null返り値が保守作業に与える悪影響の調査

木村 秀平[1]　堀田 圭佑[1]　肥後 芳樹[1]　井垣 宏[1]　楠本 真二[1]

概要：オブジェクト指向言語を用いたプログラミングでは，メソッドの返り値として null 参照が用いられることがある (null 返り値)．null 返り値は，メソッド中の処理でプログラムが満たすべき条件が満たされなかったことを知らせるために広く利用されている．一方で，null を返す可能性のあるメソッドを呼び出す場合，呼び出し元で返り値が null か否かを判定する必要があり (null チェック)，保守作業を困難にしていることが知られている．しかし，ソフトウェア開発の中で，null 返り値がどのように保守作業に悪影響を与えるかは明らかにされていない．そこで本研究では，ソースコード中の null 返り値と null チェックを調査することで，それらがどのように保守作業に悪影響を与えているかについて調査を行った．本研究では，14 のオープンソースプロジェクトを対象として分析を行った．分析の結果，null 返り値および null チェックが，規模や作業工程の種別に関係なく頻繁に修正されることを確認した．修正頻度が高いということは，バグを含みやすくなると考えられるため，null 返り値および null チェックが保守作業に悪影響を及ぼしていることを示している．さらに，全分析対象のプロジェクトにおいて，null チェックは 100 行に 1 つから 4 つの割合で存在することが判明した．

キーワード：null 返り値，null 値参照，プログラム保守，リポジトリマイニング

Investigation into Negative Effects of Return Null on Software Maintenance

Shuhei Kimura[1]　Keisuke Hotta[1]　Yoshiki Higo[1]　Hiroshi Igaki[1]　Shinji Kusumoto[1]

Abstract: Developers often use null references for returned values of methods (return-null) in object-oriented languages. Although return-null is widely used to indicate that a program does not satisfy some necessary conditions, it is generally said that return-null makes software maintenance more difficult. One of the factors is: 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). However, how return-null affects software maintenance during software evolution has not been revealed yet. This paper reveals the influences of return-null by investigating return-null and null-check in evolution of source code. The experiment on 14 open source projects showed that developers modified return-null and null-check more frequently than the statements that did not include null. This result indicates that return-null makes software maintenance more difficult. On the other hand, the size and the development phase of projects have no effect on modification probabilities of return-null and null-check. In addition, we revealed that null-check appeared from one to four times in 100 lines.

Keywords: Return Null, Null Dereference, Program Maintenance, Repository Mining

1. はじめに

Java や C++といったオブジェクト指向プログラミング言語では，null 参照が存在する．変数の初期化時や番兵としての利用など，null は多くの場面で用いられている．中でも，メソッド内で満たすべき条件を満たさなかった際に null を返す記述が多数存在する (**null 返り値**)．エラー定数や例外処理を用いる場合に比べ，null 返り値を用いる利点

[1] Graduate School of Information Science and Technology, Osaka University

は以下のとおりである．

- null は特殊な値であるため，プログラムが満たすべき条件を満たさなかったことが一目でわかること．
- 満たさなかった条件に合わせたエラーメッセージを考える必要がないこと．
- 例外処理などに比べ，簡易な記述が行えること．

null 返り値はこのように開発者の助けとなる一方で，バグを発生させる原因として知られている [1]．null 返り値に関連する主なバグとして，**null 値参照**[*1]が挙げられる．null 値参照とは，null 参照が代入された変数を参照してしまうバグである．null を返す可能性のあるメソッドを呼び出した場合，呼び出し元で null が返ってきたかどうかの判定 (**null チェック**) が必要となる．そのため，null を返す可能性のあるメソッドが追加された場合，そのメソッドを呼び出す全ての箇所で null チェックを追加するか否かを判断する必要がある．null チェックを記述し忘れた場合，実行時例外を発生させるバグ (null 値参照) の原因となる．また，null という値は，型やメッセージなどでエラー情報を保持する仕組みを持った例外とは異なり，エラー情報を含まない．そのため，null を受け取った呼び出し元ではどのようなエラーが生じたかを判別しにくい場合がある．他にも，返り値が null であった場合に，さらに null を返す，といった null が伝搬するコードでは，エラーの原因を特定することが困難となってしまう．結果として，null 返り値は保守作業を困難にしていると考えられる．

しかし，null 返り値が保守作業に悪影響を与えていることを示した研究はない．そこで，本論文では null 返り値が保守作業に与える悪影響の調査を行った．本研究では，14 のプロジェクトに対しリポジトリマイニングを行い，null 返り値および null チェックに対して行われた修正を抽出した．抽出した修正回数および修正の内容から，null 返り値および null チェックが保守作業にどのように悪影響を与えているのかを結論付けた．

