Tracking Null Checks in Open-Source Java Systems

Haidar Osman
SCG, University of Bern
Switzerland
osman@inf.unibe.ch

Manuel Leuenberger
University of Bern
Switzerland
manuel.leu@students.unibe.ch

Mircea Lungu
University of Groningen
The Netherlands
m.f.lungu@rug.nl

Oscar Nierstrasz
SCG, University of Bern
Switzerland
oscar@inf.unibe.ch

***Abstract*— It is widely acknowledged that null values should be avoided if possible or carefully used when necessary in Java code. The careless use of null has negative effects on maintainability, code readability, and software performance. However, a study on understanding null usage is still missing.**

In this paper we analyze null checks in 810 open-source Java systems and manually inspect 100 code samples to understand when and why developers use null. We find that 35% of all conditional statements contain null checks. A deeper investigation reveals many questionable practices with respect to using null. Uninitialized member variables, returning null in methods, and passing null as a method parameter are among the most recurrent reasons for introducing null checks. Developers often return null in methods to signal errors instead of throwing a proper exception. As a result, 71% of the values checked for null are returned from method calls.

Our study provides a novel evidence of an overuse of null checks and of the null value itself in Java, and at the same time, reveals actionable recommendations to reduce this null usage.

***Keywords*-Null Checks; Null Usage; Static Analysis**

I. Introduction

Tony Hoare[1] considers null as his *"billion-dollar mistake"* [1]. Besides the bugs it introduces in running systems, null usage hinders performance [2], increases maintenance costs [3], and decreases code readability.

Recent studies show that a considerable number of bug fixes are recurrent [4][5][6][7]. Interestingly, the *if*-related bug category is dominant [4], and more particularly, *missing null check* is the most frequent pattern of bugs in Java systems [7]. These results suggest that null dereferencing is a major source of bugs in Java programs, forcing developers to add guards (null checks) on objects before using them.

Many tools and techniques have been introduced to solve the null dereferencing problem. However, a study is still missing on how null is used, why null checks are introduced, and where the checked-for-null objects come from in the source code.

In this paper, we aim at understanding when, why, and how often developers introduce null checks. More concretely, we pose the following three research questions:

RQ1: *How common are null checks?*
We answer this question by measuring the ratio between the overall number of conditional statements and those containing null checks.

RQ2: *What kind of objects are checked against null?*
We consider whether the checked object is a parameter, a member variable, or a local variable.

RQ3: *How are the checked-for-null objects initialized?*
We analyze the kind of expression that was assigned to the checked-for-null object.

RQ4: *How is null used in Java programs?*
We manually investigate 100 samples from our corpus to understand what null represents in Java programs (*e.g.*, errors, default values, or other special values).

To answer the posed research questions, we developed a tool, *NullTracker*, that statically analyzes Java source code files and approximates the statistics about null check usage. Applying our analysis to a large Java software corpus, we find that 35% of the conditional statements in our corpus are null checks.

We also find that 24% of the checked objects are member variables, 23% are parameters, and 50% are local variables. Unsurprisingly, 71% of the checked objects come from method calls. In other words, developers insert null checks mainly when they use methods that may return null. Passing null values to methods and uninitialized member variables are recurrent reasons for introducing null checks.

We manually review 100 code samples from our corpus to understand the contexts and discover patterns of null check usage. In 76 samples null is used to represent errors, in 15 samples it is a representation of absence of a value, and in 9 samples null represents a meaningful value. Most of these instances of null usage can (or should) be replaced by proper exceptions or *special case objects* such as a *Null Object* [8][9].

The rest of the paper is organized as follows: in Section II, we explain why using null leads to problems. We explain the procedure we followed to extract the null checks and the devised heuristics to analyze the checked-for-null objects in Section III. Then, we explain the experimental setup,

IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016, Osaka, Japan.

[1]A computing pioneer credited with introducing the concept of a null pointer in Algol W. https://en.wikipedia.org/wiki/Tony_Hoare

the terminology, and the procedure we followed to analyze null checks and null usage in Section III. In Section IV, we demonstrate the results and answer the posed research questions. In Section V, we discuss the research questions and the implications of the results. We then explain in Section VI the possible threats to validity in our study and how we tried to mitigate them. In Section VII, we discuss the related work and how the null-related problems are approached. Finally, we conclude this paper in Section VIII.

II. Motivation

Pominville *et al.* achieved 2% to 10% performance gain in Java bytecode when they annotated Java class files with assumptions about the "nullness" and array bounds [2]. With respect to null, their framework, SOOT [10], performs intra-procedural null analysis to make assumptions about variables being null or not to be able to remove unnecessary null checks in the bytecode level. This means that null checks impose a non-negligible performance overhead on Java programs.

In a managed language like Java, null is an unusable value. Its interpretation depends on the context and when it causes a problem, its value yields "no useful debugging information" [11]. For instance, the `listFiles()` method in the `File` class returns an array of the files, `File[]`, in the specified directory. However, if the directory does not exist, it returns null. This returned null value might mean that the `File` object does not exist, that it is not a directory, or that an I/O error has occurred. This inherent ambiguity of null leads to increased difficulties in code comprehension.

Missing null checks are the most recurrent bug pattern in Java systems [7]. This bug manifests itself as the Java NullPointerException. Debugging this kind of exception is particularly hard because the stack traces do not provide enough information about the source of the null. Acknowledging this problem, Bond *et al.* introduced *Origin Tracking* [11], which records code locations where unused values (such as null) are assigned. *Origin Tracking* gathers the necessary information dynamically at run time so they can be reported at failure time to aid the debugging activities. This indirectly means that the overuse of null in program increases maintenance efforts.

To this end, we establish that the use of null in Java code often leads to performance degradation, code that is harder to read, more defective software, and increased project maintenance efforts. In the following sections we explore how often null is used in Java code and in what contexts. This knowledge can help software engineers to build better static code checkers and develop better practices for writing and reviewing Java code.

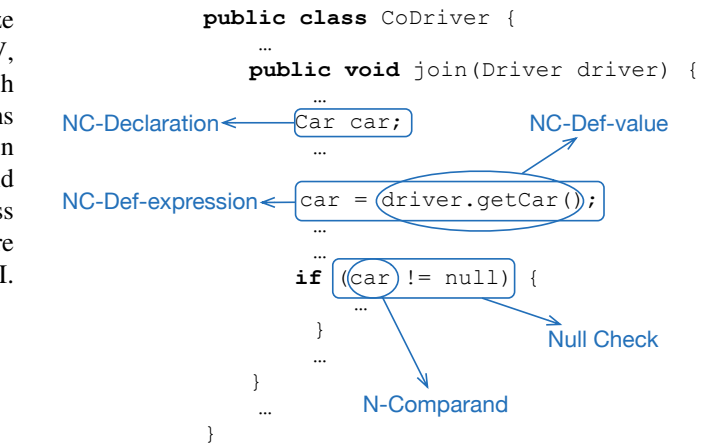


Figure 1. A code example showing the definitions of the terms used in the paper.

III. Null Check Analysis

A. Experimental Corpus

For our experiment, we used the same software corpus from a previous study [7]. This corpus was built using a crawler that queries Github[2] for Java projects that have more than 5 stars (popular) and are more than 100KB in size (relatively large). This corpus contains 810 Java projects, 371,745 Java source files, and 34,894,844 lines of code. We are making the corpus available for download through the Pangea infrastructure[3] [12].

B. Terminology

Before we explain the analysis, we define the terms used in this paper as follows (depicted in Figure 1):

- *A Conditional*: is a binary comparison expression that evaluates to a boolean value, such as:
  - `y > 0`
  - `x != null`
- *A Conditional Statement*: is a Java statement that contains a conditional, such as:
  - `if(y > 0) ...`
  - `isNull = (x != null);`
- *A Null Check*: is a conditional that contains the *null* literal as a left hand side or a right hand side operand. In other words, It is a comparison to null. For instance:
  - `Person != null`
  - `iterator.next() != null`
- *An N-Comparand*: is the expression that is compared to null in the null check (Usually an assignable l-value.)

[2]http://www.github.com/
[3]http://scg.unibe.ch/research/pangea

- *An NC-Declaration*: is the declaration expression of the *N-Comparand*.
- *An NC-Def-expression*: is the assignment expression involving the *N-Comparand* as the assignable l-value.
- *An NC-Def-value*: is the value assigned to the comparand in an *NC-Def-expression* (*i.e.*, the right-hand side operand of the *NC-Def-expression*).

C. Analysis

We implemented a tool, *NullTracker*[4], to extract null checks and analyze the kinds and definitions of the *N-Comparand*s. *NullTracker* is designed as a pipeline, following a pipes and filters architecture. *NullTracker* analyzes each Java source file as follows:

1) Parse the Java source file and extract the null checks.
2) For each null check, extract the *N-Comparand*.
3) Parse the *N-Comparand* and determine its kind (*e.g.*, name, method call, field access, *etc.*).
4) When the *N-Comparand* is a name expression, determine its kind (member variable, local variable, or parameter) by looking for its *NC-Declaration* within the current method for local variables and parameters, and within the current type declaration for member variables.
5) When the *N-Comparand* is assignable (name, array access, field access), extract all the *NC-Def-expression*s that appear lexically (textually) before the null check and within the same method as the null check itself. Then, parse them and extract the kind of the *NC-Def-value*s (method call, null literal, object creation, or any expression that can evaluate to a reference value).
6) Finally, the resulting data, which conforms to the model illustrated in Figure 2, is saved in the database for further analysis.

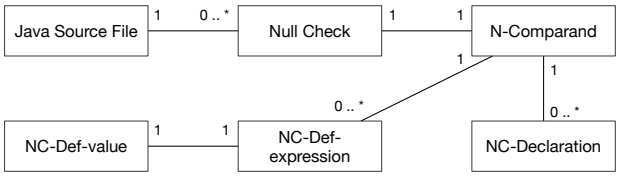


Figure 2. The data model of *NullTracker* analysis.

D. Manual Inspection

After the analysis phase, we manually inspect multiple instances of the null checks belonging to different categories and types. More concretely, we inspect 10 random samples of each of the following categories to gain more insight:

1) Method call *N-Comparand*s.
2) Field access *N-Comparand*s.
3) local variable name *N-Comparand*s.
4) Parameter name *N-Comparand*s.
5) Member variable name *N-Comparand*s.
6) Method invocation *NC-Def-value*.
7) Null literal *NC-Def-value*.
8) Cast *NC-Def-value*.
9) Object creation *NC-Def-value*.
10) Name *NC-Def-value*.

In this phase we aim at understanding how and why developers use null values and null checks. We look specifically at the following:

1) The intended semantics of the null value.
2) Potentially missing null checks (*e.g.*, a member variable that is sometimes checked against null and sometimes not before dereferencing it).
3) The type of the checked-for-null object (String, List, Tree, Number, *etc.*).
4) The source of the null value. (*e.g.*, uninitialized local variables, a return null statement in a method body, *etc.*)

We do not derive any statistics from this phase, as we only want to gain deeper insights into how null and null checks are used in the code and for what reasons.

IV. Results

We applied our analysis to the 810 Java projects in our dataset and we manually reviewed 100 code samples. In the following subsections which are organized around the research questions, we explain the results.

A. How Common Are Null Checks?

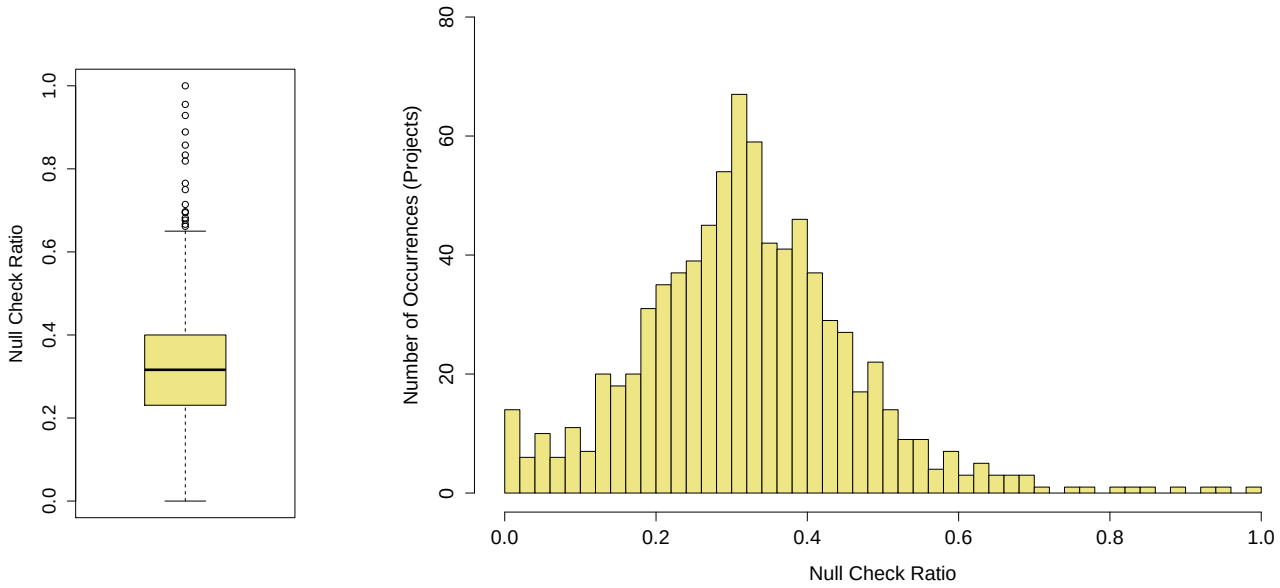
To our knowledge, only Kimura *et al.* have measured the density of null checks in source code [3]. They measured the ratio between the number of null checks and the number of lines of code. They found this ratio to be from one to four per 100 lines of code, depending on the project.

We, on the other hand, go one step further and measure the ratio of the conditional statements containing null checks with respect to all conditional statements. This will enable us in answering the first research question: **RQ1: How Common are Null Checks?**

We call the ratio between the null checks and the overall number of conditional statements the *null check ratio*. Analyzing our dataset, we found 2,329,808 conditionals, 818,335 of which contain null checks. This means that a staggering 35% of the conditional statements contain null checks.

As detailed in Table I and Figure 3, the null check ratio exhibits a bell-shaped distribution around the value of 32%. In other words, more than half of the projects have a null check ratio of more than 32%. Our results show evidence of an existing overuse (or abuse) of null checks by Java programmers, which indicates the over-use of the "null" value itself. As discussed in Section II, this practice affects the readability of code, the maintainability of the project, and the performance of the running system.

[4]https://github.com/haidaros/NullTracker



(a) (b)

Figure 3. The distribution of the per-project null check ratio. The boxplot in (a) shows that more than half the projects have null check ratios of more than 32%. The histogram in (b) shows that the null check ratio distribution demonstrates a bell-shaped curve around the value of 32%.

Table I
A SUMMARY OF THE PER-PROJECT NULL CHECK RATIO.

	Number of Conditionals	Number of Null Checks	Null Check Ratio
Min	1.0	0.0	0%
1st quartile	117.5	33.0	23%
Median	531.5	138.5	32%
Mean	2,876.3	1,010.3	32%
3rd quartile	2,171.8	671.5	40%
Max	196,812.0	76,609.0	100%

B. What Entities Do Developers Check For Null?

To answer the second research question (**RQ2: What kind of objects are checked against null?**), first, we analyze the kind of the *N-Comparand* itself. Second, we look for the *NC-Declaration* of the *N-Comparand* when it is an l-value (*i.e.*, *name expression, array, or field access expression*).

We find that *N-Comparand*s are mainly *name expressions* (78%) and *method call expressions* (15%), as Figure 4 shows. Figure 5 shows that 50% of the name *N-Comparand*s are local variables, 24% are member variables, and 23% are parameters.

Inspecting 10 code samples where the null check is against a method call, the method calls are all for getter methods either from the same class or from another class. This puts field access *N-Comparand*s, method call *N-Comparand*s, and member variable name *N-Comparand*s in the same category, which is member variable null check. Member variables are checked against null because they might not be initialized. This happens in the manually inspected code when one or more of the following is true:

- There exists a constructor that does not initialize it.
- There exists a constructor or a setter method that can accepts null as a parameter and sets it to null.
- There exists a method that explicitly sets to null.
- The member variable is public or is returned by address in the getter method.

The code in all inspected 10 samples where the *N-Comparand* is a member variable can be improved to avoid having to check for null every time the member variable needs to be used. In our samples, there is no obvious reason why member variables are not explicitly initialized in every constructor. In fact in 5 of the inspected samples, the member variable should even be `final`. This suggests a failure in applying well-established object-oriented design principles, in particular that of establishing class invariants [13].

As we see in Figure 5, a considerable percentage of null checks are guards on method parameters. In the inspected code samples, we observe that developers check parameters

against null mainly to validate them. However, we differentiate between two recurring patterns of null checks on parameters. In the first pattern, a method throws an exception if a certain parameter is null. Listing 1 shows a real example of this pattern. The second, and more questionable, pattern is shown in Listing 2. The method skeletons in Listing 2 are the most recurrent usage scenarios of a parameter null check. In these scenarios, the method does not accept null as a parameter. However, instead of throwing a proper exception, the method does nothing and returns silently without informing the caller of any problem.

```
public File writeToFile(final HttpEntity entity
    ) throws ClientProtocolException,
    IOException {
        if (entity == null) {
          throw new LibRuntimeException(
    LibResultCode.E_PARAM_ERROR);
        }
  .
  .
  .
}
```

Listing 1. Throwing a proper exception when the parameter is null.

```
SOME_TYPE method1(Param p){
  if(p==null){
    return null;
  }
  METHOD_BODY
}

void method2(Param p){
  if(p==null){
    return;
  }
  METHOD_BODY
}


void method3(Param p){
  if(p!=null){
    METHOD_BODY
  }
}
```

Listing 2. The most recurrent usage scenarios of a parameter null check.

More often than not, this indicates a poor API design. One can use, for instance, the *specification pattern* [14][15] to extract the validation code, which throws a proper exception, in a dedicated method or class. In any case, it is widely acknowledged that passing null as an argument to methods is a bad practice and *"the rational approach is to forbid passing null by default"* [16]. Nevertheless, developers often add null checks on parameters because they expect null to be passed as a parameter. Our results show a clear gap between what is considered a good practice and how software is implemented in reality.

C. Where Does The Null Come From?

When the *N-Comparand* is a name expression, we analyze the *NC-Def-value*s assigned to it in all the *NC-Def-expression*s preceding the null check within the same

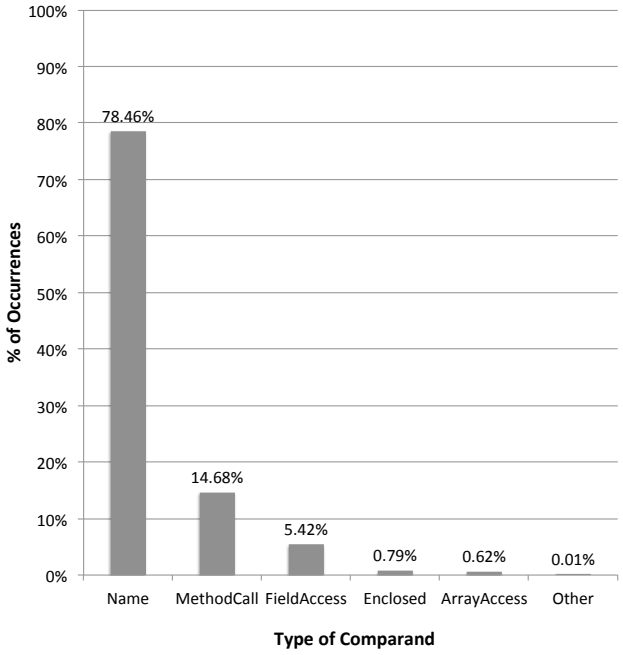


Figure 4. The immediate kind of the *N-Comparand*s as a parsing expression.

method. Figure 6 shows the kinds of the assigned *NC-Def-value*s. As an answer to the third research question (**RQ3: How are the checked-for-null objects initialized?**) we find that 71% of the time the *NC-Def-value* is a method call expression, which means that null checks are mostly applied to values returned from method invocations. In other words, when methods possibly return null, they tend to cause NullPointerExceptions in the invoking methods forcing developers to add null checks.

There is a long debate about whether methods should return null or not. In a previous study [7], we found that missing null checks represent the most frequent bug in Java programs. In this study, we show that null checks are applied to the results of method invocations. Both studies combined provide evidence that returning null in methods is a major cause of bugs. Hence, we side with the opinion that developers should avoid returning null in their method implementations and either throw an exception or return a *special case object* [16] such as a *Null Object* [8][9].

Surprisingly, some Java standard libraries exhibit this questionable design [16]. In our manually inspected code samples, we find 5 null checks because of methods from the Java standard API *e.g.*, Map.get(...), List.get(...), Iterator.next().

Another less frequent reason for checking a local variable for null is when it is initialized within a method call or a constructor that might throw an exception before completing. We observe this pattern in our manually-inspected code samples, as the code skeleton shows in Listing 3. The

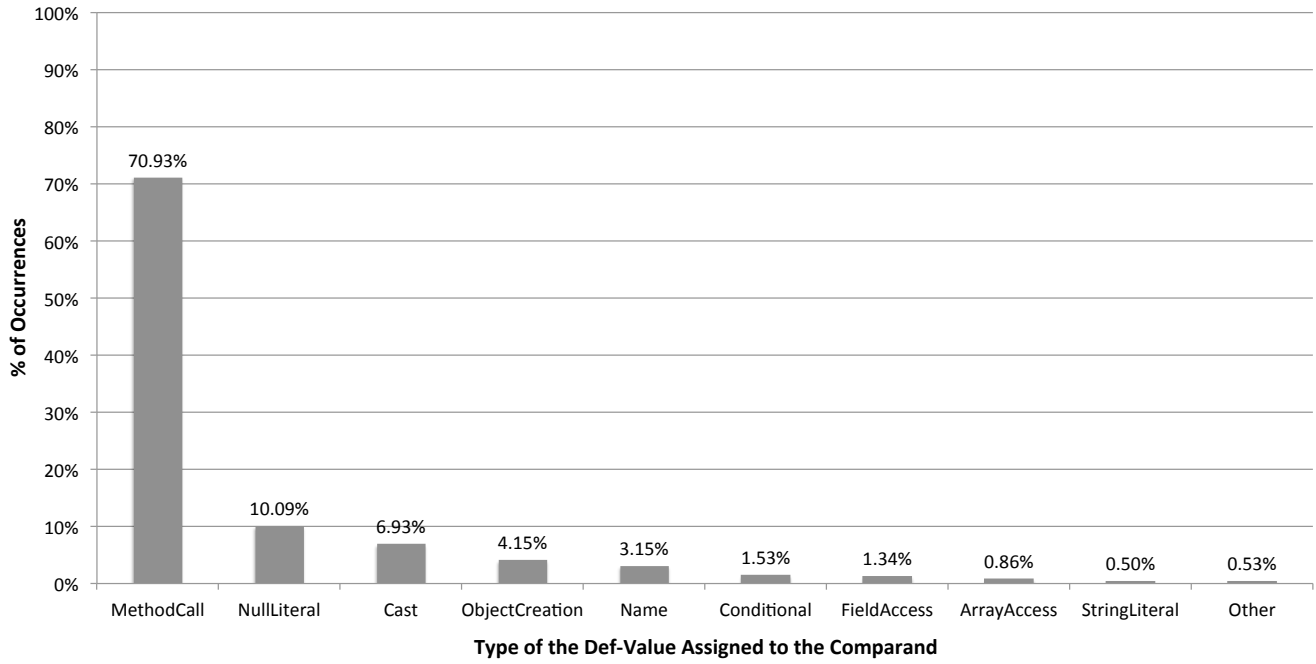


Figure 6. This diagram shows that the checked-for-null objects are mainly set or initialized using method invocations.

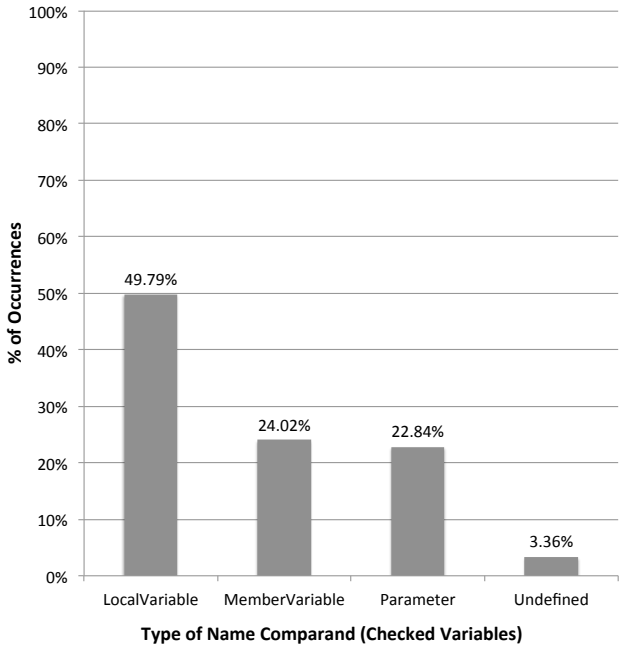


Figure 5. The type of name expression comparands. *Undefined* indicates that 3.36% cannot be determined, as explained in Section VI

variable is set to null first, initialized in a `try-catch` block, then checked for null to make sure that initialization completed and no exception was thrown.

...

```
Object obj = null;
try{
  obj = METHOD_INVOCATION or CONSTRUCTOR_CALL
}catch(...){
  ....
}
if(obj != null){
  ...
}
...
```

Listing 3. When the initialization is in a try-catch block, a null check usually follows to make sure no exception is thrown.

D. What Does Null Mean?

In the manual inspection phase we find that developers use null values for three different reasons.

As an answer to the fourth research question (**RQ4: How is null used in Java programs?**), the most recurrent usage of null is to encode or represent an error. 76 of 100 inspected samples fall into this category. For instance, if a method does not accept null as a parameter, it returns null or just returns when it encounters a null parameter. Another example is when a method returns null in a `catch` block.

The second usage of null is to represent the absence of a value (or the *nothingness*). 15 out of the 100 inspected samples fall into this category. For example, left and right branches in a leaf node are null in a binary tree. Another example is a `find(...)` method that returns null in case it cannot find the item.

The third usage of null is when it is a proxy for a meaningful value. For instance, in many code samples, we

find non-boolean variables used as ones (null means `false` and instantiated means `true`).

We argue that the first and third usage of null are bad practices as there are other language constructs to represent the corresponding semantics. For errors and failures, one should use well-defined exceptions as they are easier to read and act upon. For the third usage, one can have a dedicated `enum`, `class`, or other data type to represent the semantics in a more readable and maintainable form. In fact, 19 out of the 100 inspected classes contain potential bugs in the form of potentially missing null checks. All of these 19 potential bugs exist when null represents an error or a meaningful value.

We also argue that second usage scenario where null means the absence of a value is only tolerable and not ideal, as one should use *special case object* [16] like the *Null Object* [8][9] when applicable. Actually, even when the representation of absence of value is available for free (such as an empty string for a `String` variable), developers tend to not use it. Eight out of the inspected *N-Comparand*s are of type `String` and seven are of type `List`. Carefully reading the code, we find out that uninitialized strings and lists are equivalent to empty strings and lists correspondingly in these samples.

V. Discussion

Null usage often leads to various maintenance problems. Missing null check bugs, useless NullPointerException stack traces, and vague null semantics are often the consequences of the careless use of null value. A disciplined usage of null is highly recommended [17], and alternatives, like the Null Object Pattern [8][9], should be considered. However, the results of this study suggest that null is often misused and overused in Java code.

To demonstrate how null is often misused, we put the observations from this study and from our previous study on bug-fix patterns [7] into a hypothetical story that represents the most recurrent pattern of wrong null usage.

A developer, *A*, adds a `return null` statement in the method body of `m(...)` to indicate that a problem has occurred and the method `m(...)` cannot continue its normal execution. Another developer, *B*, uses the method `m(...)` and assigns its result to a local variable `obj`. When developer *B* runs his code, a NullPointerException is thrown. Developer *B* looks at the stack traces struggling to understand the problem and finally identifies the place of the null dereferencing. It is the local variable `obj` and the null comes from the method invocation of `m(...)`. Unable to understand the meaning of null in this situation, developer *B* adds a null check before using the local variable *obj*.

When such scenarios accumulate in a codebase, the code starts to get harder to maintain and reason about, leading to the problems discussed in Section II.

However, our study also reveals some actionable recommendations to reduce null checks and null usage:

1) A method should not return null in case of errors. A method should always throw a proper exception that explains the exact reason and even possible solutions in case of errors.
2) Null should not be passed to public methods and public methods should not accept null as a parameter. In other words, public method arguments should be non-null by default.
3) Member variables should be initialized either in all constructors or through the use of the *Builder* pattern. The point here is that objects should be fully constructed before being created and class invariants should be explicitly established.
4) String instances should be initialized to empty strings `""`.
5) List instances should be initialized to empty lists.

Following the aforementioned practices can prevent or at least mitigate the problems coming from null usage (as discussed in Section II). These practices can be ensured manually during code review or automatically using static code analyzers and annotations. An even more radical approach is to forbid the usage of null altogether in the language and observe the effects on the code quality, but this is a topic for further study.

VI. Threats to Validity

The internal threats to validity come from the known limitations of static analysis itself on one hand, and the limitations of our heuristics on the other hand.

- As can be seen in Figure 5, *NullTracker* cannot trace 3.36% of the variables back to their declarations, as can be seen in Figure 5, because we parse and analyze one Java source file at a time. For instance, inherited member variables cannot be discovered.
- *NullTracker* extracts all the *NC-Def-expression*s that appear lexically before the null check regardless of the actual data flow taking all the possible *NC-Def-expression*s into account.
- *NullTracker* cannot detect whether an *N-Comparand* is changed by passing it as a parameter to other methods. Only *NC-Def-expression*s are considered.
- When an *N-Comparand* appears within the same method in multiple null checks, every time it is considered a different *N-Comparand* leading to possible duplicates in the analysis of the *N-Comparand* kinds.

The external threats to validity come from the fact that we only analyzed 810 Java open-source projects. The results might not generalize to all open source projects or to industrial closed-source projects

VII. Related Work

A closely related work is a study on the maintainability burden imposed by the "return null" statement [3]. Kimura *et al.* found that return null statements are modified more frequently than other return statements but null conditionals are not. This indicates that the presence of return null is costly to maintain. They also found that the density of null checks are from one to four per 100 lines of code [3]. Our definition of null check ratio expresses null check usage better than null checks per LOC because it quantifies the added complexity null checks have on the code.

Null-related bugs attract the attention of many researchers and practitioners who propose various approaches to detect them as early as possible.

The first family of solutions incorporates data flow analysis techniques to detect possible null values. Some techniques are simple, fast, and intra-procedural [18][19][20][21][22] and some are more complex, thorough, and inter-procedural [23][24][25][26]. For instance, Hovemeyer and Pugh [22] perform intra-procedural forward data flow analysis to approximate the static single assignment for the values of variables. Then they analyze the dereferences as a backward data flow over the SSA approximation. This algorithm replaces the previous basic forward data flow analysis approach [27] and is now part of FindBugs [28], a static analysis tool for finding bugs in Java.

An interesting study by Ayewah and Pugh [29] compares several null dereference analysis tools and observes that, besides the reported false positives, many of the reported null dereferences (true positives) do not manifest themselves as bugs at run time. The authors claim that when the null dereference passes the initial software testing, it rarely causes bugs and "reviewing all potential null dereferences is often not as important as many other undone software quality tasks"[29]. We argue that Ayewah and Pugh underestimate the frequency of the bugs caused by dereferencing null as we show in our previous study [7].

The second family of solutions proposes to annotate the "nullness" in code. Fähndrich and Leino propose to distinguish the non-null references from the possibly-null ones at the type level (using annotations) to detect null-related bugs [30]. Papi and Ernst introduced the @NonNull annotation on types [31]. Loginov *et al.* [23], beside their inter-procedural null dereference analysis, propose null-related annotations to ensure the soundness and safety of the analysis. The idea of annotations made it to widely-used Java libraries like *Checker Framework*[5] and *Guava*[6]. Also some programming languages introduce the idea of reference declarations that are not null. For instance the Spec# programming system extends the C# programming language with the support of contracts (like non-null types), allowing the Spec# compiler to statically enforce these contracts [32].

The third family of solutions tries to solve the problem by introducing language constructs. Haskell [33] and Scala [34] have the "Maybe" and the "Option" types, which are object containers. In a similar fashion, Oracle introduced the "Optional" type in Java 8 recently [35]. Groovy and C# have the safe navigation "?." to safely invoke a method on a possibly-null object.

None of the above solutions deals with null usage problem thoroughly. The first family of solutions does not reduce null usage but points out potential null dereferencing locations in the code, encouraging developers to add even more null checks. The second family of solutions can mainly ensure that method parameters are not null, but cannot, for instance, prevent methods from returning null. The third family of solutions just encapsulates the problem with syntactic sugar rather than solving it. We argue that more research is needed to solve the problems associated with null usage by dealing with the cause of the problems instead of dealing with the symptoms. A good solution should foster good programming practices and a disciplined usage of null.

VIII. Conclusions

Null-related problems are very common in Java. The overuse of the null value may introduce more bugs, hinder performance, and lead to maintenance difficulties.

In this paper, we study null checks as a proxy to understand null usage. We conduct a census of the null checks in Java systems showing that 35% of the conditionals are null checks. Our analysis reveals many bad practices in terms of null usage. Returning null in methods, passing null as arguments, and uninitialized member variables are the most frequent, and questionable, null usage patterns causing the high null check density. Finding out the root causes behind improper null usage, we provide actionable recommendations to avoid null-related problems. These recommendations can be checked using static code analyzers to ensure a disciplined use of null and increase the quality of the code.

Acknowledgements

The authors gratefully acknowledge the funding of the Swiss National Science Foundations for the project Agile Software Assessment (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015)[7].

The authors also acknowledge the financial support of the Swiss Object-Oriented Systems and Environments (CHOOSE)[8] for the presentation of this research.

[5]http://types.cs.washington.edu/checker-framework/

[6]https://github.com/google/guava

[7]http://p3.snf.ch/Project-144126

[8]http://www.choose.s-i.ch

REFERENCES

[1] T. Hoare, "Null references: The billion dollar mistake," *Presentation at QCon London*, 2009.

[2] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing Java using attributes," in *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2000, p. 8.

[3] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Does return null matter?" in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb. 2014, pp. 244–253.

[4] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[5] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806847

[6] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with AST analysis," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, eRA Track.

[7] H. Osman, M. Lungu, and O. Nierstrasz, "Mining frequent bug-fix code changes," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, Feb. 2014, pp. 343–347. [Online]. Available: http://scg.unibe.ch/archive/papers/Osma14aMiningBugFixChanges.pdf

[8] B. Woolf, "The null object pattern," in *Design Patterns, PLoP 1996*. Robert Allerton Park and Conference Center, University of Illinois at Urbana-Champaign, Monticello, Illinois, 1996.

[9] ——, "Null object," in *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds. Addison Wesley, 1998, pp. 5–18.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot — a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[11] 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," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07)*. New York, NY, USA: ACM, 2007, pp. 405–422.

[12] A. Caracciolo, A. Chiş, B. Spasojević, and M. Lungu, "Pangea: A workbench for statically analyzing multi-language software corpora," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, Sep. 2014, pp. 71–76. [Online]. Available: http://scg.unibe.ch/archive/papers/Cara14c.pdf

[13] B. Meyer, "Applying design by contract," *IEEE Computer (Special Issue on Inheritance & Classification)*, vol. 25, no. 10, pp. 40–52, Oct. 1992. [Online]. Available: http://se.ethz.ch/~meyer/publications/computer/contract.pdf

[14] E. Evans and M. Fowler, "Specifications," in *Proceedings of the 1997 Conference on Pattern Languages of Programming*, 1997, pp. 97–34.

[15] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[16] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[18] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Softw. Pract. Exper.*, vol. 30, no. 7, pp. 775–802, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(200006)30:7⟨775::AID-SPE309⟩3.0.CO;2-H

[19] I. Dillig, T. Dillig, and A. Aiken, "Static error detection using semantic inconsistency inference," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 435–445. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250784

[20] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 57–72. [Online]. Available: http://doi.acm.org/10.1145/502034.502041

[21] 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*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/512529.512558

[22] 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*, ser. PASTE '07. New York, NY, USA: ACM, 2007, pp. 9–14. [Online]. Available: http://doi.acm.org/10.1145/1251535.1251537

[23] 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*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 213–224. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390657

[24] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 97–107. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273478

[25] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for Java," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133–143. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070515

[26] T. Ekman and G. Hedin, "Pluggable checking and inferencing of non-null types for Java," *Journal of Object Technology*, vol. 6, no. 9, pp. 455–475, 2007.

[27] 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*, ser. PASTE '05. New York, NY, USA: ACM, 2005, pp. 13–19. [Online]. Available: http://doi.acm.org/10.1145/1108792.1108798

[28] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[29] 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*, ser. PASTE '10. New York, NY, USA: ACM, 2010, pp. 65–72. [Online]. Available: http://doi.acm.org/10.1145/1806672.1806686

[30] M. Fähndrich and R. Leino, "Declaring and checking non-null types in an object-oriented language," in *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, 2003. [Online]. Available: http://research.microsoft.com/~maf/Papers/non-null.pdf

[31] M. M. Papi and M. D. Ernst, "Compile-time type-checking for custom type qualifiers in Java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 809–810. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297911

[32] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *Construction and analysis of safe, secure, and interoperable smart devices*. