Master Thesis

Title

Evaluating the Effect of “Return Null” on Maintenance

Supervisor
Prof. Shinji Kusumoto

by
Shuhei Kimura

February 5, 2014

Department of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

Evaluating the Effect of “Return Null” on Maintenance

Shuhei Kimura

Abstract

Developers often use null references for the returned values of methods (return null) in object-oriented languages. Although developers often use return null to indicate that a program does not satisfy some necessary conditions, it is generally considered that a method returning null is costly to maintain. One of the reasons is that when a method receives a value returned from a method invocation whose code includes return null, it is necessary to check whether the returned value is null or not (null check). As developers often forget to write null checks, null dereferences occur frequently. However, it has not been clarified yet to what extent return null affects software maintenance.

This thesis shows the effect of return null by investigating return null and null check in the evolution of source code. Experiments conducted on 14 open source projects showed that developers modify return null more frequently than return statements where null does not appear.

This result indicates that return null has a negative effect on software maintenance. It was also found that the size and the development phases of projects have no effect on the frequency of modifications on return null and null check. In addition, we found that all the projects in this experiment had from one to four null checks per 100 lines.

Additional experiments revealed that forgetting to write null check is a serious problem on software maintenance. This is because forgetting to write null check causes runtime exceptions, and unwritten null check accounted for 10% of all modifications to conditional expressions in our experiments. This means that return null, which is the cause of forgetting to write null check, is also costly to maintain.

As a result of the quantitative and qualitative analysis on return null and null check, we concluded that return null should be replaced by exception handling mechanisms.

Keywords

Null Reference
Return Null

Static Analysis
Software Repository
Software Maintenance

Contents

List of Figures

List of Tables

1 Introduction

A null reference is a common mechanism in object-oriented languages, such as Java and C++. Developers use *null* in various roles, for instance, as an initializer or sentinel. In particular, it is general practice that a method returns *null* values when some necessary conditions are not satisfied (hereafter we refer to return statements whose operand is *null* as "**return null**").

Developers often use a return null rather than error constants or exceptions for the following reasons:

- As *null* is a unique value, we know some errors have occurred, even if there are no error messages.
- Returning *null* is easier to write than handling exceptions.

Although return null supports developers for the above reasons, it is generally considered that return null is one of the factors that increase software maintenance efforts [1].

A null dereference, a bug by which a program dereferences the *null* address, is the main problem caused by a return null. To prevent null dereferences from occurring, returned values from a method that includes a return null need to be checked (hereafter we refer to such checking as "**null check**"). Accordingly, when a developer adds return null to a method, a null check needs to be added for all of its invocations. If a developer forgets to add null check, the program will try to dereference *null*, and NullPointerException will be thrown. In addition, *null* contains no information about occurred errors; whereas an exception contains information on errors that have occurred. For this reason, it is very difficult for caller methods that have received *null* to know what kinds of errors have occurred. This characteristic sometimes hides the root causes of errors that have occurred, because a null dereference will occur at places remote from the point of origin of *null* if a program has methods that include return null in a null check (e.g., Figure 1). For these reasons, return null is regarded as a factor that increases the cost of software maintenance.

However, it has not been clarified to what extent a return null affects software maintenance. Therefore, in this research, we conducted an experimental study on 14 open source projects to ascertain whether, and to what extent, the presence of return null and null check affect software maintenance.

We found that the presence of return null and null check increases the cost of software maintenance, and that the source code modifications that are related to them have the following characteristics:

- Developers modified return nulls more frequently than return statements where *null* does not appear.

```
final RevCommit base = walk.next();
if(base == null)
  return null;
```

Figure 1: Example of passing return null

- The size and the development phases of projects did not affect the frequency of modifications on return nulls and null checks.
- The density of null checks in source code was between one and four in 100 lines, in any project. If the density of null checks in a project exceeds 0.04, the project is presumed to have a problem (e.g., developers wrote return null where throwing exception was suitable). None of the projects had a density less than 0.01, and none of the projects had special measures in place for return nulls and null checks.
- There were modifications that had replaced return null with exception handling or an erased null check. These phenomena indicate that programmers considered that the presence of return null and null check were harmful to software evolution.

The contributions of this thesis include the following:

- This thesis revealed that return null affect software maintenance.
- The result indicated that developers should write return null carefully.
- Measurement of return null and null check showed that return null and null check are costly to maintain throughout the entire development periods.
- The result showed the average density of null checks. This value can be used as a criterion (e.g., when the density of null check in a project exceeds the average, developers should apply refactoring to return null and null check).

A developer could replace return null with other mechanisms, for example, exception handling, a proper object such as an empty array, or a NullObject pattern. However, this thesis presents only whether return null matter. Providing how to replace it is our future work.

The rest of this thesis is organized as follows. Section 2 presents the previous work by way of introducing our research. Section 3 defines our research questions and describes the experimental design. Section 4 shows the result of the experiment and answers the research questions. In Section 5, we discuss our answers to the research questions. Section 6 shows the qualitative analysis of modifications to return null and null check, and Section 7 shows the way to replace return null to other mechanisms. Section 8 discusses threats posed to validity. Section 9 provides various approaches to resolving problems related to $null$, and Section 10 concludes this thesis.

2 Background

2.1 Mining Software Repositories

Mining Software Repositories (MSR) is a field that analyzes the data in software repositories [2, 3]. Recently, data mining has attracted much attention not only in the software engineering field [4–7].

2.1.1 Software Repository

A *software repository* is a location for storing software packages. Especially, in the software maintenance field, the term "software repository" mainly means "source code repository." Source code repositories, a kind of software repositories, are locations for managing source code. They keep the change histories of source code.

Version Control Systems (VCSs) are managers of changes in files. *Git*, *Subversion*, and *CVS* are major VCSs. A stored snapshot of files is called as a *revision*.

Using source code repositories brings the following benefits.

Team Development In team development, conflict of changes is a major problem. Since source code repositories can unify the changes from multiple developers, team development becomes easier by using repositories. In addition, developers can browse and review past changes in source code.

Rollback We can rollback to any of the previous revisions. When a bug occurs, rollbacking is useful to detect the cause of bugs. This functionality enables us to use source code repositories as a backup system.

Records of Contributions A repository has records that include who added a function, which code has reviewed, and how many times have developers fixed bugs. Thus, the combination of repositories and bug tracking systems (such as Bugzilla, Trac, and Redmine) is in widespread use. By finding the inequality among project members, developers can improve software development efficiency.

2.1.2 Data in Source Code Repository

The data in source code repositories are various. If we want to obtain characteristics of source code including bugs, we can pick just previous revisions for source code including bugs. We can also know the evolution of a method, the transition of metrics (e.g., cyclomatic complexity, lines of code, couplings), who is the most active contributor, and so on. Time series data cannot be obtained from a single snapshot (revision). MSR has attracted much attention because such data are quite useful for maintenance [8]. In addition, there is a study to obtain more detailed data from repositories by converting a repository into a fine-grained repository [9].

2.1.3 Source Code Repository in Research

We can obtain considerable data from a single project by using source code repository. Multiple revisions from one project enables us to use a part of revisions as learning data and use the residual part of revisions as the reference set. Such a technique is often used in experiments of research on software evolution.

The combination of MSR and machine learning enables us to bring the predicates in software maintenance [10, 11], such as estimating fault-prone module [12, 13] and predicting bugs [14].

2.2 Related Description to Null Reference

2.2.1 Return Null

A null reference ($null$) is a common mechanism in object-oriented languages, such as Java and C++. It is general practice that a method returns $null$ values when some necessary conditions are not satisfied (**return null**). Figure 2(a) is a piece of code including return null on the 9th line.

Developers often use return null rather than error constants or exceptions for the following reasons:

- As $null$ is a unique value, we know some errors have occurred, even if there are no error messages.
- Returning $null$ is easier to write than handling exceptions.

Return null supports developers for the above reasons, it is generally considered that return null is one of the factors that increase software maintenance efforts [1].

2.2.2 Null Check

If a variable has a $null$ value when we try to use the variable, it indicates some problem has happened. To avoid the problem, a developer must check whether a variable has a $null$ value.

Figure 2(c) is a piece of code including a null check on the 5th line. This code means “if a variable named ’ref’ has a $null$ value, exit this method.” Although such an approach is generally used, developers cannot know what error happened.

2.2.3 Null Dereference

Null dereference, accessing information at the "null" address, is a runtime error that many developers face. The error is due to lack null checks. Generally, to track the root causes of null dereference is a difficult task [15].

Figure 2(b) is a piece of code including an invocation that possibly causes null dereference on the 5th line.