結果として，null 返り値および null チェックは保守作業を困難にし，以下の特徴を持っていることが判明した．

- null 返り値および null チェックは，それぞれ null を含まない return 文，条件式と比較して修正頻度が高い．
- プロジェクトの規模と null 返り値および null チェックの修正頻度には相関が無い．
- プロジェクトの作業工程は null 返り値および null チェックの修正頻度に影響を及ぼさない．
- プロジェクトの進行に伴ってソースコード中に占める null チェックの割合が増加するとはいえない．
- null チェックは，どのプロジェクトにおいても 100 行に 1 つから 4 つの割合で存在する．

以降，本論文は次のように構成されている． 2 章では本研究の背景となる関連研究について述べ， 3 章では研究課題の定義と，実験の設定について述べる． 4 章では実験結果と，結果を分析することにより得られた研究課題に対する回答を述べ， 5 章で null 返り値および null チェックが与える影響について議論する． 6 章では本研究における妥当性の脅威について述べ， 7 章で本論文をまとめる．

[*1] null dereference

2. 研究の動機

Java や C++ といったオブジェクト指向言語において，null 値参照は頻繁に発生するバグとして知られている．そのため，null 値参照を検出するための手法が多数研究されており [2–8]，FindBugs [9]，SALSA [6]，JLint，ESC/-Java [10] など，null 値参照を自動的に検出するツールが数多く公開されている [11]．しかし，これらの研究において，null 値参照がどのように保守作業に悪影響を与えるかについては言及されていない．

null 値参照を発生させる原因は，その値が null かどうかを判定せずに参照をしてしまうこと (null チェック不足) である．null チェック不足の原因として，呼び出したメソッドが null 返り値を含むにも関わらず，呼び出し元で null チェックを書き忘れてしまうことが挙げられる．図 1(b) は，JGit 中に存在したコードである．4 行目に記述されている command.call() は，図 1(a) に示したメソッドを呼び出している．このメソッドは，9 行目に記述されているとおり，null を返す可能性がある．図 1(b) において，4 行目で定義されている変数 ref は，command.call() で初期化を行なっているため，null が代入される可能性がある．そのため，5 行目の ref.getName() は null 値参照を発生させうる．図 1(c) は，この null 値参照に対するバグ修正である．+ が行頭にあるものが，このバグ修正によって追加された記述である．これは，5 行目の null チェックを追加することにより，null 値参照を発生させないようにしている．コミットメッセージは「Do not fail when checking out HEAD」であり，このことからも null 値参照に対する修正であることを示している．このように，null 返り値，null チェック，null 値参照は密接な関係にある．そのため，null 返り値および null チェックが与える影響を調査することで，どの程度 null 値参照が悪影響を及ぼしているかも調査することができると考えられる．

本研究では，null 返り値および null チェックを対象として保守作業に与える悪影響を調査した．-1 などのエラー定数も null 返り値および null チェックと同様の意味で用いられることが多い．しかし，特に Java では，null はどのオブジェクトにも代入可能であり，エラー定数より保守作業に悪影響を及ぼす可能性が高いと考えられる．そこで，本研究では null 返り値および null チェックのみを対象とした．

なお，本研究では，「修正が加えられることが多い」場合，保守作業へ悪影響を及ぼす可能性が高いと判断する．

```
1  public Ref call() throws GitAPIException,
       RefAlreadyExistsException,
       RefNotFoundException, InvalidRefNameException
       , CheckoutConflictException {
2    checkCallable();
3    processOptions();
4    try {
5      if (checkoutAllPaths || !paths.isEmpty()) {
6        checkoutPaths();
7        status = new CheckoutResult(Status.OK, paths
       );
8        setCallable(false);
9        return null;
10   }
11 ...
```

(a) null 返り値を含むメソッド

```
1  ...
2  try {
3    String oldBranch = db.getBranch();
4    Ref ref = command.call();
5    if (Repository.shortenRefName(ref.getName()).
       equals(oldBranch)) {
6      outw.println(MessageFormat.format(
7        CLIText.get().alreadyOnBranch,
8  ...
```

(b) (a) のメソッドを呼び出すコード

```
1  ...
2  try {
3    String oldBranch = db.getBranch();
4    Ref ref = command.call();
5  + if (ref == null)
6  +   return;
7    if (Repository.shortenRefName(ref.getName()).
       equals(oldBranch)) {
8      outw.println(MessageFormat.format(
9        CLIText.get().alreadyOnBranch,
10 ...
```

(c) null チェック追加後のコード

図 1: null 返り値と null 値参照，null チェックの例

Fig. 1 An example code fragments of return-null, null dereference, null-check

3. 実験の設定

この章では, null 返り値および null チェックに対する調査のために行った実験について説明する．なお, 以降では, 表 1 に示した略称を用いる．また, $|S|$ を集合 S の要素数とする．さらに, c をリビジョン r と $r+1$ の間のコミットとし，c で追加・削除された $return_{null}$, $return_{not}$, $cond_{null}$, $cond_{not}$ の集合をそれぞれ $\Delta Return^c_{null}$, $\Delta Return^c_{not}$, $\Delta Cond^c_{null}$, $\Delta Cond^c_{not}$ とする．

