry [10], changes on dependencies can be avoided or be carefully designed during bug fixing. The dependencies require substantial effort to maintain and the bug-proneness can also be propagated to other files through modified dependencies, which may cause more issues and consequently result in ripple effects. If developers could realize the changes in file dependencies as early as possible, they could save more effort by designing a better fix. Therefore, we argue that it is necessary to understand and analyze bug fixes from one more perspective-dependencies. This new perspective should be viewed as complementary to existing ones, and it can be used together to help developers understand their bug fixes.

In this paper, we take the first attempt to conduct an empirical study on bug fixes from the perspective of dependencies. The study results will provide insights into the potential of existing techniques and useful suggestions for improving them by leveraging dependency attributes. Despite its significance, it is difficult to conduct such an empirical study due to the following challenges. (1) It is challenging to collect a large-scale dataset for carrying out such an empirical study. To ensure the generalization and credibility of the study result, our study requires a large number of bug fixes and the corresponding dependency-level changes[1)] on large-scale subjects rather than toy systems or selected ones, but no existing work maintains such a dataset. (2) It is challenging to design such a study. Dependency-level changes as a new dimension have never been studied in bug fixes by existing work. There also exists a gap between software dependencies and fine-grained code changes in bug fixes. A systematic analytical method is deserved to reveal the intrinsic relation between bug fixes and dependency-level changes.

To address these challenges, we collect 140456 fixed bug reports and 182621 commits for bug fixes from 157 most popular Apache open source projects. Based on these data, we conduct our study to characterize bug fixes from the dependency-level change perspective involving 11 types of software dependencies. We systematically analyze bug fixes with dependency-level changes from three aspects including ratio analysis, bug characteristic analysis, and patched file analysis. Several findings are presented as follows.

• **Ratio analysis.** Our results present that it is imperative to concentrate on the perspective of the dependency-level changes in bug fixes: on average, 30% of bug-fix commits[2)] contain dependency-level changes involving over 33% bugs since a bug-fix commit may fix several bugs and multiple commits may be needed to fix a single bug, which is prevalent in most subjects. These bug fixes are not concerned with existing state-of-the-art approaches, for they are involved in changes on multiple types of dependencies among files. This result encourages us to further explore these bugs/fixes related to dependency-level changes in depth.

• **Bug characteristic analysis.** Our results present that dependency-level changes are more likely to be introduced when fixing bugs with high priority, large fixing churn (lines of code), long fixing time, reopening again, and inducing the presence of new bugs. These results reveal the scenarios of bug fixes with dependency-level changes. It implies that we should conduct rigorous code review and testing on bug-fix commits by taking dependency-level changes into consideration before committing.

• **Patched file analysis.** Our results present that patched files with dependency-level changes capture a significant proportion and incur huge maintenance costs on fixing frequency and churn compared with patched files without dependency-level changes. The patched files with dependency-level changes interact through changed dependencies in multiple bug-fix commits with three representative patterns that incur repeated patches. Patched files in different patterns incur drastically different maintenance costs and files with the greater numbers of patterns consume more effort. These results reveal the difference in maintenance costs between patched files in bug fixes with/without dependency-level changes. It implies that developers should test or review patched files with dependency-level changes.

In summary, we make the following contributions:

• A novel dependency-level change perspective to understand and analyze bug fixes. It complements existing fine-grained code differences/reference perspectives by providing a systematic view about the impact of a bug fix on dependencies among files of the whole software system.

• A systematic and comprehensive study towards characterizing bug fixes from the dependency-level change perspective. Our study has revealed, for the first time, that there exist a significant proportion of

1) The modification of dependencies between source files, which are defined and illustrated in Subsection 2.3.

2) Code changes committed to fix bugs.

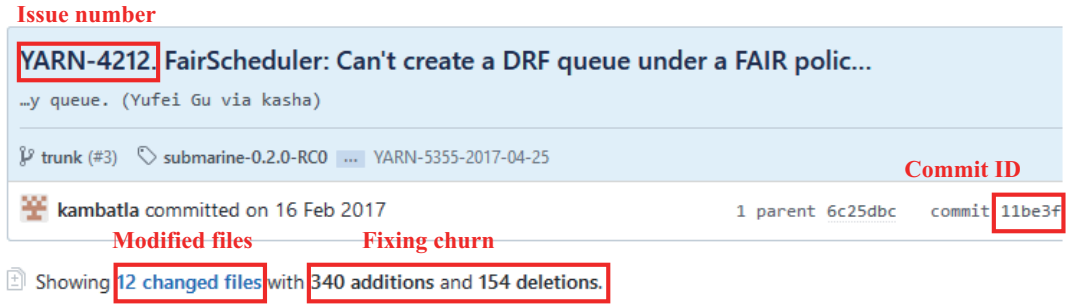

Figure 1 (Color online) The record of bug fix: commit 11be3f7 in Hadoop.

bugs and fixes introducing dependency-level changes, which advanced our understanding. Moreover, we also found that bug fixes with dependency-level changes consume much more maintenance efforts, from which we can learn that dependency-level changes should be carefully introduced during bug fixing.

• Our study enables several follow-up research directions with a large-scale benchmark, including 46164 bug reports and 54218 bug fixes involving dependency-level changes, and a reusable toolkit (`DependDiff`) to detect dependency-level changes from bug fixes. Our findings can also provide useful suggestions to improve existing approaches, such as change-level bug prediction and just-in-time bug prediction. The benchmarks are publicly available[3], which can assist developers to design a better fix.

2 Preliminary

In this section, we explain the terminologies used in our study.

2.1 Bug fix

Bug fix is an instance of code commit which is applied to fix bugs in version control systems such as SVN[4] and Git[5]. Typically, this commit adds code changes to source files and also reports textual description as a commit message. Figure 1 shows such a record of bug fix: commit 11be3f7[6] in Hadoop[7]. In this bug fix, we marked commit ID, issue number in the commit message. We also outlined the number of modified source files and fixing churn (i.e., lines of code).

2.2 Code dependency

Code dependency, according to the work of Cai et al. [9], can be modeled as a directed multi-graph, composed of a set of code elements and multiple types of dependencies between them. This model is also consistent with the definition of Bass et al. [11]: software modular structure contains multiple aspects, each type of dependencies is one of them. In our paper, we study 11 types of dependencies, including call (method invoke), cast (type cast), contain (variable/field definition), create (create an instance of a certain type), extend (parent-child relation), implement (implement interface), import (import header files), parameter (as a parameter of a method), return (returned type), throw (throw exceptions), and use (use or set variables). These dependency types are also widely and frequently considered by both academy and industry communities [8, 12]. Table 1 describes the description of each dependency type including definition, granularity, and example. In our paper, we use the file as the basic unit and aggregate code dependencies within files as the target.

2.3 Dependency-level change

Dependency-level change can be defined as a set of code changes that modify dependencies between the source files. An instance of dependency-level change between two source files can be further classified into the following four cases as illustrated in Figure 2.

3) https://github.com/cuidi34/DCBug.git.
4) https://subversion.apache.org.
5) https://git-scm.com.
6) https://github.com/apache/hadoop/commit/11be3f7.
7) https://hadoop.apache.org.

Table 1 Description of 11 types of code dependencies

Type	Definition	Granularity	Example
Import	Import is a relation between files, which indicates file A imports from file B.	File-level	Package a; import b.B;
Use	Use is a relation of an expression and the types or variables used by the expression.	Statement-level	void foo(){ int b=A.a*10;}
Call	Call is a relation of method invocation.	Statement-level	void foo(){ m.bar();}
Contain	Contain is a relation between code elements, which indicates element A contains element B. For example, a class contains a method, a method contains a variable.	Method-level Statement-level	void foo(){ B b;}
Create	Create is a relation of the function and objects it created.	Statement-level	void foo(){ A a=new A();}
Parameter	Parameter is a relation between method and its parameter.	Method-level Statement-level	void foo(B b){}
Return	Return is a relation of method and its return type.	Method-level Statement-level	B foo(){}
Cast	Cast is a relation of an expression and the casted type.	Statement-level	void foo(){ A a= (B) b;}
Extend	Extend means inheritance of OO language.	Class-level	class A extends B{ }
Implement	Implement is a relation between a function or class implementation, and its interface.	Class-level Method-level	class A implements I{ }
Throw	Throw is similar as return, it is a relation of method and its throw type.	Method-level Statement-level	void foo(){ throw new B();}

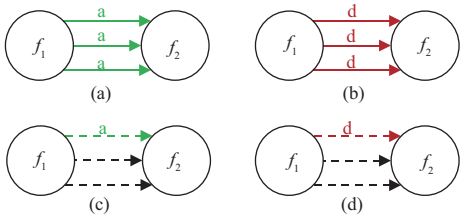

Figure 2 (Color online) Four cases of dependency-level changes. Nodes: files; Edges: dependencies; $\xrightarrow{a}$: adding dependencies; $\xrightarrow{d}$: deleting dependencies. (a) Case-1; (b) case-2; (c) case-3; (d) case-4.