2.2.4 Relationship between Return Null, Null Check, and Null Dereference

Null dereference often occurs in programs written in object-oriented programming languages. For this reason, many researchers have proposed techniques to detect null dereference [16–21], and many tools have been released [22], such as FindBugs [23], SALSA [20], JLint, and ESC/Java [24]. However, previous studies did not mention whether, and to what extent, the presence of return null and null check affected software maintenance.

A factor of null dereference is that a program unintentionally refers $null$ when programmers forget to write null check, despite the callee method possibly returns $null$. Figure 2(b) is a code fragment that appears in JGit. command.call(),in the 4th line, invokes the method shown in Figure 2(a). This method may return $null$, as written on the 9th line. The variable named "ref," defined in the 4th line in Figure 2(b), may be assigned $null$ by calling command.call(). As a result, ref.getName(), in the 5th line, possibly causes null dereference. Figure 2(c) is a bug fix for this null dereference. Prefix "+" in a line means that the line was added in this commit. This fix added null check in order to avoid occurrences of null dereference. The commit message, "Do not fail when checking out HEAD," shows the purpose of this commit is to fix the null dereference. As mentioned above, return null, null check, and null dereference are closely related to each other. Thus, we can show how null dereference affects software maintenance by investigating the effect of return null and null check on software maintenance.

To reveal the effect of return null, the experiment in this thesis covers not only return null but also null check. Error constants, such as -1, are used as having the same meaning as return null. However, mainly in Java, as all the variables — except primitive ones — can be assigned $null$, $null$ probably has a greater negative effect on software maintenance than error constants. For this reason, this research focuses on return null and null check.

In the remainder of this thesis, we use the phrase "a statement is costly to maintain" to indicate "developers modify the statement many times" and "the statement has a trend that new bugs are likely to be introduced into it."

```java
public Ref call() throws GitAPIException, RefAlreadyExistsException,
    RefNotFoundException, InvalidRefNameException,
    CheckoutConflictException {
  checkCallable();
  processOptions();
  try {
    if (checkoutAllPaths || !paths.isEmpty()) {
      checkoutPaths();
      status = new CheckoutResult(Status.OK, paths);
      setCallable(false);
      return null;
  }
...
```

(a) Method including return null

```java
...
try {
  String oldBranch = db.getBranch();
  Ref ref = command.call();
  if (Repository.shortenRefName(ref.getName()).equals(oldBranch)) {
    outw.println(MessageFormat.format(
      CLIText.get().alreadyOnBranch,
...
```

(b) Invokes the method shown in (a)

```java
...
try {
  String oldBranch = db.getBranch();
  Ref ref = command.call();
+ if (ref == null)
+   return;
  if (Repository.shortenRefName(ref.getName()).equals(oldBranch)) {
    outw.println(MessageFormat.format(
      CLIText.get().alreadyOnBranch,
...
```

(c) After adding null check

Figure 2: Example code fragments of return null, null dereference, and null check

3 Experimental Design

In this section, we explain the design of our experiment on open source projects. Herein, we use the abbreviations listed in Table 1 for convenience. In addition, $|S|$ means the number of elements in a given set S. Let c be a commit between revision r and $r+1$, and then the set of added/deleted ret_{null}, ret_{not}, $cond_{null}$, and $cond_{not}$ in c are defined as ΔRet^c_{null}, ΔRet^c_{not}, $\Delta Cond^c_{null}$, and $\Delta Cond^c_{not}$.

3.1 Research Questions

We investigated the following research questions.

RQ1: Were ret_{null} and $cond_{null}$ modified more frequently than ret_{not} and $cond_{not}$?

RQ2: Did the size of projects affect the frequency of modifications to ret_{null} and $cond_{null}$?

RQ3: Did the development phases of the projects affect the frequency of modifications to

Table 1: Abbreviations

Abbreviation	Explanation of Abbreviation
ret_{null}	return statements whose operands are $null$
ret_{not}	return statements whose operands are NOT $null$
$cond_{null}$	conditional expressions having comparison with $null$
$cond_{not}$	conditional expressions NOT having comparison with $null$
Ret^r_{null}	a set of ret_{null} in revision r
Ret^r_{not}	a set of ret_{not} in revision r
Ret^r	$Ret^r_{null} \cup Ret^r_{not}$
$Cond^r_{null}$	a set of $cond_{null}$ in revision r
$Cond^r_{not}$	a set of $cond_{not}$ in revision r
$Cond^r$	$Cond^r_{null} \cup Cond^r_{not}$
$Desc^r_{null}$	$Ret^r_{null} \cup Cond^r_{null}$
$Desc^r_{not}$	$Ret^r_{not} \cup Cond^r_{not}$
$Desc^r$	$Desc^r_{null} \cup Desc^r_{not}$
C	a set of all commits in a target project
R	a set of all revisions in a target project
loc^r	lines of code in revision r
$latest$	the latest revision in the specified term

ret_{null} and $cond_{null}$?

RQ4: Did the density of $cond_{null}$ increase as projects proceeded?

RQ1 is a question on whether statements including $null$ are, in fact, costly to maintain. In this research, "a statement is costly to maintain" means that the statement is modified frequently. RQ2 and RQ3 are questions on whether the frequency of modifications depends on the characteristics and development periods of projects. RQ4 is a question on how the density of $cond_{null}$ changes during software evolution. If the density of $cond_{null}$ increases as a project proceeds, the occurrences of $cond_{null}$, which is not the functionality we want to realize, are so many that $cond_{null}$ and ret_{null} are costly to maintain. ret_{null} is also costly because it is the cause of writing $cond_{null}$.

3.2 Target Projects

Table 2 shows the list of target projects. All the target projects are Java systems managed using git, and all had a high number of revisions and the size of each revision was not small. They were also used in previous research [16, 17, 22, 23].

Table 2: Target projects and their size

Project	loc^{latest}	$\|R\|$
ant	131,265	12,783
commons-io	25,031	1,526
eclipse.jdt.core	1,155,484	19,140
egit	92,305	3,126
jEdit	115,842	6,221
jboss-as	551,426	10,764
jetty	207,517	6,082
JGit	124,662	2,321
log4j	30,010	3,226
lucene-soir	537,150	8,026
maven	72,201	9,312
org.eclipse.cdt	1,029,497	21,157
org.eclipse.hudson.core	81,876	1,008
tomcat	240,086	9,172
Total	4,394,352	122,116

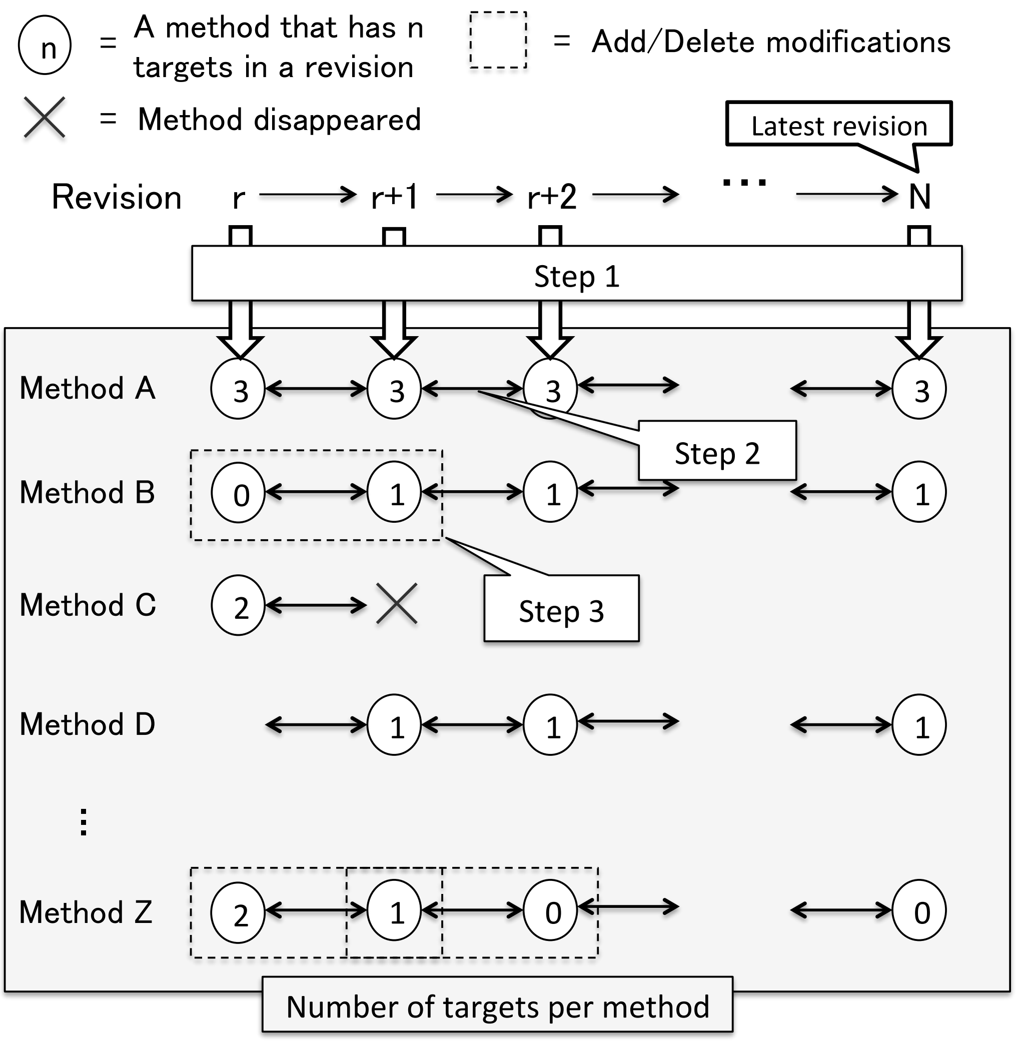


Figure 3: Overview of steps

3.3 Experimental Method

In this experiment, the input was a target repository, and the outputs were the followings:

- Ret^r_{null}, Ret^r_{not}, $Cond^r_{null}$, $Cond^r_{not}$, loc^r, for $\forall r \in R$
- ΔRet^c_{null}, ΔRet^c_{not}, $\Delta Cond^c_{null}$, $\Delta Cond^c_{null}$, for $\forall c \in C$

In order to collect the necessary data, we proceeded as follows. Figure 3 is an overview of the steps.

Step 1) We analyzed the source code in each revision to obtain the number of instances of $ret_{null/not}$ and $cond_{null/not}$ for each method. ret_{null} is only the literal sequence of tokens "return null," and $cond_{null}$ are conditional predicates comparing a variable to null, such as "a == null," and "null != a."

Step 2) By using the data obtained in Step 1, we found additions/deletions of $ret_{null/not}$ and $cond_{null/not}$. We regarded the changed number of instances of $ret_{null/not}$ and $cond_{null/not}$ in a method between two revisions to be additions/deletions, respectively. As we used the number of instances of $ret_{null/not}$ and $cond_{null/not}$, some changes were ignored in a case where some $ret_{null/not}$ were modified, but the number of instances of $ret_{null/not}$ in the method was not changed.

Step 3) We filtered out unnecessary additions/deletions. If $ret_{null/not}$ and $cond_{null/not}$ were added/deleted as a part of module additions/deletions, they were filtered out. This was done because the changes in numbers arising from adding/deleting modules are not commonly caused by bug fixes. In this experiment, when a method was added/deleted, we considered it to be an addition/deletion of the modules. In summary, our filtering omitted the additions/deletions of $ret_{null/not}$ and $cond_{null/not}$ from the experimental target if they satisfied any of the following conditions: their number in their owner method was not changed, their owner method disappeared, or their owner method appeared anew at a given commit.

We obtained the remains of this filtering as necessary modifications. In the example shown in Figure 3, we obtained only modifications where the numbers were changed "0 to 1," "2 to 1," and "1 to 0."

3.4 Calculating the Frequency of Modifications

To answer the research questions, it is necessary to compare the frequencies of modifications between $ret_{null/not}$ and $cond_{null/not}$.

$f_{ret_{null}}(C)$, $f_{ret_{not}}(C)$, $f_{cond_{null}}(C)$, and $f_{cond_{not}}(C)$ are the frequency of modifications to ret_{null}, ret_{not}, $cond_{null}$, and $cond_{not}$. They were calculated as follows:

$$f_{ret_{null}}(C) = \frac{\sum\limits_{c \in C} |\Delta Ret^{c}_{null}|}{|Ret^{latest}_{null}|} \tag{1}$$

$$f_{ret_{not}}(C) = \frac{\sum\limits_{c \in C} |\Delta Ret^{c}_{not}|}{|Ret^{latest}_{not}|} \tag{2}$$

$$f_{cond_{null}}(C) = \frac{\sum\limits_{c \in C} |\Delta Cond^{c}_{null}|}{|Cond^{latest}_{null}|} \tag{3}$$

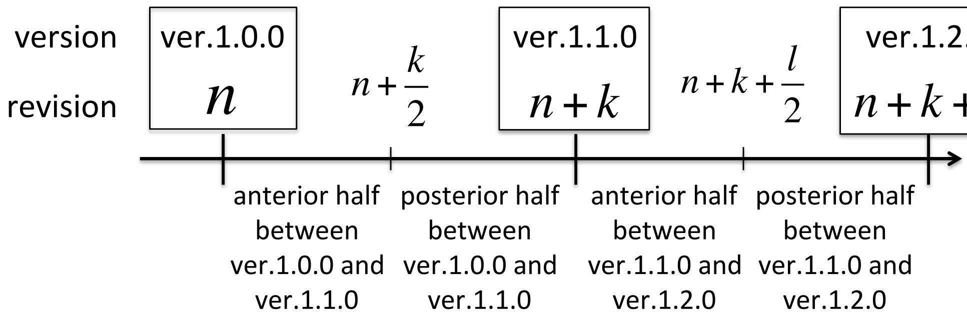

version
revision
ver.1.0.0
n
$n+\frac{k}{2}$
ver.1.1.0
$n+k$
$n+k+\frac{l}{2}$
ver.1.2.0
$n+k+l$
anterior half between ver.1.0.0 and ver.1.1.0
posterior half between ver.1.0.0 and ver.1.1.0
anterior half between ver.1.1.0 and ver.1.2.0
posterior half between ver.1.1.0 and ver.1.2.0

Figure 4: Overview of division

$$f_{cond_{not}}(C) \quad = \quad \frac{\sum_{c \in C} |\Delta Cond^{c}_{not}|}{|Cond^{latest}_{not}|} \tag{4}$$

For example, in Eq.1, the denominator is the total number of ret_{null} in the latest revision, and the numerator is the number of added/deleted ret_{null} throughout all the commits. The result is the frequency of modifications to one ret_{null}. By dividing $|Ret^{latest}_{null}|$, $|Ret^{latest}_{not}|$, $|Cond^{latest}_{null}|$, and $|Cond^{latest}_{nut}|$, we can reduce the influence arising from the difference in the number of modifications simply caused by the difference of their number.

3.5 Unit of Analysis

In this experiment, by using the caller/callee relationships of methods, we obtained the modifications to $cond_{null}$ in a caller method when ret_{null} was added to a callee method. As this research focuses on methods, $|\Delta Desc^{c}|$ for $\forall c \in C$ was calculated for each method.

3.6 Definition of Development Phases

We could not divide the development phases clearly because the experimental targets were open source projects. In this experiment, we considered a period between major version releases as one software development period, and we divided each of the development periods into an anterior half and a posterior half. Figure 4 shows the division of the periods between version 1.0.0 and version 1.2.0. We considered that modules had been added in the anterior half and bugs had been fixed in the posterior half. We performed such divisions, and calculated the frequency of modifications in the anterior half and the posterior half.

Division and statistical testing were conducted on 12 projects listed in Table 2 except for tomcat and eclipse.jdt.core. As the borders among major version releases were vague, we excepted those projects.

3.7 Statistical Testing Methodology

In order to answer the research questions, we needed to check whether two samples had a significant difference or a statistical correlation.

We used the Wilcoxon signed rank test [25] to determine whether two paired-samples had a significant difference. If the obtained p value is low, there is a low probability that the observed differences are accidental. In this experiment, the significance level is 1%. This means that there is a significant difference if $p \leq 0.01$, and there is no significant difference if $0.01 < p$.

Additionally, we calculated the Spearman's rank correlation coefficient (ρ value) to determine whether two paired-samples had a statistical correlation. After that, we calculated the p value corresponding to the ρ value. The two samples have a positive correlation if the ρ value is positive, and a negative correlation if the ρ value is negative. The meaning of the p value is the same as for the Wilcoxon signed rank test. Two samples have no correlation if $|\rho| < 0.5$, and they have a correlation if $0.5 \leq |\rho|$.

4 Answers to Research Questions

4.1 Answer to RQ1

To ascertain whether ret_{null} and $cond_{null}$ were modified more frequently than ret_{not} and $cond_{not}$, we tested the difference between them. Figure 5 shows a comparison between $f_{ret_{null}}(C)$ and $f_{ret_{not}}(C)$, $f_{cond_{null}}(C)$ and $f_{cond_{not}}(C)$. In the figure, black regions are $Desc_{null}$, and gray regions are $Desc_{not}$. In addition, the figure does not show a value just as it is, but the percentage for the value. For example, the percentage of $f_{ret_{null}}(C)$ is calculated as follows: $f_{ret_{not}}(C)$, $f_{cond_{null}}(C)$, and $f_{cond_{not}}(C)$ are the same as $f_{ret_{null}}(C)$.

$$\frac{f_{ret_{null}}(C)}{f_{ret_{null}}(C) + f_{ret_{not}}(C)} \tag{5}$$

The p value obtained from the Wilcoxon signed rank test between $f_{ret_{null}}(C)$ and $f_{ret_{not}}(C)$ is 2.44×10^{-4} (Figure 5(a)). That means $f_{ret_{null}}(C) - f_{ret_{not}}(C)$ in almost all the projects are positive values. As $p \leq 0.01$, ret_{null} was modified more frequently than ret_{not}. On the other hand, the p value between $f_{cond_{null}}(C)$ and $f_{cond_{not}}(C)$ is 0.714 (Figure 5(b)). Thus, $cond_{null}$ were not modified more frequently than $cond_{not}$.

Therefore, our answer to RQ1 is that ret_{null} was modified more frequently than ret_{not}, and $cond_{null}$ was not modified more frequently than $cond_{not}$.

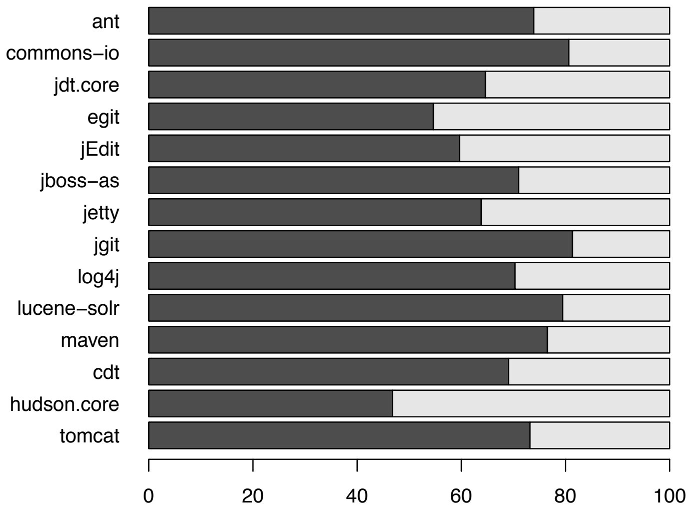
4.2 Answer to RQ2

To ascertain whether the size of projects and the frequency of modifications have any correlation, we assumed that loc^{latest} indicated the size of projects, and tested the correlation between them.

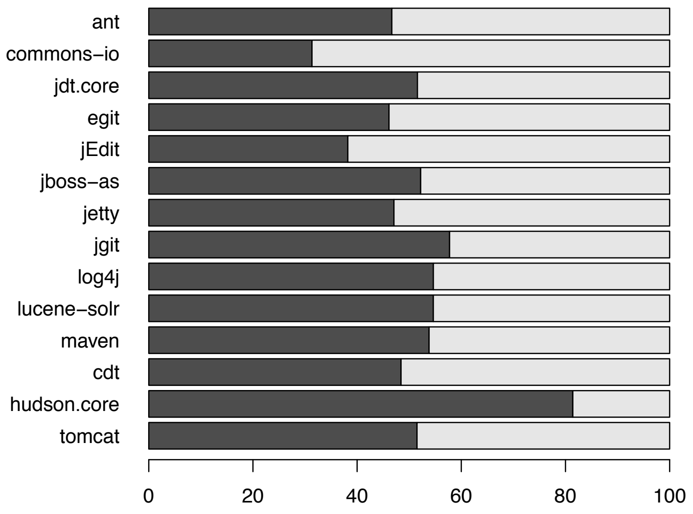
Figure 6 shows the scattergram between $f_{ret_{null}}(C)$, $f_{cond_{null}}(C)$ and loc^{latest}. The x-axis is loc^{latest}, and the y-axis is $f_{ret_{null}}(C)$ and $f_{cond_{null}}(C)$. We calculated that the Spearman's rank correlation coefficient (ρ value) and the p value corresponds to the ρ value.

The result from the Spearman's rank correlation test between $f_{ret_{null}}(C)$ and loc^{latest} was $\rho = -0.0330$, and $p = 0.916$ (Figure 6(a)). As $0.01 < p$, there was no significant correlation. Similarly, the result between $f_{cond_{null}}(C)$ and loc^{latest} was $\rho = 0.169$, and $p = 0.563$ (Figure 6(b)). As it was also $0.01 < p$, there was no significant correlation.

As a result, our answer to RQ2 is that the size of the projects did not have a significant effect on the frequency of modifications to ret_{null} and $cond_{null}$.

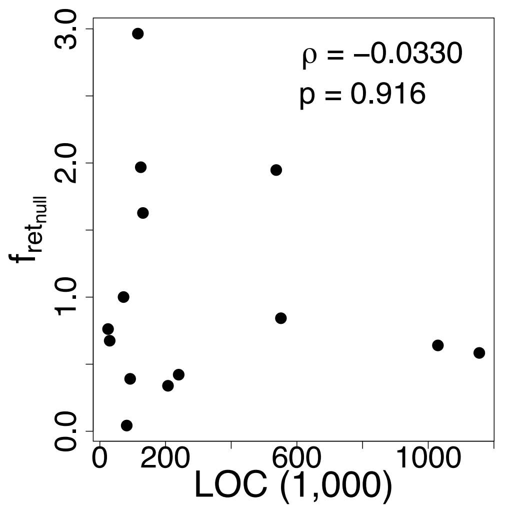


(a) Comparison between $f_{ret_{null}}(C)$(black) and $f_{ret_{not}}(C)$(gray)

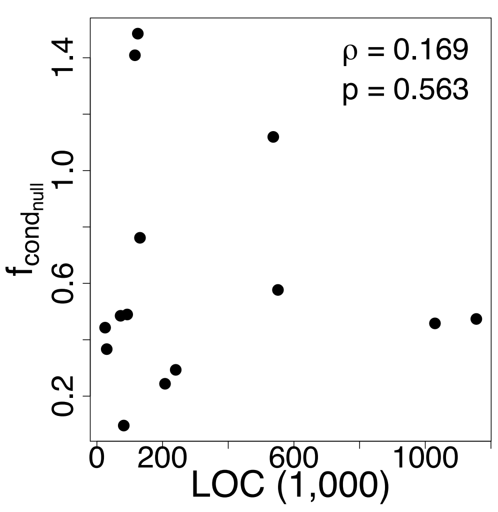


(b) Comparison between $f_{cond_{null}}(C)$(black) and $f_{cond_{not}}(C)$(gray)

Figure 5: Comparison between $f_{Desc_{null}}(C)$(black) and $f_{Desc_{not}}(C)$(gray)



(a) $f_{ret_{null}}(C)$



(b) $f_{cond_{null}}(C)$

Figure 6: Scattergrams of loc^{latest} and $f_{ret_{null}}(C)$, $f_{cond_{null}}(C)$

4.3 Answer to RQ3

There were 109 major versions among the target projects. After dividing them, we calculated $f_{ret_{null}}(C)$ and $f_{cond_{null}}(C)$ for the divided 218 periods. Figure 7 shows the calculated values.

We tested $f_{ret_{null}}(C)$ and $f_{cond_{null}}(C)$ for differences between the anterior half and the posterior half. The result from the Wilcoxon signed rank test between $f_{ret_{null}}(C)$ in the anterior half and the posterior half was $p = 0.126$ (Figure 7(a)). As $0.01 < p$, there was no significant difference. Similarly, the result between $f_{cond_{null}}(C)$ in the anterior half and the posterior half was $p = 0.383$ (Figure 7(b)). There was also no significant difference.

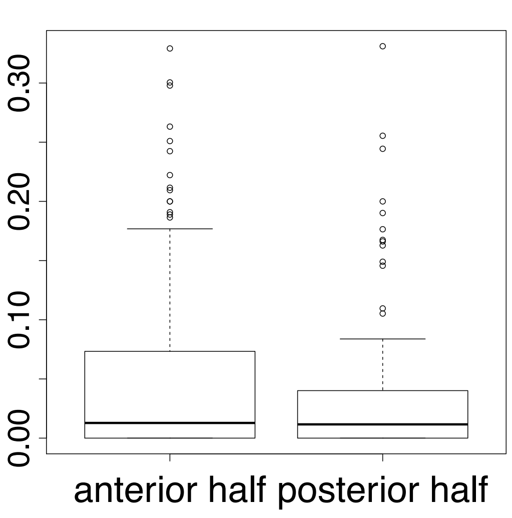
As a result, our answer to RQ3 is that the development phases did not have a significant effect on the frequency of modifications on ret_{null} and $cond_{null}$.