3.1 実験の目的

本実験の目的は，以下の研究課題に対して回答を行うことにより，null 返り値および null チェックが保守作業に与える悪影響を調査することである．

RQ1 $return_{null}$ および $cond_{null}$ は，$return_{not}$ および $cond_{not}$ と比べて修正される頻度が高いか？

RQ2 プロジェクトの規模は，$return_{null}$ および $cond_{null}$ の修正頻度に影響を与えるか？

RQ3 あるバージョンをリリースするにあたり，その開発の前期と後期で，$return_{null}$ および $cond_{null}$ の修正頻度に差はあるか？

RQ4 ソースコード中に占める $cond_{null}$ の割合はプロジェクトの進行に伴って増加するか？

3.2 実験対象

対象としたプロジェクトを表 2 に掲載する．これらは，Java 言語で記載され，git で管理されている，一定以上の規模を持つプロジェクトである．

3.3 実験手法

本実験では，各実験対象から以下の情報を得る．

- $\forall r \in R$ について，$Return^r_{null}$, $Return^r_{not}$, $Cond^r_{null}$, $Cond^r_{not}$, loc^r, moc^r
- $\forall c \in C$ について，$\Delta Return^c_{null}$, $\Delta Return^c_{not}$, $\Delta Cond^c_{null}$, $\Delta Cond^c_{not}$

3.3.1 手順の詳細

これらの情報を取得するため，リポジトリに対し以下の手順を行う．なお図 2 は実験手順を図示したものである．

Step1 各リビジョンで，メソッドごとに $return_{null}$, $return_{not}$, $cond_{null}$, $cond_{not}$ の個数を計測する．

Step2 Step1 で算出した個数を用いて，次のリビジョンで新たに追加された，または，前のリビジョンで存在して

表 1: 実験で用いる略称

Table 1 Abbreviations

略称	その説明
$return_{null}$	オペランドが null である return 文
$return_{not}$	オペランドが null でない return 文
$cond_{null}$	null との比較を行なっている条件式
$cond_{not}$	null 以外と比較を行なっている条件式
$Return^r_{null}$	リビジョン r における，$return_{null}$ の集合
$Return^r_{not}$	リビジョン r における，$return_{not}$ の集合
$Return^r$	$Return^r_{null} \cup Return^r_{not}$
$Cond^r_{null}$	リビジョン r における，$cond_{null}$ の集合
$Cond^r_{not}$	リビジョン r における，$cond_{not}$ の集合
$Cond^r$	$Cond^r_{null} \cup Cond^r_{not}$
$Desc^r_{null}$	$Return^r_{null} \cup Cond^r_{null}$
$Desc^r_{not}$	$Return^r_{not} \cup Cond^r_{not}$
$Desc^r$	$Desc^r_{null} \cup Desc^r_{not}$
C	実験対象の指定範囲におけるコミットの集合
R	実験対象の指定範囲におけるリビジョンの集合
loc^r	リビジョン r における，ソースコードの行数
moc^r	リビジョン r における，メソッド数
$latest$	指定範囲内で最新のリビジョン

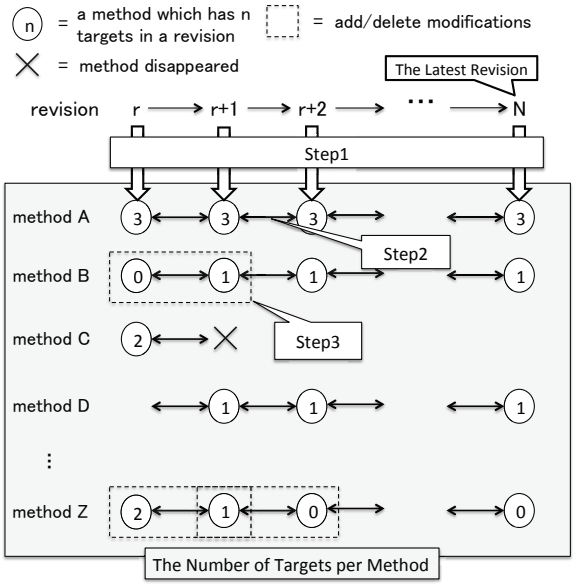


図 2: 修正回数の算出方法
Fig. 2 The overview of the steps

いたが削除された $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ を特定する．メソッドごとに $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ の個数を直前のリビジョンと比較し，個数が変化していれば追加・削除があったとみなす．個数を用いているため，直前のリビジョンで存在している $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ の (メソッド内での) 位置が変更された場合や，メソッド内で追加された同数削除された場合には，特定されない．

Step3 Step2 で特定した，個数の変化が生じた $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ から，本実験で不要なものを除外する．本実験で不要な追加・削除とは，機能を追加・削除したことに伴って追加・削除された $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ である．なぜなら，機能の追加・削除によって個数が変化した場合，バグ修正のためにそれらの数が変化したのではないためである．この実験では，メソッドそのものが追加・削除された場合は，それらを機能の追加・削除とみなし，除外した．これにより，機能の追加・削除のみでなく，Move Method リファクタリングなどによって，メソッドそのものが移動した場合に伴って発生した $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ の追加・削除についても除外することができる．

このようなフィルタリングを行い，残った追加・削除を，研究課題への回答に必要な情報として記録する．図 2 の例では，個数が「0 から 1」「2 から 1」「1 から 0」へと変化した場合のみを記録する．個数が変化しなかった場合，メソッドごと消滅した場合，または新しくメソッドが出現した場合はそれに伴って発生した $return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ の追加・削除は記録しない．

表 2: 対象としたプロジェクトとその規模
Table 2 Target projects and their size

Project	loc^{latest}	$\|R\|$
ant	131,265	12,783
commons-io	25,031	1,526
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-solr	537,150	8,026
maven	72,201	9,312
cdt	1,029,497	21,157
hudson.core	81,876	1,008
tomcat	240,086	9,172
Total	4,394,352	122,116

3.3.2 修正頻度の算出

研究課題への回答を行うために，$return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ それぞれの修正頻度を比較する必要がある．本研究において，$return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ の修正頻度 $f_{return_{null}}(C)$，$f_{return_{not}}(C)$，$f_{cond_{null}}(C)$，$f_{cond_{not}}(C)$ は，全コミットを通して修正された回数を合計し，その値を最新リビジョンにおけるそれぞれの個数で割った値とする．最新リビジョンでの個数で割ることにより，個数の差による影響を軽減することができる．これを式で表すと，以下のとおりとなる．

$$f_{return_{null}}(C) = \frac{\sum_{c \in C} |\Delta Return^{c}_{null}|}{|Return^{latest}_{null}|} \quad (1)$$

$$f_{return_{not}}(C) = \frac{\sum_{c \in C} |\Delta Return^{c}_{not}|}{|Return^{latest}_{not}|} \quad (2)$$

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

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

例えば，式 (1) では，全コミットで追加・削除された $return_{null}$ の個数を，最新リビジョンにおける $return_{null}$ の個数で割っている．そのため，$f_{return_{null}}(C)$ は 1 つの $return_{null}$ が変更される頻度を示す．

3.4 解析単位

本研究では，メソッドの呼び出し関係を用いて，$return_{null}$ が加わった際に呼び出し元で $cond_{null}$ がどのように変更さ

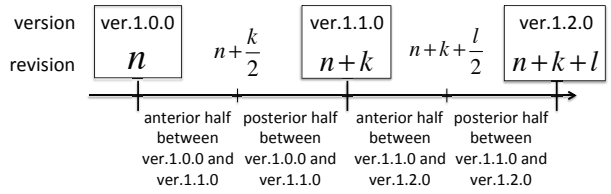


図 3: バージョン間分割の例

Fig. 3 The overview of the division

れたかを取得している．そのため，$return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$ に対して追加・削除が行われた個数もメソッド単位での計測を行った．

3.5 作業工程の分割

実験対象がオープンソースソフトウェアであるため，明確に作業工程を分けることはできない．そのため，本研究では，あるメジャーバージョンのリリースから次のメジャーバージョンのリリースまでの期間を 1 つのソフトウェア開発期間として想定し，その期間を前半と後半に分割した．図 3 では，バージョン 1.0.0 とバージョン 1.1.0 の期間を分割し，さらに，バージョン 1.1.0 からバージョン 1.2.0 の期間を分割した例を示している．前半では主に機能追加による修正が，後半では主にバグに関する修正が行われると想定した．このような分割を行い，前半と後半それぞれに対して修正頻度を求めた．

作業工程の分割および検定は，tomcat，jdt.core を除く 12 プロジェクトに対して行った．この 2 つのプロジェクトは，リポジトリにメジャーバージョンのタグが付けられておらず，開発の区切りが明らかでなかったため除外した．

3.6 検定方法

研究課題に回答するためには，二群間に差があるか否か，また，二群間に相関があるか否かを検定する必要がある．検定方法は，ノンパラメトリック検定の中で，この分野でよく用いられる方法を採用した．

まず，二群間に差があるか否かは，ウィルコクソンの符号順位検定 [12] を行い判定した．検定によって求められた p 値が低ければ，差が無いことが偶然である可能性が低いことを表している．本実験では，有意水準を 1% と定め，$p \leq 0.01$ であれば二群間に有意な差があり，$0.01 < p$ であれば有意な差は無い，とした．二群間に相関があるか否かは，スピアマンの順位相関係数 ρ を求め，それに対する有意性検定を行い p 値を求めて判定した．相関係数 ρ が正の値では正の相関を持ち，ρ が負の値では負の相関を持つ．p 値に対する棄却域の設定は，ウィルコクソンの符合順位検定と同様である．また，相関の強さは，$|\rho| < 0.5$ では相関が無く，$0.5 \leq |\rho|$ の場合，相関があると定めた．