• **Case-1.** Dependencies between two independent source files are first introduced, which is illustrated in Figure 2(a).

• **Case-2.** Dependencies between two connected source files are all deleted, which is illustrated in Figure 2(b).

• **Case-3.** Some but not all types of dependencies are introduced between two source files that already have dependencies, which is illustrated in Figure 2(c).

• **Case-4.** Some but not all types of dependencies are deleted between two source files connected with dependencies, which is illustrated in Figure 2(d).

We still use the commit 11be3f7 of Hadoop in Subsection 2.1 as an illustrative example. Figure 3 presents dependency-level changes in this commit, where each node represents a file and each edge represents the modified dependencies between files. As presented, this commit contains 6 instances of case-1 dependency-level changes, 4 instances of case-3 dependency-level changes, 5 instances of case-4 dependency-level changes. For example, the change between f_6 and f_7 is an instance of case-1 dependency-level change, since the dependency is firstly introduced between them. The change between f_6 and f_1 contains instances of both case-3 and case-4 dependency-level change, since a new type of dependency (i.e., parameter) is added and an existing type (i.e., use) is deleted.

2.4 Dependency change-related bug and fix

In our paper, we term the overlap between dependency-level changes and bug/bug fix as follows:

Dependency change-related fix, is a bug fix which introduces dependency-level changes.

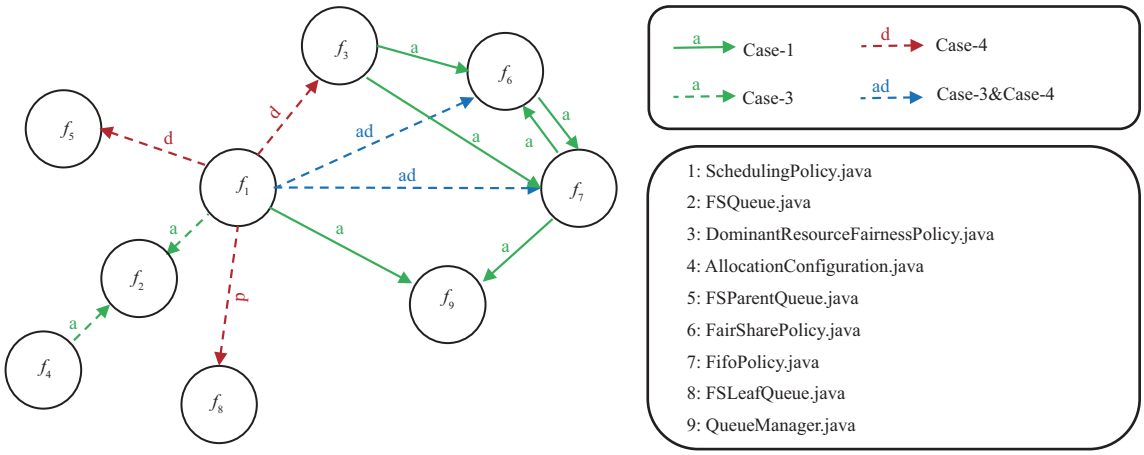

Figure 3 (Color online) Dependency-level changes of a bug fix: commit 11be3f7 in Hadoop. Nodes: files; $\xrightarrow{a}$: adding dependencies; $\xrightarrow{d}$: deleting dependencies; $\xrightarrow{ad}$: both adding and deleting dependencies.

Dependency change-related bug, is a bug where at least one of its fixes introduces dependency-level changes (i.e., dependency change-related fix). Note that one bug may be fixed through several commits.

For example, commit 11be3f7 illustrated in Subsections 2.1 and 2.2, is identified as a dependency change-related fix because this commit is a bug fix and also introduces dependency-level changes. Therefore, the bug: YARN-4212[8] fixed by this commit is regarded as a dependency change-related bug.

3 Overview

Figure 4 presents the overview of our study. We select 157 open source projects from Apache[9] as our subjects (Subsection 3.2), and gather dependency change-related fixes and bugs by (1) mining code repositories to collect bugs fixes and to distill dependency change-related fixes, and (2) crawling bug reports from issue repositories and matching related reports as dependency change-related bugs. Based on the collected data, we conduct an empirical study to understand bug fixes through dependency-level changes by answering three research questions in Section 4. This study enables several follow-up research detailed in Section 5.

3.1 Studied subjects

We choose Apache open source projects as our subjects for they are active in the open-source community and most of them are also frequently investigated in bug prediction/detection research [3, 4, 8, 13]. According to the ranking in OpenHub[10], we finally selected 157 Apache open source projects, varying in sizes, domains, and functionalities.

3.2 Data collection

Table 2 summarizes the statistics of the collected data, including bug reports, bug fixes, and dependency change-related bugs, and fixes. The details are explained as follows.

Bug report collection. In our study, we focus on bug reports that contain fixed bugs. To automatically crawl bug reports of the 157 projects, we implement a web crawler based on the Python library: jira-python[11] to crawl reports from JIRA[12]. For each bug report, our crawler gathers its issue number, create time, resolution time, priority, issue links, and tracking traces. These attributes are further used in Section 4. Finally, we collected 183428 bug reports of the 157 studied subjects from JIRA till November 2019, among which 140456 are fixed bugs.

8) https://issues.apache.org/jira/browse/YARN-4212.
9) http://www.apache.org.
10) https://www.openhub.net/.
11) https://github.com/pycontribs/jira.
12) http://issues.apache.org/jira.

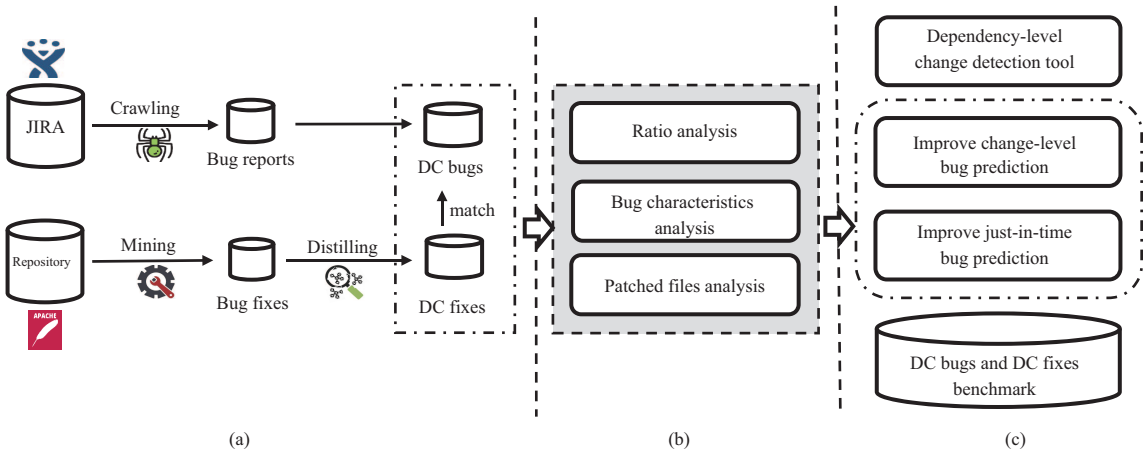

Figure 4 (Color online) Overview of our study and its applications. (DC bugs: dependency change-related bugs; DC fixes: dependency change-related fixes). (a) Data collection; (b) empirical study; (c) applications.

Table 2 Statistics of collected dataset

Bug reports		Bug fixes		Dependency changes	
#Bugs	183428	#Commits	970786	#Related bugs	**46164**
#Fixed bugs	**140456**	#Bug fixes	**182621**	#Related fixes	**54218**

Bug fix collection. Following the previous studies [3, 4, 14, 15], we collect bug fixes from commits by heuristically mapping the commit messages and issue numbers of studied bugs. In JIRA, the issue number is a unique identifier to each bug, following a "name-number" format where the name represents the project name. For example, as shown in Figure 1, this commit is identified as a bug fix in Hadoop for containing the issue number: YARN-4212. In total, we collect 182621 commits (i.e., bug fixes) which fixed the collected 140456 bugs.

Dependency change-related bugs and fixes collection. To identify whether a bug fix is a dependency change-related fix, for each bug fix, we take the following three steps (i.e., check out target files, extract dependencies among files, and obtain dependency-level changes).

• **Step1: check out target files.** In this step, given a bug fix, we first need to determine target files that have the potential to cause dependency-level changes. We heuristically use the committed files and their imported files as target files. The reason is that code changes in committed files may modify dependencies among them. As a result, we employ JGit[13], a git assistant tool, to automatically check out the two versions of these files before and after the fix.

• **Step2: extract dependencies among files.** In this step, we employ Depends[14], a state-of-the-art static analysis tool, to extract dependencies among target files before and after the fix. Depends can analyze 11 types of dependencies between files. It also supports incomplete analysis due to its implementation of partial program analysis (PPA)[15].

• **Step3: obtain dependency-level changes.** For extracted file dependencies before and after the fix, we deem them as a pair of graphs and compute their differences using graph edit distance algorithm [16]. Based on these collected differences, we further distill several instances of dependency-level changes. We implement a toolkit to label the detection results of dependency-level changes automatically. For each instance of two files, our toolkit first collects the set of dependency types between them before and after the commit as: DepSet_0 and DepSet_1. Next, our artifact compares these two sets, calculates its differences, and classifies them as follows:

(1) $|\text{DepSet}_0| = 0 \wedge |\text{DepSet}_1| > 0$: this case can be mapped to Figure 2(a).
(2) $|\text{DepSet}_1| = 0 \wedge |\text{DepSet}_0| > 0$: this case can be mapped to Figure 2(b).
(3) $|\text{DepSet}_1 - \text{DepSet}_0| > 0 \wedge |\text{DepSet}_0| \neq 0 \wedge |\text{DepSet}_1| \neq 0$: this case can be mapped to Figure 2(c).
(4) $|\text{DepSet}_0 - \text{DepSet}_1| > 0 \wedge |\text{DepSet}_0| \neq 0 \wedge |\text{DepSet}_1| \neq 0$: this case can be mapped to Figure 2(d).

Our toolkit iteratively and automatically runs each instance and gathers all the calculated differences

13) https://github.com/eclipse/jgit.
14) https://github.com/multilang-depends/depends.
15) http://www.sable.mcgill.ca/ppa.

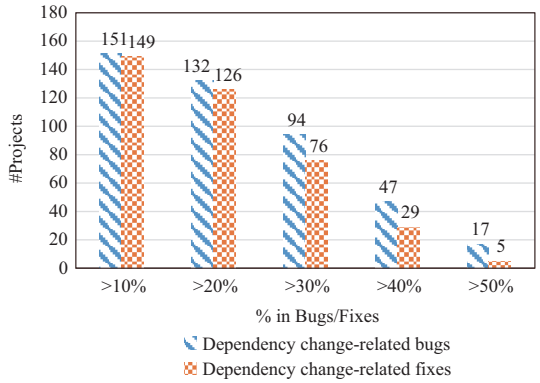

Figure 5 (Color online) The proportion of dependency change-related bugs/fixes in 157 studied project accumulation.

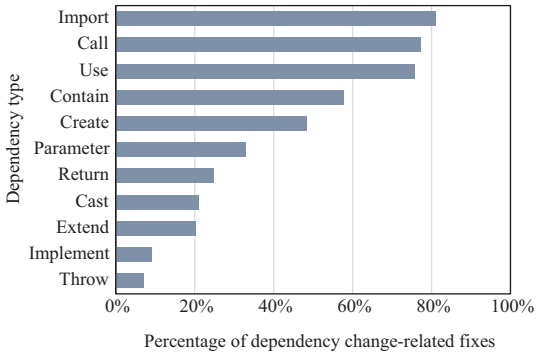

Figure 6 The distribution of dependency types in dependency change-related fixes.

as the label results, which can further be integrated.

In total, we identify 54218 dependency change-related fixes from 182621 fixes. Furthermore, 46164 dependency change-related bugs from 140456 bugs are obtained.

4 Empirical study

Based on the collected data, in this paper, we aim to explore the following three research questions.

RQ1: What percentage of bug fixes introduce dependency-level changes for each project?

The answer to this question would help us better understand the relation between bugs/fixes and dependency attributes.

RQ2: What characteristics of bugs make it more prone to introduce dependency-level changes when fixing them?

The answer to this question would shed light on the scenarios of introducing dependency-level changes through the bug characteristic analysis.

RQ3: Do patched files with dependency-level changes tend to incur more maintenance costs than patched files without dependency-level changes?

The answer to this question would explore the differences between bug fixes with/without dependency-level changes in depth through its patched file analysis.

4.1 RQ1: ratio analysis

To investigate the percentage of bug fixes that introduce dependency-level changes, we conduct a quantitative analysis of dependency change-related bugs and fixes in the 157 subjects collected in Subsection 3.2. We further analyze the types of dependency-level changes introduced by fixing bugs.

4.1.1 *Quantity analysis*

To investigate the number of bugs and fixes that are related to dependency-level changes, for each subject, we compute the proportion of bug fixes that introduce dependency-level changes in all the fixes and the proportion of bugs whose fixes lead to dependency-level changes in all the bugs.

Result. Figure 5 plots the distribution, where the horizontal axis shows the proportion of dependency change-related bugs (%dependency change-related bugs) or dependency change-related fixes (%dependency change-related fixes), and the vertical axis shows the proportion of projects (%projects) that own the corresponding dependency change-related bugs and fixes. As presented in Table 2, we can find that in total, 30% of bug fixes (54218/182621) introduce dependency-level changes. These fixes are further involved in 33% of the bugs (46164/140456) since a bug-fix commit may fix several bugs and multiple commits may be needed to fix a single bug. This result is also prevalent in most of the studied projects. As presented in Figure 5, 60% of the subjects (94/157) contain at least 30% dependency change-related bugs, and 48% of subjects (76/157) contain at least 30% dependency change-related fixes.

4.1.2 *Dependency type analysis*

We further explore the dependency types involved in the bug-fix commits that introduce dependency-level changes. To achieve it, we conduct a multi-label classification for the 54218 dependency change-related fixes. Specifically, 11 labels, illustrated in Subsection 2.2, can be individually assigned to each fix with dependency-level changes. Each label corresponds to a dependency type. For example, if a dependency change-related fix introduces changes on two dependency types: import and call, it can be assigned both labels, namely, import and call.

Result. Figure 6 presents the results, where the vertical axis shows various types, and the horizontal axis shows the proportion containing each type in dependency change-related fixes. As presented in Figure 6, all of these dependency types are involved in bug fixes with dependency-level changes. 9 of these 11 types of dependencies, except Implement (implementing interfaces) and Throw (throw exceptions), are involved in at least 20% of dependency change-related fixes. We also observed that, import (importing header classes), call (method call), and use (use/set variables of other classes) are the three most common changed dependency types in bug fixes related to dependency-level changes. These three types are frequently involved in over 60% dependency change-related fixes.

Answer to RQ1. About one-third of bugs (33%) and their fixes (30%) are related to dependency-level changes, involving multiple types of dependencies. The non-trivial quantities and multiple aspects in dependencies of these bugs/fixes inform us to further explore the impact of dependency-level changes on other quality attributes such as the severity of involved bugs and maintenance costs of involved patched files.

Implications. This result advanced our understandings: (1) when fixing bugs, developers not only modified a few lines of code as we expected, but they may also commit numerous complex dependency-level changes in bug-fix commits like commit 11be3f7 presented in Figure 3; (2) the dependency change-related bugs are not well supported by current bug prediction techniques [17, 18] for they focus on bugs on the per-file level and do not reflect dependencies among files. The dependency change-related fixes are also not well supported by current bug fix/code change analysis techniques [4–6] for they focus on low-level differences on code elements/references rather than high-level abstractions of multiple types of dependencies among files. This result encourages us to further explore dependency change-related bugs and fixes in depth.

4.2 RQ2: bug characteristics analysis through dependency-level changes

To understand the scenarios of introducing dependency-level changes, we revisit the 140456 bugs in these 157 subjects from five aspects by considering dependency-level changes, including bug priority, bug fixing churn (lines of code), bug fixing time, bug reopening, and bug inducing, which are frequently investigated in bug characteristics research [13, 14, 19, 20].

4.2.1 *Bug priority analysis*

Bug priority reflects the awareness of developers toward bugs. To understand the relation between introducing dependency-level changes and bug priority, we investigate whether fixing bugs with high priority are more likely to introduce dependency-level changes by using the priority data from bugs reports collected in Subsection 3.2.

Result. Table 3 presents the results. The column: priority presents five types of bug priorities, where Blocker > Critical > Major > Minor > Trivial. The columns: #bugs and #dependency change-related bugs present the number of bugs and dependency change-related bugs for each priority respectively. As presented in Table 3, fixing bugs of high priorities (Blocker (34.7%), Critical (33.3%), and Major (34.9%)) are prone to introduce dependency-level changes compared with the average value (33% of all the bugs related to dependency-level changes). One possible reason might be that developers carelessly and unconsciously introduce these changes due to the time pressure of fixing according to the work of Medvidovic et al. [21–24]. Fixing bugs of low priorities (Minor (16.0%) and Trivial (28.4%)) present a relatively low probability of causing dependency-level changes. A possible explanation is that developers may commit fixes on trivial program anomalies (e.g., exception handling, problems with a return value) during fixing these bugs with trivial/minor priority. Bugs with minor priorities present the lowest probability. The reason might be bugs with this priority contain the least number compared with bugs with other priorities.