4.4 Answer to RQ4

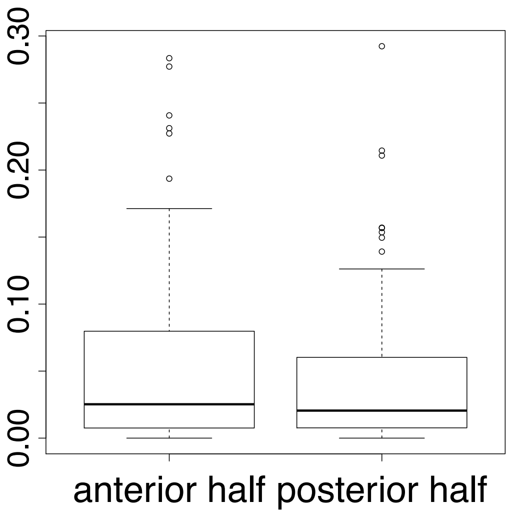
We calculated $density(Cond^{r}_{null})$, representing the number of $cond_{null}$ per line of code, for each revision r in the target projects. The value was calculated as follows:

$$density_{cond_{null}}(r) = \frac{|Cond^{r}_{null}|}{loc^{r}} \tag{6}$$

For each project, we tested whether there was a positive correlation between the revision number (r) and $density_{cond_{null}}(r)$. A positive correlation means that the density of $cond_{null}$ is increasing as a project proceeds. Table 3 shows the test result. The ρ value indicates the strength of



(a) $f_{ret_{null}}(C)$



(b) $f_{cond_{null}}(C)$

Figure 7: Box plot showing $f_{ret_{null}}(C)$ and $f_{cond_{null}}(C)$ of the anterior half and the posterior half

the correlation, and the p value indicates whether there is a significant correlation or not.

As a result, four projects showed significant positive correlations, five projects showed significant negative ones, and three projects had no correlation. Two projects did not have significant correlation because the p value of two projects is $p > 0.01$. This shows there was no regular pattern in the results.

Consequently, our answer to the RQ4 is that the density of $cond_{null}$ did not increase as the projects proceeded.

Table 3: Spearman's rank correlation coefficient (ρ value) and p value for revision number r and $density_{cond_{null}}(r)$

Software	ρ value	p value
ant	0.432	under 2.2×10^{-16} [a]
commons-io	0.733	under 2.2×10^{-16}
jdt.core	-0.974	under 2.2×10^{-16}
egit	0.223	under 2.2×10^{-16}
jEdit	0.668	under 2.2×10^{-16}
jboss-as	-0.765	under 2.2×10^{-16}
jetty	-0.992	under 2.2×10^{-16}
jgit	0.598	under 2.2×10^{-16}
log4j	0.006	0.740
lucene-solr	-0.947	under 2.2×10^{-16}
maven	0.934	under 2.2×10^{-16}
cdt	0.156	under 2.2×10^{-16}
hudson.core	-0.151	2.898×10^{-4}
tomcat	-0.986	under 2.2×10^{-16}

[a] This indicates the p value is too small to calculate a precise value.

5 Discussion

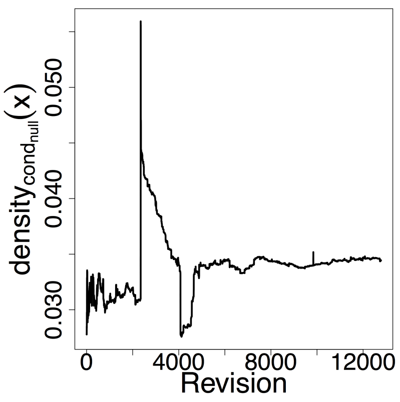
5.1 Discussion on the Results of RQ1, RQ2, and RQ3

Our answer to RQ1 is that ret_{null} was modified more frequently than ret_{not}. As ret_{null} was frequently modified, we can say that their presence is probably costly to maintain. Our answers to RQ2 and RQ3 are that the size and the development phases of projects did not affect the frequency of modifications to ret_{null} and $cond_{null}$. This means a developer modified ret_{null} and $cond_{null}$ throughout the entire development period of a project. Our answers above show that ret_{null} affects software maintenance throughout the entire development period of a project.

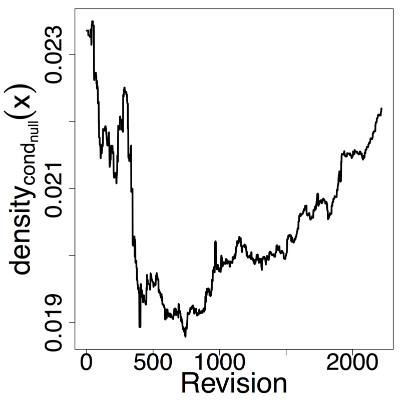
5.2 Discussion on RQ4

In our answer to RQ4 (see 4.4), we showed that many projects had positive or negative correlations between revision number r and $density_{cond_{null}}(r)$. We conducted additional experiments to find common characteristics in the density on $cond_{null}$.

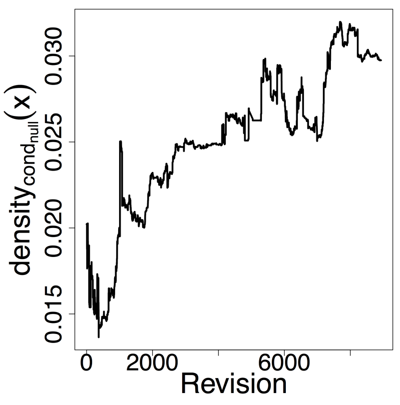
Figure 8 shows the relationship between revision numbers and $density_{cond_{null}}(r)$ in four



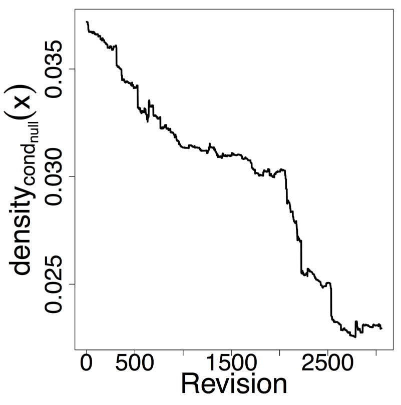
(a) ant



(b) egit



(c) maven



(d) jetty

Figure 8: Change in frequency of modifications to $cond_{null}$ in some projects. Tendencies of $cond_{null}$ differ in each project

projects. In Figure 8, the x-axis represents the revision number, and the y-axis represents $density(Cond^{x}_{null})$. These projects show that the continuation of the change has been maintained. This means that the density will have been decreasing/increasing, and it will become a very low/high value. However, inspecting the overall $density_{cond_{null}}(r)$ gives us another idea.

Figure 9 shows $density_{cond_{null}}(r)$ of $\forall r \in R$ in all the projects. Position x in the x-axis means the revision in the top x% of revisions sorted by ascending order of revision numbers. The y-axis is $density_{cond_{null}}(x)$. The thick bar shows the median of all the projects. This graph shows that $density_{cond_{null}}(r)$ of all the projects is in the range 0.01 to 0.04, and the median is stable regardless of the progress of their projects. In other words, the number of $cond_{null}$ is from one to four lines per 100 lines, and this density will be kept in the future of any project. This result also means that countermeasures to reduce ret_{null} and $cond_{null}$ have not been taken in all the projects.

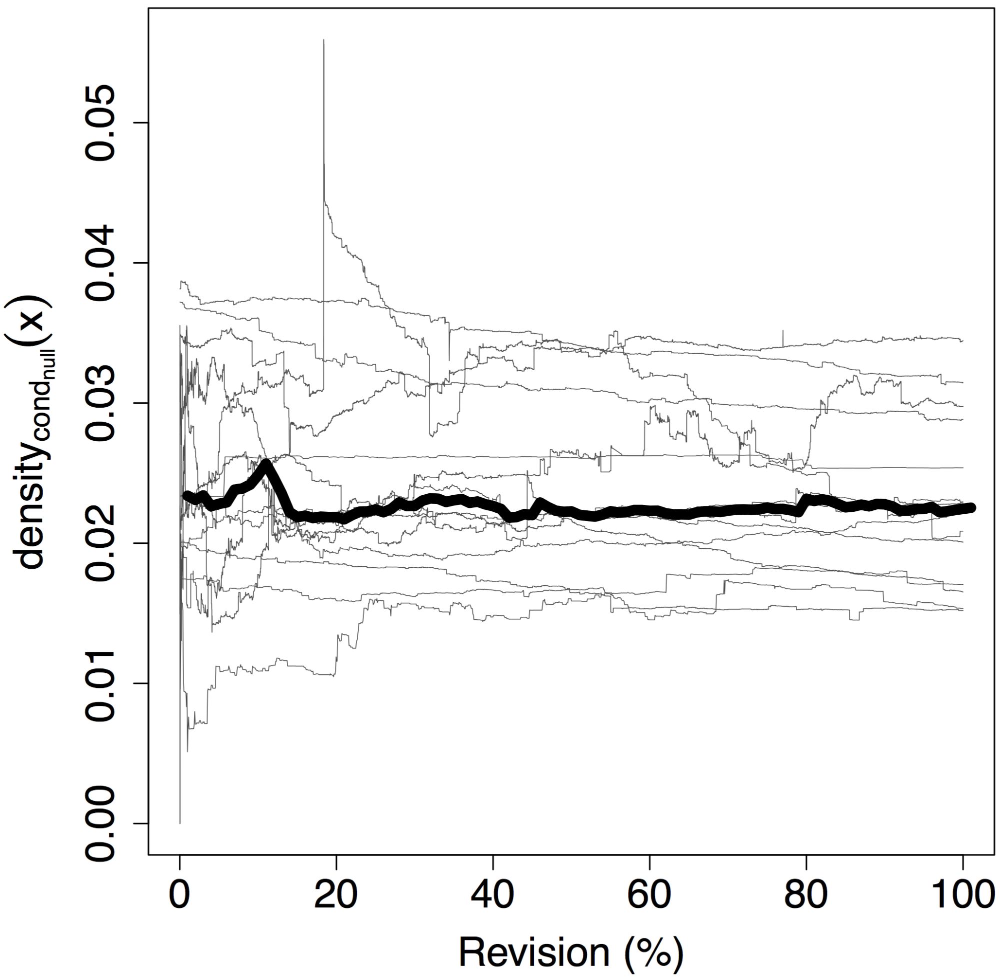


Figure 9: $density_{cond_{null}}(r)\ \forall r \in R$ of all projects. This figure shows that $density_{cond_{null}}(latest)$ of all projects are between 0.01 and 0.04

After we see Figure 9, we can recognize that the density reaches the area between 0.01 and 0.04 in all projects as shown in Figure 8. Once a density reaches the area, it is unlikely that the density will go out from the area. This is presumably a common characteristic of the density in $cond_{null}$.

In addition, if $density_{cond_{null}}(latest)$ in a project is high ($>$ 0.04), the project presumably has some problems. This means that the number of $cond_{null}$, which is not a functionality that we want to realize, has increased too much. Thus, if $density_{cond_{null}}(latest)$ is high, we suggest that the developer should refactor (e.g., check the existence of unnecessary $cond_{null}$, returning suitable values instead of $null$, and remove $cond_{null}$ by using a NullObject pattern).

```
+  if (prefixes == null) {
+      throw new IllegalArgumentException("The prefix must not be null");
+  }
```

Figure 10: Fix NPE

```
   // Now, we tokenize the string. The fourth element is what we want.
   StringTokenizer tok = new StringTokenizer(line2, " ");
   if (tok.countTokens() < 4) {
-      throw new IOException(
-         "Command line 'df' did not return data as expected for path '" +
-         path + "'- check path is valid");
+      // could be long Filesystem, thus data on third line
+      if (tok.countTokens() == 1 && line3 != null) {
+          line3 = line3.trim();
+          tok = new StringTokenizer(line3, " ");
+      } else {
+          throw new IOException(
+              "Command line 'df' did not return data as expected " +
+              "for path '" + path + "'- check path is valid");
+      }
+  } else {
+      tok.nextToken(); // Ignore Filesystem
   }
```

Figure 11: Added functionality

5.3 Discussion on the forgetting to write $cond_{null}$

As developers often forget to write $cond_{null}$, they regard adding $cond_{null}$ as a costly task. Thus, effects with a focus on adding $cond_{null}$ are worth evaluating.

The purpose of adding $cond_{null}$ is classified as follows.

Fix NPE (Null Pointer Exception) As developers forgot to add $cond_{null}$, they add $cond_{null}$ to prevent null dereference from occurring (Figure 10).

Add Functionality Developers added $cond_{null}$ to add new functionality (Figure 11).

Move Developers moved $cond_{null}$ from one method to another. Such movements could be detected as additions of $cond_{null}$ due to the limitation of our tool.

Flag Developers used $null$ as a flag (Figure 12).

Other Other reasons (Figure 13).

"Forgetting to write $cond_{null}$" is "fix NPE." If the number of additions classified as "fix NPE" is large, this is one of evidence that ret_{null} are harmful with regard to maintenance.

```
  public static Collection listFiles(File directory, String[] extensions,
    boo
-     String[] suffixes = toSuffixes(extensions);
-     IOFileFilter filter = new SuffixFileFilter(suffixes);
+     IOFileFilter filter;
+     if (extensions == null) {
+        filter = TrueFileFilter.INSTANCE;
+     } else {
+        String[] suffixes = toSuffixes(extensions);
+        filter = new SuffixFileFilter(suffixes);
+     }
      return listFiles(directory, filter,
          (recursive ? TrueFileFilter.INSTANCE : FalseFileFilter.INSTANCE
    ));
  }
```

Figure 12: Flag

```
  if (filename1 == null || filename2 == null) {
-     return filename1 == filename2;
+    return (filename1 == null && filename2 == null);
  }
```

Figure 13: Other reasons

Detecting "forgetting to write $cond_{null}$" from a repository automatically is a difficult task. We used the following steps to classify the additions of $cond_{null}$.

Step 1) Picking out commits that have no addition of ret_{null} (including addition/deletion of methods) and some addition of $cond_{null}$.

Step 2) Filtering out sets of source code for software testing in the commits.

Step 3) Inspecting residual files manually, and grouping them.

As Step 3 was costly, we evaluated only commons-io, which was the smallest project in our experimental targets. The number of residual commits after Step 2 is 27 in commons-io, and the number of added $cond_{null}$ is 55.

The result of Step 3 is the followings: the number of "Fix NPE" is 30 (55%), "Add Functionality" is 14 (25%), "Move" is seven (13%), "Flag" is two (4%), and "Other" is two (4%). "Fix NPE" is 55% in additions of $cond_{null}$, and this is 20% of all the changes of $cond_{null}$. This is a considerable number. In addition, unwritten $cond_{null}$ shows that developers were not able to find the bug in commit time. This means that forgetting $cond_{null}$ is a serious problem with respect to software maintenance.

As mentioned above, forgetting $cond_{null}$ is not a small percentage of modification and it is difficult to fix. In conclusion, unwritten $cond_{null}$ have a negative impact on maintenance. This means ret_{null}, which are the cause of $cond_{null}$, are an actual problem to be solved.

```
  } else {
-   return null;
+   if (merger.failedAbnormally())
+     return new CherryPickResult(merger.getFailingPaths());
+   // merge conflicts
+   return CherryPickResult.CONFLICT;
  }
```

Figure 14: Example of replacing ret_{null} into an error constant

```
  DirCacheCheckout dco;
- RevCommit commit = walk.parseCommit(repo.resolve(commitId));
+ if (commitId == null)
+   throw new JGitInternalException(
+     JGitText.get().abortingRebaseFailedNoOrigHead);
+ ObjectId id = repo.resolve(commitId);
+ RevCommit commit = walk.parseCommit(id);
  if (result.getStatus().equals(Status.FAILED)) {
    RevCommit head = walk.parseCommit(repo.resolve(Constants.HEAD));
    dco = new DirCacheCheckout(repo, head.getTree(),
```

Figure 15: Example showing lack of $cond_{null}$

6 Modifications to ret_{null} and $cond_{null}$

We selected modifications from $\Delta Desc^c$ in $\forall c \in C$ and we discussed the effect on the density of $cond_{null}$. In this study, we discussed the modifications to ret_{null} and $cond_{null}$ in JGit. The reason we targeted JGit was that the size was not too large, meaning we could check all the modifications, in addition the JGit project recorded information on bugs using a bug control system, Bugzilla [26]. We show the characteristic modifications and discussion.

Figure 14 shows an example of replacing ret_{null} into an error constant. This is a representative example of replacing ret_{null} with other mechanisms. This is necessary change to identify the errors that occurred in the same method. However, such a replacement (from $null$ to an error constant) often causes wide changes on source code because the returned value is changed. It is difficult to perform such a replacement without causing any problem in a fully automated way. Therefore, the presence of ret_{null} is costly to maintain.

Figure 15 shows the modification occurred in a commit whose commit message is "[findBugs] Don't pass $null$ for non-null parameter in RebaseCommand." This commit message shows that developers detected a null dereference by using FindBugs, and fixed the null dereference by adding $cond_{null}$. Many instances of null dereference occurred because developers had forgotten to write $cond_{null}$, but developers could easily detect this by using null dereference detection tools.

In Figure 16, as there was no $cond_{null}$ for the variable "objectID," developers added $cond_{null}$

```
@@ -108,7 +110,10 @@
  ObjectId objectId = repository.resolve(revstr);
- tree = new RevWalk(repository).parseTree(objectId);
+ if (objectId != null)
+   tree = new RevWalk(repository).parseTree(objectId);
+ else
+   tree = null;
  this.initialWorkingTreeIterator = workingTreeIterator;

@@ -125,9 +130,13 @@
  this.repository = repository;
- tree = new RevWalk(repository).parseTree(objectId);
+ if (objectId != null)
+   tree = new RevWalk(repository).parseTree(objectId);
+ else
+   tree = null;
  this.initialWorkingTreeIterator = workingTreeIterator;
```

Figure 16: Example of $cond_{null}$ being fixed in many places

and the error handling when "objectID" was $null$. The same problem was fixed in two places, but developers often forgot to fix them all. In such cases, null dereference detection tools are effective. In addition, we need to fix bugs in multiple places not only when a developer forgets to write $cond_{null}$, but also when a developer adds ret_{null} to existing methods.

As described above, developers can detect null dereferences by using null dereference detection tools and fix them. On the other hand, sometimes we can remove $null$ instead of simply adding $cond_{null}$. Figure 17 shows refactoring that removes $cond_{null}$ by assigning a suitable value "repo.lockDirCache()" to variable "dc" as an initialization. However, it is difficult to perform such refactorings because we need to analyze all expressions that include a variable so as to be able to assign a suitable value to a variable. In addition, there is no popular tool that assigns suitable values to variables automatically. Therefore, the number of instances of $cond_{null}$ rarely decreased. We can regard such a change as a factor of maintenance stem from $null$. Our experiment took such a change into consideration because our experiment did not focus on only addition of $cond_{null}$.

In summary, there were ret_{null} and $cond_{null}$ replaced with other mechanisms in order to avoid bad effects of ret_{null}. We can detect null dereference automatically by using null dereference detection tools, and developers add $cond_{null}$ when null dereference is detected. However, as it requires complex analyses, reducing the number of instances of ret_{null} and $cond_{null}$ is difficult both when using manual methods and when using automatic methods.

```
  private void resetIndex(RevCommit commit) throws IOException {
-   DirCache dc = null;
+   DirCache dc = repo.lockDirCache();
    try {
-     dc = repo.lockDirCache();
      dc.clear();
      DirCacheBuilder dcb = dc.builder();
      dcb.addTree(new byte[0], 0, repo.newObjectReader(), commit.getTree
    ());
      dcb.commit();
-   } catch (IOException e) {
-     throw e;
    } finally {
-     if (dc != null)
-       dc.unlock();
+     dc.unlock();
    }
  }
```

Figure 17: Example of removing $cond_{null}$ by modifying the default value

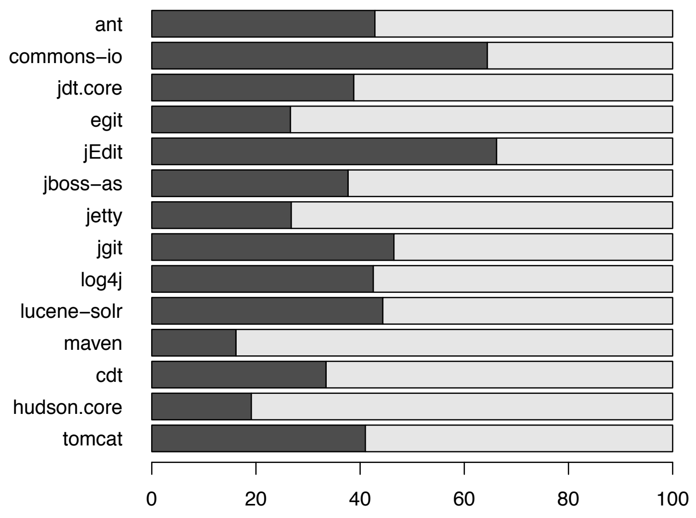
```
Object referencedObject = getTaskContext().getDataValue(reference);
if (referencedObject == null) {
  throw new ExecutionException("Unable to locate the reference specified by
     refid '" + getReference() + "'");
}
if (!this.getClass().isAssignableFrom(referencedObject.getClass())) {
  throw new ExecutionException("The object referenced by refid '" +
    getReference() + "' is not compatible with this element ");
}
```

Figure 18: Example of throw statements being written in multiple places

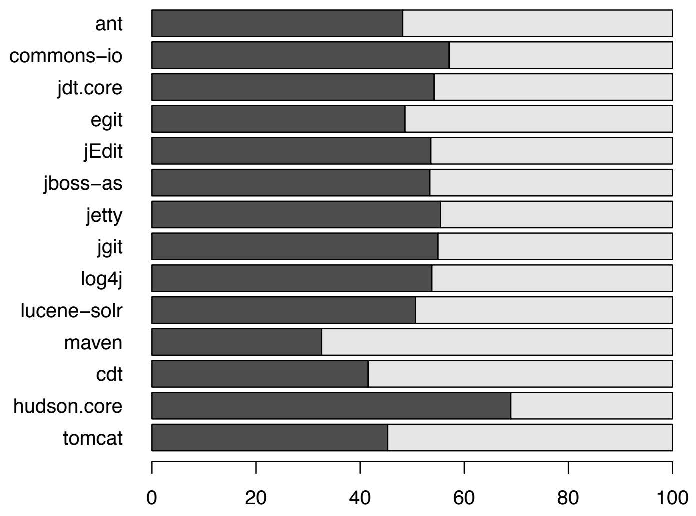
7 Replacing Return Null with Other Mechanisms

Exception handling is one of the mechanisms for replacing ret_{null}. We measured the frequency of modifications on throw statements and try statements. Then, we compared them with $f_{ret_{null}}(C)$ and $f_{cond_{null}}(C)$. Figure 19 shows the comparison result. The result of the Wilcoxon signed rank test between $f_{ret_{null}}(C)$ and throw statements was $p = 0.0245$. As $0.01 < p$, there was no significant difference. Similarly, the result between $f_{ret_{null}}(C)$ and try statements was $p = 0.390$. As it was also $0.01 < p$, there was no significant difference.

However, developers sometimes write multiple throw statements, all of which correspond to a single ret_{null}. This is because throw statements have error messages and exception types. Developers can change error messages and exception types depending on the reasons of errors (Figure 18 shows an example of such a situation). For this reason, the frequency of the modifications on throw statements became a larger value than the actual one. Therefore, it is not fair to compare the frequencies of the modifications between ret_{null} and exception handling. Exception handling



(a) Comparison between $f_{ret_{null}}(C)$(black) and the frequency of modifications on throw statements(gray)



(b) Comparison between $f_{cond_{null}}(C)$(black) and the frequency of modifications on try statements(gray)

Figure 19: Comparison between $f_{Desc_{null}}(C)$(black) and the frequency of modifications on exception handling mechanisms(gray)

is better than ret_{null} from the qualitative points of view, such as the examples in Section 6, and the characteristic that ret_{null} have no error message.

Replacing ret_{null} and $cond_{null}$ is related to research that is intended to detect null dereference, because detecting null dereference is the same as detecting passing $null$ from one method to another method and a variable can be assigned $null$. The information as to which variables can be assigned $null$ is very useful when it comes to replacing ret_{null} and $cond_{null}$. Therefore, the results of this research reinforce the motivation of those studies, e.g., "As null dereference occurs frequently, it is useful to detect them," with the motivation "Supporting replacing ret_{null} and $cond_{null}$ that affect software maintenance."

8 Threats to Validity

8.1 The fairness of comparison on RQ1

As stated in Section 7, it is not appropriate to compare ret_{null} with throw statements, because the frequency of modifications on throw statements was larger than the actual one. The unfairness also happened to ret_{not}. Although $f_{ret_{not}}(C)$ was presumably larger than the actual one, $f_{ret_{null}}(C)$ was larger than $f_{ret_{not}}(C)$ under adverse conditions (as shown in Section 4.1). Thus, our experimental results were not altered by the unfairness.

8.2 Code outside the methods

In this experiment, $\Delta Desc^c$ for $\forall c \in C$ were calculated for all methods. This calculation had the issue that $ret_{null/not}$ and $cond_{null/not}$ were ignored if $ret_{null/not}$ and $cond_{null/not}$ were outside the methods. However, $ret_{null/not}$ and $cond_{null/not}$ were rarely written outside the methods because $ret_{null/not}$ and $cond_{null/not}$ were return statements or conditional predicates. Thus, we excluded from consideration $ret_{null/not}$ and $cond_{null/not}$ that were outside the methods.

8.3 Changed/Moved code elements

$\Delta Desc^c$ for $c \in C$ consisted only of additions/deletions. Thus, we did not detect "Change" modifications, for example, a variable compared to $null$ had been changed to another variable. This may have an effect on the experimental result and the discussion. In addition, as our experimental steps could not detect a code move of $ret_{null/not}$ and $cond_{null/not}$, a code move caused by refactoring, such as Extract Method, was recognized as modifications of $ret_{null/not}$ and $cond_{null/not}$. A code move caused by refactoring should be filtered out, similarly to an addition/deletion of modules. However, in this experiment, we considered that the density of code moves was low, and ignored them.

8.4 Variables whose values are null

ret_{null} is only a return statement whose operand is $null$. If a return statement has a variable whose value is $null$, the statement is not counted. The situation is the same for $cond_{null}$. For this reason, the number of ret_{null} and $cond_{null}$ could differ from the actual number.

8.5 The division of the development phases

In the comparison of the development phases, we divided the periods between major versions into two periods, and tested whether they had a significant difference. However, as this division was performed based on the number of revisions, we could not obtain a precise division of the development phases. As a result, the experimental result could differ from the result when the periods were divided more precisely.

9 Related Work

Many previous studies have tackled null dereference problems from many viewpoints. Currently, there are many tools to detect null dereference automatically, which were introduced in Section 2. Those tools target existing null dereference.

By contrast, there are techniques that focus on preventing null dereference from occurring in compile time. One such technique is that developers add the constraint “null value is never assigned to this variable” (non-null) to some variables. This technique enables compilers to verify whether the variables could be $null$. As a result, these techniques can reduce mistakes in which a $null$ value is unintentionally assigned to some variables. In formal methods, developers represent a program as a formal specification using formal specification languages, such as JML (Java Modeling Language) [27], and check whether the constraints are violated or not by performing verifications. Developers can obtain proof that the variable will never become $null$ by using these methods. However, writing formal specifications needs a great deal of effort on the part of developers. Therefore, researchers tend to implement non-null features as the preferred type system and annotations. Fähndrich et al. proposed a NonNull type of system and implemented it on C# as an annotation [28]. Papi et al. extended the Java-type system and enabled checks to be made as to whether variables annotated $@NonNull$ can be $null$ or not [29]. There are many researchers that are trying to make an environment in which non-null can be used in a practical environment.

There also are methods that enforce null check on variables that can be $null$. Hovemeyer et al. developed $@CheckForNull$ annotation, and proposed a method that enforces null check on variables on Java [19]. Haskell [30] and Scala [31] have variables of *Maybe type* or *Option type* that represent variables that can be $null$. Variables of these types contain actual values or *Nothing* values, which are equivalent to a $null$ value. When developers use such variables, they spontaneously handle cases in which the variables have *Nothing* values. Such types of systems make developers aware of exception handling, and can detect the lack of a null check in compile time. As a result, such systems prevent null dereference from occurring.

In addition, $null$ as an initializer is a subject of research, because such an initializer is another factor in the occurring of null dereference. By using initialization analysis, which analyzes whether variable initializations have been done or not, developers can know which variables are uninitialized [32]. This decreases the possibility of null dereference occurring. In addition, there are techniques that track the origins of $null$ values [15]. As described above, many approaches have been proposed.

10 Conclusion

In this thesis, we investigated whether, and to what extent, the presence of return null and null check is costly to maintain by mining modifications during software evolution. This research focused on return null and null check, and an experiment conducted on 14 projects showed that return null were modified more frequently than statements that do not have $null$. In addition, we suggested that developers should avoid writing return null by reason of our qualitative analysis. Moreover, it was found that the frequencies of modifications on return null and null check bore no relation to the size and the development phases of projects. The density of null checks was from one to four in 100 lines, in each project. If the density of null check in a project exceeds 0.04, we can say that developers should refactor in order to decrease the density. A developer could replace return null with other mechanisms, for example, exception handling, a proper object such as an empty array, or a NullObject pattern.

We plan to research the reasons for writing return null and null check, and see to what extent we can lower the frequency of modifications. We can focus our experiment on "addition of null check." Results of such an experiment will be worth discussing. In addition, it is our future work to support developers by replacing return null and null check automatically. We also plan to find a method in order to compare return null with other error handling mechanisms quantitatively, and conduct the same experiment for other programming languages such as C.

Acknowledgment

During this work, I have been fortunate to have received assistance from many people. Without their guidance and valuable contributions, this work would not have materialized.

First, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, at Osaka University, for his considerate support, encouragement, and adequate guidance for this work.

Also, I would like to thank to Associate Professor, Kozo Okano, at Osaka University, for his valuable suggestions and discussions for this work.

I am also deeply grateful to Associate Professor, Hiroshi Igaki, at Osaka University, for his constructive comments and valuable suggestions.

I would like to show my greatest appreciation to Assistant Professor, Yoshiki Higo, at Osaka University, for his insightful comments, continuous support, and encouragement throughout this work.

Finally, I would like to thank all the members in Kusumoto Laboratory, at Osaka University, for their helpful advices, suggestions and assistance.

References

[1] T. Hoare, “Null References: The Billion Dollar Mistake,” in *Presented at QCon*, Mar. 2009.

[2] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, “Advances in Knowledge Discovery and Data Mining,” *The MIT Press*, 1996.

[3] M. J. Berry and G. Linoff, *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley & Sons, Inc., 1997.

[4] D. J. Hand, “Principles of data mining,” *Drug safety*, vol. 30, no. 7, 2007.

[5] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques*. Morgan kaufmann, 2006.

[6] M. Berry and G. Linoff, *Mastering data mining: The art and science of customer relationship management*. John Wiley & Sons, Inc., 1999.

[7] H. Kagdi, M. L. Collard, and J. I. Maletic, “A survey and taxonomy of approaches for mining software repositories in the context of software evolution,” *Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, 2007.

[8] M. B. Gethers, II, “Information Integration for Software Maintenance and Evolution,” Ph.D. dissertation, College of William & Mary, 2012.

[9] H. Hata, O. Mizuno, and T. Kikuno, “Historage: Fine-grained version control system for java,” in *Proceedings of the 12th International Workshop on Principles on Software Evolution and 7th ERCIM Workshop on Software Evolution (IWPSE-EVOL 2011)*, Sep. 2011.

[10] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, 2009.

[12] H. Hata, O. Mizuno, and T. Kikuno, “Fault-Prone Module Detection Using Large-Scale Text Features Based on Spam Filtering,” *Empirical Software Engineering*, vol. 15, no. 2, Apr. 2010.

[13] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, Sep. 2005.

[14] H. Hata, O. Mizuno, and T. Kikuno, "Bug Prediction Based on Fine-grained Module Histories," in *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*, Jun. 2012.

[15] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley, "Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors," *ACM SIGPLAN Notices*, vol. 42, no. 10, Oct. 2007.

[16] M. G. Nanda and S. Sinha, "Accurate Interprocedural Null-Dereference Analysis for Java," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, May 2009.

[17] N. Ayewah and W. Pugh, "Null Dereference Analysis in Practice," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2010)*, Jun. 2010.

[18] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, Jun. 2000.

[19] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, Sep. 2005.

[20] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda, "Verifying Dereference Safety via Expanding-scope Analysis," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, Jul. 2008.

[21] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007)*, Jun. 2007.

[22] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran, "Making Defect-finding Tools Work for You," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE 2010)*, 2010.

[23] D. Hovemeyer and W. Pugh, “Finding Bugs is Easy,” *SIGPLAN Notices*, vol. 39, no. 12, Dec. 2004.

[24] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for java,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, Jun. 2002.

[25] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods, 2nd Edition.* Wiley-Interscience, Jan. 1999.

[26] Bugzilla, “http://www.bugzilla.org/.”

[27] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary Design of JML: A Behavioral Interface Specification Language for Java,” *SIGSOFT Software Engineering Notes*, vol. 31, no. 3, May 2006.

[28] M. Fähndrich and K. R. M. Leino, “Declaring and checking non-null types in an object-oriented language,” in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA 2003)*, Oct. 2003.

[29] M. M. Papi and M. D. Ernst, “Compile-time Type-checking for Custom Type Qualifiers in Java,” in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA 2007)*, Oct. 2007.

[30] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, *The Haskell 98 Language Report*, Dec. 2002.

[31] M. Odersky, *The Scala Language Specication Version 2.9*, May 2011.

[32] F. Spoto and M. D. Ernst, “Inference of Field Initialization,” in *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, May 2011.