4. 研究課題に対する回答

4.1 回答を行うために用いるデータ

14 のプロジェクト全体に対して実験を行い，$return_{null}$，$return_{not}$，$cond_{null}$，$cond_{not}$，および $\Delta Return^{c}_{null}$，$\Delta Return^{c}_{not}$，$\Delta Cond^{c}_{null}$，$\Delta Cond^{c}_{not}$ を算出した．

各プロジェクトに対する $f_{return_{null}}(C)$ と $f_{return_{not}}(C)$，$f_{cond_{null}}(C)$ と $f_{cond_{not}}(C)$ の比較を図 4 に示す．黒く塗られた棒は $Desc_{null}$ を示し，灰色に塗られた棒は $Desc_{not}$ を示す．なお，この図では，全プロジェクトでの比較を行いやすくするため，個数ではなく割合で表示している．例えば，$f_{return_{null}}(C)$ の占める割合は，下記の式により表され，$f_{return_{not}}(C)$，$f_{cond_{null}}(C)$，$f_{cond_{not}}(C)$ それぞれの占める割合に関しても同様に求めることができる．

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

また，実験対象プロジェクトには，109 のメジャーバージョンが格納されていた．メジャーバージョン間を二分割し，分割された 218 の期間それぞれに対して $f_{return_{null}}(C)$ および $f_{cond_{null}}(C)$ を取得した．結果を箱ひげ図として表した図を図 5 に示す．

4.2 RQ1 に対する回答

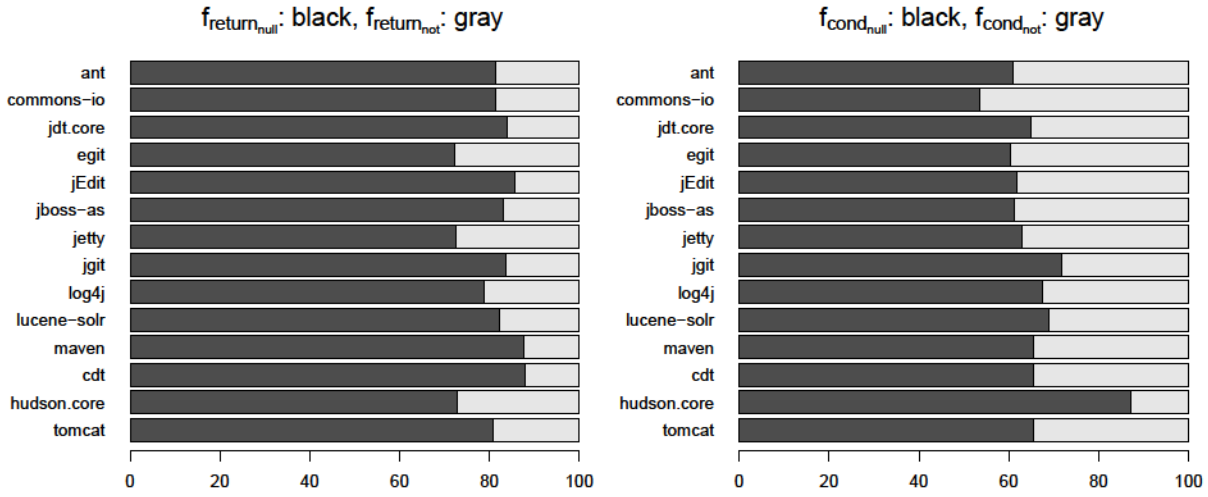
$return_{null}$ および $cond_{null}$ が，それぞれ $return_{not}$ および $cond_{not}$ に比べ頻繁に修正されるかを確かめるため，検定を行った．

$f_{return_{null}}(C)$ と $f_{return_{not}}(C)$ 間に差があるか否かを判定するため，ウィルコクソンの符号順位検定により p 値を求めたところ，$p = 1.22 \times 10^{-4}$ となった．これは，$n = 14$ の場合の，ウィルコクソンの符合順位検定で示される最小値である．すなわち，$f_{return_{null}}(C) - f_{return_{not}}(C)$ は全てのプロジェクトで正の値を示した．$p \leq 0.01$ であるため，$return_{null}$ は $return_{not}$ に比べ頻繁に修正されるといえる．同様に，$f_{cond_{null}}(C)$ と $f_{cond_{not}}(C)$ に対する検定においても，$p = 1.22 \times 10^{-4}$ となった．これは，$p \leq 0.01$ であるため，$cond_{null}$ は $cond_{not}$ に比べ頻繁に修正されるといえる．

これらの結論により，RQ1 への回答は，$return_{null}$ および $cond_{null}$ は $return_{not}$ および $cond_{not}$ と比べて頻繁に修正される，となった．

4.3 RQ2 に対する回答

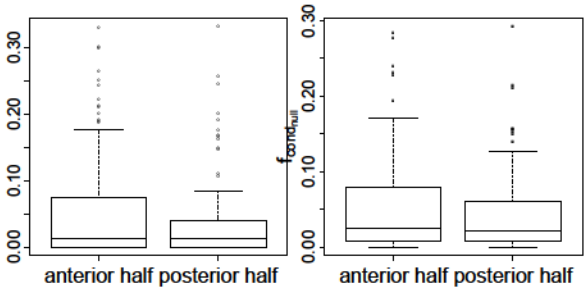
プロジェクトの規模と修正頻度に相関があるか否かを確かめるため，loc^{latest} がプロジェクトの規模を反映していると仮定し，検定を行った． 4.1 章の結果を，loc^{latest} を x 軸に，$f_{return_{null}}(C)$ および $f_{cond_{null}}(C)$ を y 軸に取り，散布図に表した図を図 6 に示す．また，スピアマンの順位相関係数 ρ を算出し，それに対する p 値を求めた．



(a) $f_{return_{null}}(C)$ と $f_{return_{not}}(C)$ の比較　(b) $f_{cond_{null}}(C)$ と $f_{cond_{not}}(C)$ の比較

図 4: $f_{Desc_{null}}(C)$(黒) と $f_{Desc_{not}}(C)$(灰) の割合比較

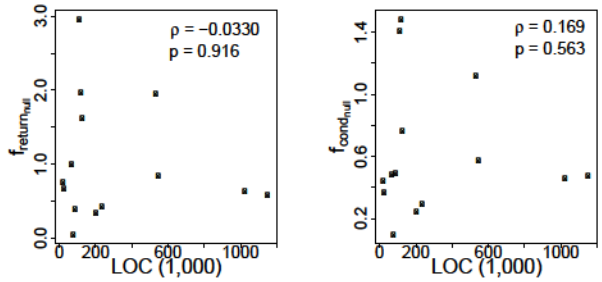
Fig. 4 Comparison between $f_{Desc_{null}}(C)$(black) and $f_{Desc_{not}}(C)$(gray)



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

図 5: バージョンの前半と後半で算出した $f_{return_{null}}(C)$ と $f_{cond_{null}}(C)$ の分布

Fig. 5 The box-plot that indicates $f_{return_{null}}(C)$ and $f_{cond_{null}}(C)$ of the anterior half and the posterior one



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

図 6: 各プロジェクトでの $f_{return_{null}}(C)$, $f_{cond_{null}}(C)$ と，プロジェクトの規模との相関

Fig. 6 Scattergrams of loc^{latest} and $f_{return_{null}}(C)$, $f_{cond_{null}}(C)$

プロジェクトの規模と $f_{return_{null}}(C)$(図 6(a)) に対して．$\rho = -0.0330$, $p = 0.916$ が得られた．これは，$0.01 < p$ であるため，相関は無いと判定できる値である．また，プロジェクトの規模と $f_{cond_{null}}(C)$(図 6(b)) に対しては，$\rho = 0.169$, $p = 0.563$ が得られた．同様に，$0.01 < p$ であるため，相関は無いと判定できる値である．

以上より，RQ2 に対する回答は，$return_{null}$ および $cond_{null}$ の修正頻度は，プロジェクトの規模による影響を受けない，といえる．

4.4 RQ3 に対する回答

図 5 に示した前半と後半それぞれの修正頻度を用いて，プロジェクト開発の前期と後期で $f_{return_{null}}(C)$ と $f_{cond_{null}}(C)$ の修正頻度に差は無いか否か検定を行った．

ウィルコクソンの符合順位検定を行ったところ，$return_{null}$(図 5(a)) に対する検定では，$p = 0.126$ となった．この値は $0.01 < p$ であるため，前半と後半の $f_{return_{null}}(C)$ に有意な差は存在しなかった．また，$cond_{null}$(図 5(b)) に対する検定では，$p = 0.383$ となり，$0.01 < p$ であるため同様に有意な差は存在しなかった．

以上より，RQ3 に対する回答は，$return_{null}$ と $cond_{null}$ は開発の前期と後期で修正頻度に差は無い，となった．

4.5 RQ4 に対する回答

対象プロジェクトの各リビジョン r に対して，1 行に存在する $cond_{null}$ の数を表す $density_{cond_{null}}(r)$ を，以下の式を用いて算出した．

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

プロジェクトごとに，リビジョン番号 r と $density_{cond_{null}}(r)$ との間に正の相関があるか否か，に対して検定を行った．リビジョン番号 r と $density_{cond_{null}}(r)$ が正の相関を持つということは，プロジェクトの進行に

伴って $cond_{null}$ の密度が増加してゆくことを示す．検定を行った結果を表 3 に表す．ρ は相関の強さを表し，p は統計的に有意な差があるか否かを表している．

結果として，4 つのプロジェクトは有意な正の相関を示し，5 つのプロジェクトは有意な負の相関を示し，3 つのプロジェクトは相関が無いことが示された．なお，2 つのプロジェクトは，$p > 0.01$ であるため統計的に有意な相関が示されなかった．このように，正の相関を持つか負の相関を持つかはプロジェクトによってまちまちであり，共通して有意な正の相関を持つことは無かった．

これにより，RQ4 への回答は，コード中に占める $cond_{null}$ の割合はプロジェクトの進行に伴って増加するとはいえない，となった．

5. 議論

5.1 RQ の結果に対する考察

RQ1 に対する回答から，$return_{null}$ および $cond_{null}$ は $return_{not}$ および $cond_{not}$ と比べて頻繁に修正されることがわかった．すなわち，$return_{null}$ および $cond_{null}$ を多く含むコードは修正されやすくなるため，結果として保守作業に悪影響を及ぼす可能性があるといえる．RQ2，RQ3 に対する回答から，プロジェクトの規模および開発の時期は，$return_{null}$ および $cond_{null}$ の修正頻度に影響を及ぼさないことがわかった．このことは，様々なプロジェクトの様々な時期において，$return_{null}$ および $cond_{null}$ の修正が行われることを示している．RQ1 の結果と合わせると，$return_{null}$ および $cond_{null}$ は，常に保守作業に悪影響を及ぼす要因となるといえる．

結果として，null 返り値および null チェックは，プログラム保守の観点から置き換えるべきであると判断できる．なお，null 返り値および null チェックの置き換えは，null 値参照を検出する研究と密接な関係がある．なぜなら，null 値参照を検出することは，null の伝搬および null となりうる変数を検出することであり，これらの情報は，null 返り値および null チェックを他の記述に置き換える際に非常に有用であるためである．そのため，null 値参照を検出する研究の動機として，「null 値参照は数多く発生するバグであり，そのバグを検出し修正することは有用」という既存の動機に加え，「保守作業に影響を与える null 返り値および null チェックを，他の記述に置き換えることの支援」という動機を加えることができるといえる．

表 3: 対象プロジェクトのリビジョン番号 r と $density_{cond_{null}}(r)$ に対するスピアマンの順位相関係数 ρ，および ρ に対する有意性判定値 p

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

Software	ρ value	p value
ant	0.432	under 2.2×10^{-16} *2
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}

*2 小さすぎて正確な値を表せないことを示す

5.2 $cond_{null}$ の占める割合に関する考察

RQ4 では，表 3 に示したとおり，多くのプロジェクトは修正頻度とリビジョン番号の間に有意な相関を持っていることを示した．相関があるということは，開発が進んだ際に，今までと同じ増加・減少・変化なしの傾向が現れる可能性が高いことを表す．また，各プロジェクトが持つ相関の正負は異なっており，プロジェクト間で共通する傾向は見いだせなかった．

図 7 は，全プロジェクトの $r \in R$ において，$density_{cond_{null}}(r)$ を図示したものである．x 軸は，リビジョン r がそのプロジェクトの全リビジョン番号で占める割合，y 軸は $density_{cond_{null}}(r)$ を表している．また，薄い点は各プロジェクトの値を，濃い点は全プロジェクトの中央値を示している．このグラフから，ほぼ全てのプロジェクトにおいて $density_{cond_{null}}(r)$ は 0.01 から 0.04 の間に収束しており，また，中央値はプロジェクトの進行に関わらずほぼ一定の値となっていることがわかる．このことから，$cond_{null}$ は 100 行に 1 つから 4 つ程度の割合で存在し，将来的にもこの割合が保たれると考えられる．

6. 妥当性への脅威

本実験では，計測要素の個数をメソッド単位で計測している．この手法は，計測要素がメソッドに含まれない場合に情報が取得できないという問題点がある．しかし，個数を取得する対象は return 文と条件式であるため，メソッド外に配置されることは少ない．そのため，本実験ではメソッド外に存在する計測要素は考慮しないものとした．

$return_{null}$ に対する修正として，追加および削除によって個数が変化する場合のみ検出している．そのため，$cond_{null}$ において，null との比較を行う変数が修正された場合などは計測していない．また，本実験手法では計測要素の移動を検知できないため，Extract Method などのリファクタリングによる移動も計測要素の修正と認識してしまう．こ

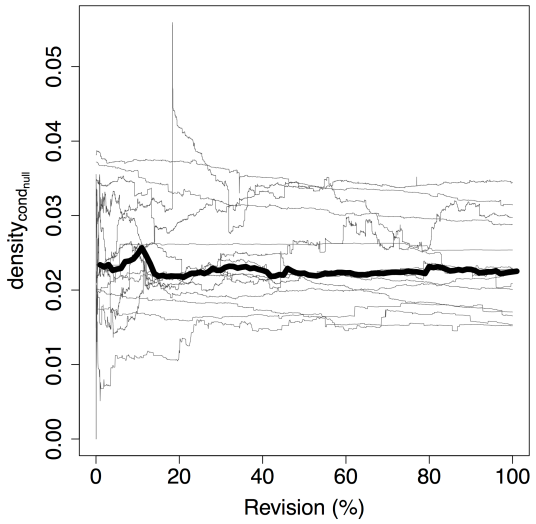


図 7: 全プロジェクトの $r \in R$ における，$density_{cond_{null}}(r)$ の推移 (各プロジェクトごとに，左端が最初のリビジョン，右端が最新のリビジョンを表す)

Fig. 7 $density(Cond^{r}_{null})$ of $\forall r \in Revisions$ in all projects

のようなリファクタリングは，機能追加の場合と同様に修正回数から取り除くべきである．しかし，本実験では，このような移動の占める割合は少ないものとして，考慮せず実験を行った．

本実験では，$return_{null}$ として計上されるものは，return 文のオペランドに直接 null が記述された場合のみである．変数に null が代入され，その変数を return している場合など，null が return される可能性のある文は数に加えていない．$cond_{null}$ に関しても同様である．このことにより，本実験で計上した個数は，実際の $return_{null}$ 数および $cond_{null}$ 数と乖離してしまう恐れがある．

作業工程の比較を行う際，メジャーバージョン間を二等分し，その 2 つの間で差があるかどうかの比較を行った．しかし，これはリビジョン数のみを利用した分割であり，正確に作業工程に基づいた分割を行っているわけではない．そのため，正しい作業工程で分割を行った場合と異なる結果となる恐れがある．

本研究は，対象とするプロジェクトによって結果が変動する可能性がある．プロジェクトによる差分を吸収するため，より多くのプロジェクトを用意することによって，さらに一般的な結果が得られると考えられる．

7. まとめ

本論文では，null 返り値が保守作業に悪影響を与えているのかを調査するため，null 返り値と null チェックに焦点を当て分析を行った．14 プロジェクトのリポジトリから null 返り値および null チェックに対する修正を抽出した結果，null 返り値および null チェックは，null を含まない記述と比べて頻繁に修正されることが示された．また，null 返り値および null チェックの修正されやすさは，プロジェクトの成熟度や，開発の段階に影響を受けない事を確かめた．加えて，null チェックがコード中に占める割合は，実験対象の全プロジェクトで 100 行に 1 つから 4 つ程度であることがわかった．

これらの分析により，様々なプロジェクトにおいて，null 返り値および null チェックが保守作業に悪影響を与えていることがわかった．そのため，開発者は null 返り値を安易に記述してしまう前に，他の記述で置き換える事ができないかを考えるべき，といえる．具体的には，例外処理を用いる方法，null の代わりに適切なオブジェクト（空の配列，特殊なクラスなど）を返す，などが挙げられる．ただし，null 返り値および null チェックに対する置き換えは自動化されておらず，開発者の負担は増大してしまう．そのため，今後の研究として，null 返り値の置き換えを支援する手法を研究する予定である．また，null 返り値および null チェックがどのような状況で用いられることが多いのか，また，null 返り値の置き換えにより，保守作業にどのような影響が生じるのかについて調査を行う予定である．

謝辞　本研究は，MEXT/JSPS KAKENHI 25220003, 24650011，および 24680002 の助成を得た．

参考文献

[1] Hoare, T.: Historically Bad Ideas: "Null References: The Billion Dollar Mistake, *QCon* (2009).
[2] Nanda, M. G. and Sinha, S.: Accurate Interprocedural Null-Dereference Analysis for Java, *ICSE*, pp. 16–24 (2009).
[3] Ayewah, N. and Pugh, W.: Null Dereference Analysis in Practice, *PASTE*, pp. 65–72 (2010).
[4] Bush, W. R., Pincus, J. D. and Sielaff, D. J.: A static analyzer for finding dynamic programming errors, *Software: Practice and Experience*, Vol. 30, No. 7, pp. 775–802 (2000).
[5] Hovemeyer, D., Spacco, J. and Pugh, W.: Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs, *PASTE*, pp. 13–19 (2005).
[6] Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N. and Nanda, M.: Verifying Dereference Safety via Expanding-Scope Analysis, *ISSTA*, pp. 213–224 (2008).
[7] Hovemeyer, D. and Pugh, W.: Finding More Null Pointer Bugs, But Not Too Many, *PASTE*, pp. 9–14 (2007).
[8] Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N. and Nanda, M.: Verifying Dereference Safety via Expanding-Scope Analysis, *ISSTA*, pp. 213–224 (2008).
[9] Hovemeyer, D. and Pugh, W.: Finding Bugs is Easy, *OOPSLA*, pp. 24–28 (2004).
[10] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B. and Stata, R.: Extended Static Checking for Java, *PLDI*, pp. 17–19 (2002).
[11] Nanda, M. G., Gupta, M., Sinha, S., Chandra, S., Schmidt, D. and Balachandran, P.: Making Defect-Finding Tools Work for You, *ICSE*, pp. 99–108 (2010).
[12] Hollander, M. and Wolfe, D. A.: *Nonparametric Statistical Methods, 2nd Edition*, Wiley-Interscience (1999).