Table 3 The dependency change-related bugs in bugs with various priorities

Priority	#Bugs	#Dependency change-related bugs
Blocker	7302	2531 (34.7%)
Critical	11395	3799 (33.3%)
Major	92300	32193 (34.9%)
Minor	5918	948 (16.0%)
Trivial	23541	6693 (28.4%)

Table 4 The dependency change-related bugs in bugs with various fixing churn

Fixing churn	#Bugs	#Dependency change-related bugs
Top 10%	14046	12626 (89.9%)
Top 20%	28091	23568 (83.9%)
Top 30%	42137	32203 (76.4%)
Top 40%	56182	38204 (68.0%)
Top 50%	70228	42108 (60.0%)
Top 60%	84274	44967 (52.8%)
Top 70%	98319	45817 (46.6%)
Top 80%	112365	45957 (40.9%)
Top 90%	126410	46140 (36.5%)
Top 100%	140456	46164 (32.9%)

4.2.2 *Fixing churn/time analysis*

Fixing churn/time reflects the efforts spent on bug fixes by developers. To understand the correlation between fixing churn/time and introducing dependency-level changes, we measure whether fixing bugs with large churn or long time is more likely to introduce dependency-level changes from bugs reports collected in Subsection 3.2. The fixing churn is gathered from collected bug-fix commits. For a bug containing multiple bug-fix commits, we sum up fixed lines of code of each commit as the fixing churn. The fixing time is calculated using the create time and resolution time from tracking traces. For a bug reopening multiple times, we sum up the time of each fix as the fixing time.

Result. Table 4 presents the results of fixing churn. The column: fixing churn presents three sets of bugs, which are the top 10% to 100% percentile of bugs ranking with fixing churn. The columns: #bugs and #dependency change-related bugs present the number of bugs and dependency change-related bugs in each set. As presented in Table 4, bugs with the most fixing churn (top 10%) have an extremely high probability (89.9%) to cause dependency-level changes compared with the average value (33% of all the bugs). Bugs with the moderate fixing churn (top 30%) and major fixing churn (top 50%) also have a high probability (76.4% and 60.0%) to introduce dependency-level changes in bug fixes. Intuitively, the reason for the high probabilities of dependency-level changes in bugs with large churn might be that these code changes are complex on modifying dependency attributes and thus consume more fixing lines of code.

Table 5 presents the results of fixing time. The column: fixing time presents three sets of bugs, which are top 10% to 100% percentile of bugs ranking with fixing time. The columns: #bugs and #dependency change-related bugs present the number of bugs and dependency change-related bugs in each bug set. As presented in Table 5, bugs with the most fixing time (top 10%) have a relatively high probability (40.9%) to cause dependency-level changes compared with the average value (33% of all the bugs). Bugs with moderate fixing time (top 30%) and major fixing time (top 50%) also have a relatively high probability to introduce dependency-level changes in fixes. The possible explanation for the relatively high probabilities of dependency-level changes in bugs with a long time might be these code changes present complexity on dependency attributes and thus consume more fixing time. Some of these bugs are also assigned with high priorities presented in Subsection 4.2.1 and required to be fixed in a short time, thus causing not extremely high probabilities as we expected.

4.2.3 *Bug reopening/inducing analysis*

Bug reopening or inducing reflects the risk of fixing bugs. To understand this correlation, we investigate whether bugs reopened again are more likely to introduce dependency-level changes. Furthermore, we also measure whether bugs that induce the presence of new bugs during fixing are prone to cause dependency-level changes. The reopening bugs are gathered based on the tracking traces (i.e., marked as "reopen")

Table 5 The dependency change-related bugs in bugs with various fixing time

Fixing time	#Bugs	#Dependency change-related bugs
Top 10%	14045	5747 (40.9%)
Top 20%	28091	11489 (40.9%)
Top 30%	42137	17149 (40.7%)
Top 40%	56182	22529 (40.1%)
Top 50%	70228	27541 (39.2%)
Top 60%	84274	32108 (38.1%)
Top 70%	98319	36378 (37.0%)
Top 80%	112365	40114 (35.7%)
Top 90%	126410	43359 (34.3%)
Top 100%	140456	46164 (32.9%)

Table 6 The dependency change-related bugs in reopening/inducing bugs[a)]

Type	#Bugs	#Dependency change-related bugs
Reopening bugs	9599	4116 (42.9%)
Inducing bugs	5627	2945 (52.3%)

a) Inducing bugs: the bugs whose fixes induce new bugs during fixing.

collected in Subsection 3.2. The inducing bugs are gathered based on the issue link from bug reports collected in Subsection 3.2. If a bug has the issue link to other bugs marked with "break" and "cause", it can be identified as the inducing bug [25].

Result. Table 6 presents the results. The column: type presents two bug sets that reopen again (reopening bugs) or induce the presence of new bugs during fixing (inducing bugs). The columns: #bugs and #dependency change-related bugs present the number of bugs and dependency change-related bugs in each bug set, respectively. As presented in Table 6, bugs reopening again have a high probability to introduce dependency-level changes (42.9%) compared with the average value (33% of all the bugs). Similarly, bugs inducing the presence of new bugs have a higher probability (52.3%) to cause dependency-level changes. A possible explanation of these results is that the introduced dependency-level changes during fixing propagate the bug-proneness among files through changed dependencies and thus cause bugs difficult to be eradicated. Table 7 presents the distribution of dependency types in dependency change-related bugs from reopening bugs and inducing bugs. We found that the rankings of the proportion of 11 studied dependency types in these two kinds of bugs are almost the same. However, there still exist several differences. (1) The proportions of 11 studied dependency types in dependency change-related bugs from inducing bugs are slightly higher than the proportions in dependency change-related bugs from reopening bugs. It pointed out that dependency change-related bugs from inducing bugs present higher complexity in dependency types than dependency change-related bugs from reopening bugs. (2) The proportion of Use type is higher than the proportion of Call type in dependency change-related bugs from inducing bugs. The opposite result occurs in dependency change-related bugs from reopening bugs. It indicates that the call dependency type is more significant in dependency change-related bugs from reopening bugs. When a dependency change-related bug with potential risks is identified, these observations on dependency types will assist developers to further classify it into the reopening bug or the inducing bug. The corresponding measurements on this dependency change-related bug will be taken according to the classification result.

Answer to RQ2. Dependency-level changes in bug-fix commits are more likely to be introduced when fixing bugs with high priority, large fixing churn (lines of code), long fixing time, reopening again, and inducing the presence of new bugs. These results inform us to conduct rigorous code review and testing on bug-fix commits with dependency-level changes before integration. The dependency attributes in bug-fix commits can be further leveraged as features to improve change-level bug prediction research.

Implications. The result of RQ2 is intuitive and advanced our understanding as follows. (1) Dependency-level changes are more likely to be introduced when fixing bugs with high costs and risks from studied five characteristics. For these bugs, although many possible factors are explored in previous studies [25–28], dependency attributes are not considered yet. This result implies that the tool is needed to report warnings when developers commit dependency-level changes, which may further undergo rigorous code review and testing as needed. (2) The dependency attributes in bug-fix commits can be further leveraged as features to improve change-level bug prediction research. During software evolution and

Table 7 The distribution of dependency types in dependency change-related bugs in reopening/inducing bugs

Dependency type	Inducing bugs (%)	Reopening bugs (%)
Import	**84.5**	**82.9**
Use	**82.8**	**78.4**
Call	**81.9**	**81.0**
Contain	67.1	61.0
Create	56.0	52.3
Parameter	44.7	33.4
Return	34.2	24.8
Cast	28.8	23.9
Extend	27.4	23.2
Implement	13.8	9.4
Throw	10.3	7.7

maintenance, developers always face challenges for non-trivial quantities and fast delivery of commits on a day-to-day basis [24, 29]. Change-level bug prediction techniques can assist to detect bug-introducing commits as first introduced. Although existing techniques improved its performance by deriving features from fine-grained code changes in commits [27,30,31], dependency-level changes as a new dimension have not been fully exploited in this area yet. Our results provide hints to further improve change-level bug predicting research deriving features from dependency-level changes in commits.

4.3 RQ3: patched file analysis through dependency-level changes

As presented in Subsection 4.2, dependency-level changes are introduced when fixing bugs with high costs and risks. In this subsection, we further explore the differences between files with or without dependency-level changes. Following the definition of bug-prone files in most bug prediction work, we regard the files that are modified in at least one bug-fixing commit as patched files. To investigate the impact of dependency-level changes on attributes of patched files, we need to go deep into each bug fix for each project. We gather all patched files and further classify them into two categories: with dependency-level changes and without dependency-level changes. It is not realistic to conduct such a study on the whole dataset used in RQ2 for some projects that may have noise in their bug fixes. Thus, we limit our scope and focus on 8 representative Apache open source projects as our subjects. The bug fixes in these projects have high quality and most of them are also frequently investigated in related research [12,32,33]. We intensively studied 12722 dependency change-related fixes involving 11037 bugs in these 8 subjects.

4.3.1 *Impact on patched files*

To explore how dependency-level changes in bug-fix commits impact patched files, we first compute the proportion of patched files involved in dependency change-related fixes, and also investigate the maintenance cost of these patched files compared with patched files without dependency-level changes. Specifically, for each project, we use Top_x%BF to represent the top x% percentile of patched files in a project based on the number of fixing times, which indicates the files that are most frequently modified in bug-fixing commits. We use Top_x%BC to represent the top x% percentile of patched files based on the fixing lines of code, which indicates the files that are modified with most lines of code in bug-fixing commits.

To represent the proportion of patched files with dependency-level changes, for each project, we gather the involved files in dependency change-related fixes and compute its proportion on all patched files, denoted by Prop_PF; its proportion on the top x% percentile of patched files based on bug-fixing frequency (Top_x%BF), denoted by Prop_x%BF; and its proportion on top x% percentile of patched files base on bug-fixing churn (Top_x%BC) as Prop_x%BC. Following the work of Mo et al. [12], we further measure the maintenance costs of patched files with dependency-level changes using BF_inc and BC_inc, which are defined as follows.

If one bug involves multiple files containing both maintenance costs of dependencies and non-dependencies, we intuitively count all the costs on files having dependency-level changes as dependency costs and count the cost on other files as non-dependency costs. This measurement requires high quality of bug fixes and we use bug fixes in JIRA to limit this threat. If one file relates to the dependency bug, we only count its costs on bug fixes when causing dependency-level changes as dependency costs. Other

costs are regarded as non-dependency costs. Based on these definitions, we obtain two collections as follows:

$$\text{DepFreCost} = \{(f_i, \text{DepFre}(f_i), \text{DepCost}(f_i)) \mid i = 1, 2, \ldots, m \wedge \text{DepFre}(f_i) \neq 0\}, \tag{1}$$

$$\text{NonDepFreCost} = \{(f_j, \text{DepFre}(f_j), \text{DepCost}(f_j)) \mid j = 1, 2, \ldots, n \wedge \text{DepFre}(f_j) \neq 0\}. \tag{2}$$

DepFreCost and NonDepFreCost represent the collections of dependency costs and non-dependency costs. For a file, it may both have dependency costs on DepFreCost and non-dependency costs on NonDepFreCost. We further define BF_inc and BC_inc as

$$\text{BF\_inc} = \frac{\frac{1}{m}\sum_{i=1}^{m}\text{DepFre}(f_i) - \frac{1}{n}\sum_{j=1}^{n}\text{NonDepFre}(f_j)}{\frac{1}{n}\sum_{j=1}^{n}\text{NonDepFre}(f_j)} \times 100\%, \tag{3}$$

$$\text{BC\_inc} = \frac{\frac{1}{m}\sum_{i=1}^{m}\text{DepCost}(f_i) - \frac{1}{n}\sum_{j=1}^{n}\text{NonDepCost}(f_j)}{\frac{1}{n}\sum_{j=1}^{n}\text{NonDepCost}(f_j)} \times 100\%. \tag{4}$$

BF_inc represents the average frequency of involving dependency costs for each file over the average frequency of involving non-dependency costs for each file. BC_inc represents the increase of the average dependency costs for each file over the average non-dependency costs for each file.

Result. Tables 8–10 present these results of studied subjects. As presented in column Prop_PF of Table 8, patched files with dependency-level changes capture a significant proportion of all the patched files. 48.2% to 71% of patched files (61.3% on average) are involved in dependency change-related fixes. Moreover, these files capture the most severe patched files. Actually, as presented in column 10% on Prop_x%BF of Table 9, 84.1% to 99.2% of the top 10% percentile patched files with fixing frequency (93.8% on average) are fixed with dependency-level changes. As presented in column 10% on Prop_x%BC of Table 10, 46.8% to 92.1% of top 10% percentile patched files with fixing churn (77.8% on average) are also captured by dependency change-related fixes. Patched files with dependency-level changes also consume expensive maintenance costs. The columns: BF_inc and BC_inc of Table 8 suggest that, on average, patched files with dependency-level changes consume an extra 30.7% fixing times (bug frequency) and 26.1% fixing lines of code (bug churn) over patched files without dependency-level changes.

4.3.2 *Patch pattern analysis*

Since patched files with dependency-level changes take significant maintenance efforts and costs, we further investigate the reasons for it. Specifically, we study the increasing costs of patched files with dependency-level changes in multiple bug-fix commits by manually analyzing how two of these fixes interact with each other and thus make overlapped files incurring repeated patches. We first assume a dependency change-related fixes as a pair of files having modified dependencies with them. Given two dependency change-related fixes, they may overlap with 0, 1, and 2 files. We only consider the case of overlapping with one file because this case modified dependencies among the most three files and also can propagate the bug-proneness to a group of files causing significant costs later. For the case of overlapping with one file, the possible interaction pattern can be summarized into three representative patch patterns shown in Figure 7, where each node represents a file involving in a bug-fixing commit that introduces dependency-level changes and each edge ($f_1 \rightarrow f_2$) represents the dependency between these files modified by the certain bug-fixing commit. For an edge, $f_1 \rightarrow f_2$, f_1 is called the leading file and f_2 is called the subordinate file. Thus, an instance of the pattern can be described as follows.

• **Dissemination.** This pattern describes the case: for two fixes with dependency-level changes above, they share the same leading file, and two different subordinate files shown in Figure 7(a). We term this case as the dissemination pattern for its branch-like shape. For example, developers modify dependencies between AppInfo.java (f_1) and YarnApplicationState.java (f_2) when fixing YARN-1407[16]. Dependencies between AppInfo.java (f_1) and AppInfoXmlVerifications.java (f_3) are also modified when fixing YARN-7451. Frequent dependency-level changes on AppInfo.java (f_1) may make it evolve as an unstable utility/library class depended by other files that consume more costs during maintenance.

• **Concentration.** This pattern describes the case: for two fixes with dependency-level changes above, they share the same subordinate file and two different leading files. We term this case as an instance of concentration pattern for its merging-like shape. For example, developers modify dependencies between

16) https://issues.apache.org/jira/browse/YARN-1407.

Table 8 The maintenance cost of patched files with dependency-level changes

Subjects	Prop_PF (%)	BF_inc (%)	BC_inc (%)
Pig	55.7	**42.2**	**9.1**
Hadoop	67.8	29.7	28.6
Cassandra	**71.0**	27.5	23.4
Camel	57.8	19.4	**44.9**
Cxf	62.6	25.0	**35.6**
Openjpa	**48.2**	**41.8**	6.2
Hbase	61.2	32.8	29.6
Pdfbox	66.5	27.5	31.2
Average	**61.3**	**30.7**	**26.1**

Table 9 The impact of dependency-level changes on the bug frequencies of patched files (%)

Subjects	Prop_x%BF									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Pig	96.3	92.6	84.5	80.2	74.5	69.9	64.9	62.0	58.8	55.7
Hadoop	97.2	93.4	88.8	84.9	82.6	77.7	73.2	70.0	67.8	67.8
Cassandra	**99.2**	96.5	93.5	89.9	85.9	82.8	79.3	75.3	73.0	**71.0**
Camel	**84.1**	72.1	64.3	60.8	59.8	58.8	58.2	58.4	58.1	57.8
Cxf	92.5	86.2	78.9	74.0	70.3	67.8	66.4	64.9	63.3	62.6
Openjpa	89.6	82.6	80.0	69.0	61.7	56.5	53.2	51.0	49.4	**48.2**
Hbase	97.4	89.8	82.1	78.3	74.9	71.8	68.1	65.3	63.1	61.2
Pdfbox	94.8	90.7	86.4	83.1	79.6	76.0	72.5	69.1	67.4	66.5
Average	**93.9**	**88.0**	**82.3**	**77.5**	**73.7**	**70.2**	**67.0**	**64.5**	**62.6**	**61.3**

Table 10 The impact of dependency-level changes on the bug churn of patched files (%)

Subjects	Prop_x%BC									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Pig	54.6	56.7	58.1	60.0	62.7	62.6	62.1	60.2	57.9	55.7
Hadoop	88.5	87.0	84.7	83.4	81.5	79.2	76.1	73.2	69.9	67.8
Cassandra	91.2	86.9	84.6	84.1	82.8	80.9	80.5	78.3	74.7	**71.0**
Camel	88.4	89.6	89.4	89.0	86.0	81.4	75.2	67.8	62.7	57.8
Cxf	**92.1**	88.5	84.5	82.9	80.5	78.5	74.0	70.5	67.1	62.6
Openjpa	**46.8**	47.2	45.1	42.5	40.5	41.5	42.9	47.1	50.1	**48.2**
Hbase	70.4	77.8	79.1	78.8	77.5	75.2	72.5	68.9	65.3	61.2
Pdfbox	91.1	88.9	86.4	84.8	84.4	83.1	79.8	75.8	71.1	66.5
Average	**77.9**	**77.8**	**76.5**	**75.7**	**74.5**	**72.8**	**70.4**	**67.7**	**64.8**	**61.3**

IntrospectionSupport.java (f_1) and MailConsumer.java (f_3) when fixing CAMEL-6905[17]. Dependencies between SynchronizationAdapter.java (f_2) and MailConsumer.java (f_3) are also modified when fixing CAMEL-5376[18]. Frequent dependency-level changes on MailConsumer.java (f_3) may make it evolve as a God class [34], which are more bug-prone for its increasing complexity and size.

• **Domino.** This pattern describes the case: for two fixes with dependency-level changes above, the subordinate file in one fix is also the leading file in the other. We term this case as an instance of domino pattern for a cascade of fixes can be correlated in this way. For example, developers modify dependencies between Column.java (f_1) and DBDictionary.java (f_2) when fixing OPENJPA-274[19]. Dependencies between DBDictionary.java (f_2) and SQLErrorCodeReader.java (f_3) are also modified when fixing OPENJPA-458[20]. Dependency-level changes may cause a chain from Column.java (f_1) to SQLErrorCodeReader.java (f_3), having probability to form a cycle [34] with more effort and costs.

We first discover these three patch patterns using two steps: (1) First, for two fixes with dependency-level changes, we identify several instances of patch patterns by recursively examining each overlapped patched file for it may contain more than three files causing the propagation of bug-proneness; (2) Then,

17) https://issues.apache.org/jira/browse/CAMEL-6905.
18) https://issues.apache.org/jira/browse/CAMEL-5376.
19) https://issues.apache.org/jira/browse/OPENJPA-274.
20) https://issues.apache.org/jira/browse/OPENJPA-458.

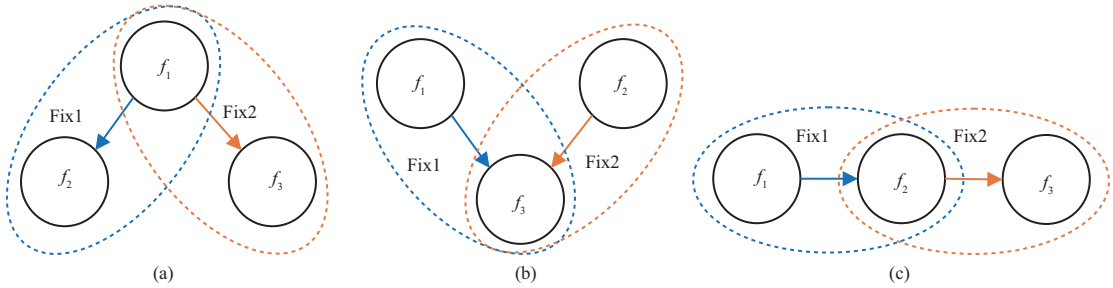

Figure 7 (Color online) Patterns of patched files with dependency-level changes. (a) Dissemination; (b) concentration; (c) domino. (node: fixed files; edge: changed dependencies)

Table 11 The impact of different dependency-level change patterns on patched files in studied projects (%)

Subjects	Dissemination			Concentration			Domino		
	Prop_PF	BF_inc	BC_inc	Prop_PF	BF_inc	BC_inc	Prop_PF	BF_inc	BC_inc
Pig	50.2	50.0	13.9	36.2	79.9	28.2	45.2	53.8	**15.1**
Hadoop	53.9	47.1	48.7	37.9	83.5	73.1	45.9	60.2	62.3
Cassandra	**58.0**	47.9	40.1	**50.5**	**60.0**	52.1	**58.4**	**44.2**	33.8
Camel	47.9	**26.3**	**57.5**	**14.4**	**153.9**	**124.8**	41.5	27.9	**66.7**
Cxf	40.6	**57.5**	**70.2**	25.1	104.3	**97.3**	31.2	**75.6**	**89.1**
Openjpa	**26.3**	**95.9**	43.5	18.3	**129.9**	56.5	**22.6**	**108.4**	48.8
Hbase	49.0	51.4	−13.2	37.8	77.5	95.1	46.3	55.3	58.2
Pdfbox	52.3	45.1	46.8	33.5	92.2	91.1	45.3	57.3	55.7
Average	**47.3**	**52.7**	**38.4**	**31.7**	**97.7**	**77.3**	**42.1**	**60.3**	**53.7**

we exhaustively detect all instances of patterns given all fixes with dependency-level changes. For involved patched files, we aim to investigate the impact of different patterns and pattern combinations on these patched files.

Result. Table 11 presents the impact of different patterns. For patched files in each pattern, we measure its proportion on all patched files using Prop_PF and its maintenance cost over patched files without dependency-level changes using BF_inc and BC_inc, which are illustrated in Subsection 4.3.1. As presented in the Prop_BF column in Table 11, the dissemination pattern covers the largest number of patched files (47.3% on average), while the concentration pattern captures the minimum number of patched files (31.7% on average). The domino pattern is in between (42.1% on average). These results suggest each pattern only captures a subset of all the patched files. As presented in the BF_inc and BC_inc columns in Table 11, the concentration pattern consumes the most maintenance costs with extra 97.7% fixing time and 77.3% fixing churn. However, the dissemination pattern consumes the least with extra 52.7% fixing time and 38.4% fixing churn. The domino pattern is in between with an extra 60.3% fixing time and 53.7% fixing churn. This result also reveals that each pattern consumes different costs. If a dependency change-related fix is committed, we should avoid the presence of concentration pattern and employ the dissemination way, which will attempt to avoid the propagation of bug-proneness from upstream files to downstream files. The features of patterns can further assist us to predict the bug of committed files for the concentration pattern is riskier than the dissemination pattern and the domino pattern. We should further take these into account during fixing bugs.

Table 12 presents the correlation between maintenance costs and the number of patterns using Pearson correlation analysis [12]. We classify all the patched files into four file sets according to the number of involved dependency-level change patterns (0→3). For files in each set, we measure the average fixing frequency in the #BF column and the average fixing lines of code in the #BC column. As presented in the PCC row in Table 12, files in greater numbers of patterns incur more effort and costs. The correlation between bug frequency and the number of patterns ranges from 0.76 to 0.85. The average value is 82% through calculating. The correlation between bug churn and the number of patterns ranges from 0.36 to 0.91. The average value is 78% through calculating. From the statistics of p-value, Cohen'd, and effect size (r), its strong correlation is also validated. This result implies that, with the increasing number of patterns, files are captured with expensive costs. When fixing bugs, we should realize pattern combinations and their severe consequences in time.

Table 12 The correlation between maintenance costs and the number of patterns[a)]

#Patterns	Pig		Hadoop		Cassandra		Camel	
	#BF	#BC	#BF	#BC	#BF	#BC	#BF	#BC
0	2.1	176.3	1.9	48.3	2.0	91.3	1.7	42.8
1	2.8	106.9	2.1	57.8	2.9	74.0	2.0	45.7
2	2.7	110.8	2.4	87.9	2.8	165.4	1.6	72.6
3	6.3	224.2	7.7	223.6	10.2	314.6	6.2	137.4
PCC	**0.85***	**0.34***	0.82*	0.88*	0.82*	0.90*	**0.76***	0.91*
Cohen'd	1.39$^\$$	4.43$^\$$	1.07$^\$$	2.06$^\$$	1.19$^\$$	2.38$^\$$	0.87$^\$$	2.71$^\$$
Effect size (r)	0.57$^+$	0.91$^+$	0.47	0.71$^+$	0.51$^+$	0.76$^+$	0.40	0.80$^+$
#Patterns	Cxf		Openjpa		Hbase		Pdfbox	
	#BF	#BC	#BF	#BC	#BF	#BC	#BF	#BC
0	1.6	43.1	1.4	106.0	2.2	366.6	2.4	113.6
1	2.1	57.7	1.7	127.5	2.7	178.2	2.4	135.0
2	2.4	79.2	2.2	190.2	2.9	1491.5	2.4	110.1
3	7.1	165.5	6.5	384.3	8.9	931.0	8.5	349.7
PCC	**0.85***	**0.92***	**0.85***	0.91*	0.83*	0.65*	0.77*	**0.76***
Cohen'd	1.02$^\$$	2.52$^\$$	0.87$^\$$	2.58$^\$$	1.27$^\$$	2.03$^\$$	1.19$^\$$	2.48$^\$$
Effect size (r)	0.45	0.78$^+$	0.39	0.79$^+$	0.53$^+$	0.71$^+$	0.51$^+$	0.77$^+$

a) PCC: pearson correlation coefficient. #patterns: the number of involved patterns. *: p-value<0.05. $: Cohen'd>0.8. +: effect size(r)>0.5.

Answer to RQ3. Patched files with dependency-level changes capture a significant proportion (61.3%) and incur huge maintenance costs on fixing frequency (30.7%) and churn (26.1%) compared with patched files without dependency-level changes. These files interact through changed dependencies in multiple bug-fix commits with three representative patterns and thus incur repeated patches. Patched files in different patterns incur drastically different maintenance costs and files in greater numbers of patterns consume more efforts. These results inform developers to test/review patched files with dependency-level changes before committing bug fixes in time. The dependency attributes in patched files in bug-fix commits can further be leveraged as features to improve file-level bug prediction research, especially just-in-time bug prediction.

Implications. The result of RQ3 advanced our understanding as follows. (1) Patched files with dependency-level changes capture a large proportion and consume significant maintenance costs. Although previous studies [9,12] indicated files with complex dependencies are hard to fix, our work pointed out that continuous dependency-level changes may be the root cause for increasing bug-proneness. Compared with the work analyzing large quantities of files and involved dependencies in the whole project, our results suggest that concentrating on modified dependencies among several files per bug-fix commits to ease developers' burden deserves our more attention. We believe the tool is needed to report warnings of files with dependency-level changes in time before committing bug fixes. These files may further be reviewed or tested as needed. (2) The dependency attributes in bug-fix commits can be further leveraged as features to improve just-in-time bug prediction research. During software evolution and maintenance, developers may be lost in non-trivial quantities and fast delivery of commits on a day-to-day basis [24,29]. Just-in-time bug prediction techniques can assist developers to detect bug-prone files in commits and reduce their potential threats through fixing. Although existing techniques improved its performance by deriving low-level features from code hunks within the single file [35–39], high-level abstractions of multiple types of changed dependencies among files as a new dimension have not been fully exploited in this area yet. Our results provide hints to further improve just-in-time bug predicting research by deriving features from files with dependency-level changes. Our results of patterns and their combinations can provide useful suggestions to improve these techniques.

5 Applications of our study

This section discusses the follow-up research motivated by our study, including the benchmark, toolkit, and insights for improving existing research.

Benchmarks of dependency change-related bugs and fixes. Our study outputs a large and comprehensive dataset of bugs containing dependency-level changes in fixes, including 46164 dependency

change-related bugs and 54218 dependency change-related fixes of 157 Apache open source projects. Each bug is labeled with priority, fixing time, fixing churn, bug reopening, and bug inducing. Each fix is further assigned labels based on its types of changed dependencies. We believe this dataset can (1) provide an effective and reliable basis for detecting dependency change-related bugs and fixes in open source projects; (2) support the research on improving existing approaches on dependency change-related bugs and fixes.

Dependency-level change detection tool. Our study outputs a dependency-level change detection tool: `DependDiff` to assist us in detecting dependency change-related fixes committed by developers in time during software evolution. We extract the core component of our analysis framework and implemented the automatic tool, `DependDiff`. The input is a bug fix, and the output is the contained dependency changes. It has three steps, including checking out target files, extracting dependencies among files, and obtaining dependency changes, which are described in Subsection 3.2; `DependDiff` can report warning of dependency-level changes in time to avoid increasing costs and endless fixing. The output of `DependDiff` can be further leveraged as the feature (e.g., pattern studied in Subsection 4.3.2) to improve change-level and file-level bug prediction techniques.

Useful insights for improving bug prediction techniques by leveraging dependency-level changes. Although great efforts are spent on bug prediction techniques, they cannot support dependency change-related bugs and fixes, as they cannot support the analysis of multiple types of dependencies among files in bug fixes. (1) Our study provides useful insights and hints on improving change-level bug prediction techniques. Our results shed light on the correlation between dependency change-related bugs and five characteristics of bugs with high severity (Subsection 4.2). We can leverage dependency-level changes as features to improve change-level bug prediction. For state-of-the-art techniques rely on deriving features from fine-grained code changes within singles files, our study of dependency change-related fixes provides new insights from a high-level abstraction of multiple dependencies among files; (2) Our study provides useful insights and hints on improving file-level bug prediction techniques especially just-int-time bug prediction. Our results shed light on patched files with dependency-level changes that capture a significant proportion and incur expensive maintenance costs. We also derive three representative patterns to reflect the dependency-level change features in these files, which can be leveraged in improving just-in-time bug prediction. For state-of-the-art techniques rely on deriving metrics from low-level code hunks as features and our study target for multiple types of changed dependencies in the file level. Overall, our study of dependency change-related bugs and fixes can provide useful insights to improve existing bug prediction research.

6 Threats to validity

In this section, we discuss the threats to validity and limitation of our study.

Internal threats. First, we only investigate the correlation between dependency-level changes and bug fixes but not causality. We investigated the proportion of dependency-level changes in other intents such as adding new features, refactoring/improvement, and testing. We observed that adding new features and fixing bugs are the two types that are most likely to introduce dependency-level changes. This result encourages us to further explore the causality relation in depth by studying each bug report in our future work.

Second, we focus on syntactic dependencies among source files excluding dependencies with third-party libraries. As previous work pointed [8], historical dependencies and semantic dependencies also have an impact on software quality while syntactic dependencies capture most of the patched files. In our future work, we will consider changes with historical and semantic dependencies to further improve our work.

Third, Depends may produce wrong dependencies. To reduce this threat, we reported our found defects. If they are not fixed, we tried to fix them by ourselves. This threat can be further reduced by using more advanced tools.

Fourth, we employ some measures to extract findings on a large dataset. This opens up threats for the correctness of measurements. We run statistical experiments to measure the statistical confidence in our measurements and the results are proved to be significant. We also make all measurement results and related artifacts publicly available to further limit this threat.

External threats. The first threat comes from selected 157 Apache Java open source projects. We only studied dependencies among Java files. It is still unclear whether our study results will generalize to closed source industrial projects and open source projects from other communities. We are also

uncertain whether our results also generalize to projects with other types of programming languages, such as functional languages. Replicating our study on more subjects is our ongoing work.

The second threat comes from locating bug fixes from commits using bugID. Previous work [1] pointed out that, developers may commit bug fixes using the wrong bugID or even without reporting the bugID. To reduce this threat, we studied bug reports in the most influential issue tracking system: JIRA, and most of them are manually entered by experts containing less noise. Studying the impact of missing links on our study results is our future work.

The third threat comes from the quality of bug reports and bug fixes. Previous studies [40,41] pointed out that a reported bug may not be a bug but a feature and a bug fix may not be committed to fix bugs but to finish other tasks. To reduce this threat, we studied bug reports and fixes from open source projects in the most popular community: Apache community. We use the original bug fixes from these projects. These bug data are pragmatic and most of them are manually entered by experts/developers on a day-to-day basis containing less noise, which is also widely studied in the research community. We will further limit this threat by leveraging related methods like untangling changes [40].

7 Related work

In this section, we compare our work with related research.

Bug fix analysis. Prior studies show that developers usually applied bug fixes to multiple code locations in multiple files. Park et al. [13] studied supplementary fixes and find that complex code references can be the root cause of frequent fixing. Zhong et al. [4] intensively conducted a large-scale study of real-world bug fixes, and a significant proportion of these fixes contain changes of multiple program entities, which cannot be solved by state-of-the-art automatic program repair techniques (APR). Wang et al. [3] further studied the patterns of these bug fixes by editing multiple entities and provided some insights for fixing them. This work revealed the fact that bug-prone files are somehow correlated with each other, causing challenges for fixing and patching. Fan et al. [14] conducted a large-scale study on the characteristics of Android framework-specific bug fixes from multiple aspects, including root cause and fixing patterns. Fan et al. [42] also proposed an approach to detect Android framework-specific errors and suggested fixing solutions. Compared with these studies leveraging multiple code locations to automatically generate bug fixes, our results can motivate early detect dependency-level changes in bug fixes and report warnings in time avoiding extra maintenance efforts. Recently, the fixes of some new types of bugs are also investigated by researchers. For example, Lou et al. [43] studied the symptom and fixing pattern of build issues and Chen et al. [44] explore the bugs fixes in the scenario of deploying deep learning applications on the mobile platform. These studies mainly focus on fine-grained code analysis for bug fixes although they are targeting for new scenarios. However, our work sheds insight on bug fixes from high-level analysis of dependency-level changes. It is still uncertain whether dependency-level changes can be applied to these new scenarios of bug fixes, which can be explored in our future work.

Correlation between bug-proneness and dependency. Selby et al. [45] first studied software structure to predict bug-prone files. Zimmermann et al. [46] reported that structure-based network measures can be used to construct successful defect predictor. Xiao et al. [9] further employed the to capture the overlap of dependencies and bugs as the structural anti-patterns. Cui et al. [8] systematically summarized the impact of various types of dependencies on bug detection. Compared with these studies studying the file bug-proneness from complex dependencies, our work reveals the introduction of complex dependencies and we can avoid its increasing costs through early detection of these changes. Compared with these studies analyzing dependencies for the whole project, our results/tools focus on several files with dependency-level changes in a bug fix, which are easier for developers to digest.

Architectural change analysis. Garcia et al. [21] reported the software architecture is difficult to maintain for introduced architectural changes. Le et al. further conducted an empirical study of architectural changes [32] and architectural decay [33] in open source projects. The results showed that architectural changes are frequently induced, causing challenges for software maintenance. Paixao et al. [24] reported, in most cases, developers are not aware of introducing architectural changes in their commits. Compared with these studies, our work explores dependency-level changes, which can further be distilled to detect architectural changes.

8 Conclusion

In this paper, we presented our characterizing study of bug fixes from the dependency-level change perspective. We conducted our study on 157 Apache open source projects involving 140456 bug reports and 182621 bug fixes. Supported by our dependency-level change detection tool: `DependDiff`, we investigated three research questions based on these collected data. The results demonstrated the ratio analysis, bug characteristic analysis, and patch file analysis of bugs and bug fixes related to dependency-level changes. We also presented a suite of qualitative and quantitative results, which provide new insights that may benefit existing approaches such as change-level bug prediction techniques and just-in-time bug prediction techniques.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2020AAA0108800), National Natural Science Foundation of China (Grant Nos. 61632015, 61772408, U1766215, 61721002, 61532015, 61833015, 61902306, 62072351), China Postdoctoral Science Foundation (Grant Nos. 2019TQ0251, 2020M673439), Youth Talent Support Plan of Xi'an Association for Science and Technology (Grant No. 095920201303), Ministry of Education Innovation Research Team (Grant No. IRT 17R86), and Project of China Knowledge Center for Engineering Science and Technology.

References

1 Bachmann A, Bird C, Rahman F, et al. The missing links: bugs and bug-fix commits. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010. 97–106
2 Martinez M, Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng, 2015, 20: 176–205
3 Wang Y, Meng N, Zhong H. An empirical study of multi-entity changes in real bug fixes. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018. 287–298
4 Zhong H, Su Z D. An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering, 2015. 913–923
5 Falleri J R, Morandat F, Blanc X, et al. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014. 313–324
6 Huang K F, Chen B H, Peng X, et al. Cldiff: generating concise linked code differences. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 679–690
7 Hassan A E. Predicting faults using the complexity of code changes. In: Proceedings of the 31st International Conference on Software Engineering, 2009. 78–88
8 Cui D, Liu T, Cai Y F, et al. Investigating the impact of multiple dependency structures on software defects. In: Proceedings of the 41st International Conference on Software Engineering, 2019. 584–595
9 Xiao L, Cai Y F, Kazman R. Design rule spaces: a new form of architecture insight. In: Proceedings of the 36th International Conference on Software Engineering, 2014. 967–977
10 Williams B J, Carver J C. Characterizing software architecture changes: a systematic review. Inf Softw Tech, 2010, 52: 31–51
11 Bass L, Clements P, Kazman R. Software Architecture in Practice. Boston: Addison-Wesley Professional, 2003
12 Mo R, Cai Y F, Kazman R, Xiao L. Hotspot patterns: the formal definition and automatic detection of architecture smells. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture, 2015. 51–60
13 Park J, Kim M, Ray B, et al. An empirical study of supplementary bug fixes. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, 2012. 40–49
14 Fan L L, Su T, Chen S, et al. Large-scale analysis of framework-specific exceptions in Android apps. In: Proceedings of the 40th International Conference on Software Engineering, 2018. 408–419
15 Wu R X, Zhang H Y, Kim S, et al. Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 15–25
16 Kinable J, Kostakis O. Malware classification based on call graph clustering. J Comput Virol, 2011, 7: 233–245
17 da Costa D A, McIntosh S, Shang W, et al. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. IEEE Trans Softw Eng, 2017, 43: 641–657
18 Kim S, Zimmermann T, Jr E J W, et al. Predicting faults from cached history. In: Proceedings of the 29th International Conference on Software Engineering, 2007. 489–498
19 Chen T H, Nagappan M, Shihab E, et al. An empirical study of dormant bugs. In: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014. 82–91
20 Zaman S, Adams B, Hassan A E. Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th Working Conference on Mining Software Repositories, 2011. 93–102
21 Garcia J, Ivkovic I, Medvidovic N. A comparative analysis of software architecture recovery techniques. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, 2013. 486–496
22 Garcia J, Krka I, Mattmann C, et al. Obtaining ground-truth software architectures. In: Proceedings of International Conference on Software Engineering, 2013. 901–910
23 Lutellier T, Chollak D, Garcia J, et al. Comparing software architecture recovery techniques using accurate dependencies. In: Proceedings of the 37th IEEE International Conference on Software Engineering, 2015. 69–78
24 Paixao M, Krinke J, Han D, et al. Are developers aware of the architectural impact of their changes? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017. 95–105
25 Wen M, Wu R X, Liu Y P, et al. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019. 326–337
26 Mi Q, Keung J. An empirical analysis of reopened bugs based on open source projects. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016
27 Yin Z N, Yuan D, Zhou Y Y, et al. How do fixes become bugs? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011. 26–36

28 Zimmermann T, Nagappan N, Guo P J, et al. Characterizing and predicting which bugs get reopened. In: Proceedings of the 34th International Conference on Software Engineering (ICSE), 2012. 1074–1083
29 Shihab E, Hassan A E, Adams B, et al. An industrial study on the risk of software changes. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering, 1991. 1–11
30 Eyolfson J, Tan L, Lam P. Do time of day and developer experience affect commit bugginess? In: Proceedings of the 8th Working Conference on Mining Software Repositories, 2011. 153–162
31 Kim S, Whitehead, E J, Zhang Y. Classifying software changes: clean or buggy? IEEE Trans Softw Eng, 2008, 34: 181–196
32 Le D M, Behnamghader P, Garcia J, et al. An empirical study of architectural change in open-source software systems. In: Proceedings of the 12th Working Conference on Mining Software Repositories, 2015. 235–245
33 Le D M, Link D, Shahbazian A, et al. An empirical study of architectural decay in open-source software. In: Proceedings of IEEE International Conference on Software Architecture (ICSA), 2018
34 Lanza M, Marinescu R. Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Berlin: Springer, 2007
35 Hoang T, Dam H K, Kamei Y, et al. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In: Proceedings of the 16th International Conference on Mining Software Repositories (MSR), 2019. 34–45
36 Kamei Y, Fukushima T, McIntosh S, et al. Studying just-in-time defect prediction using cross-project models. Empir Softw Eng, 2016, 21: 2072–2106
37 Kondo M, German D M, Mizuno O, et al. The impact of context metrics on just-in-time defect prediction. Empir Softw Eng, 2020, 25: 890–939
38 McIntosh S, Kamei Y. Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng, 2018, 44: 412–428
39 Pascarella L, Palomba F, Bacchelli A. Fine-grained just-in-time defect prediction. J Syst Softw, 2019, 150: 22–36
40 Herzig K. The impact of tangled code changes. In: Proceedings of Working Conference on Mining Software Repositories, 2013
41 Herzig K, Just S, Zeller A. It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of Working Conference on Mining Software Repositories, 2013
42 Fan L L, Su T, Chen S, et al. Efficiently manifesting asynchronous programming errors in Android apps. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. 486–497
43 Lou Y L, Chen Z P, Cao Y B, et al. Understanding build issue resolution in practice: symptoms and fix patterns. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020. 617–628
44 Chen Z P, Yao H H, Lou Y L, et al. An empirical study on deployment faults of deep learning based mobile applications. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE), 2021
45 Selby R W, Basili V R. Analyzing error-prone system structure. IEEE Trans Softw Eng, 1991, 17: 141–152
46 Zimmermann T, Nagappan N. Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering, 2008. 531–540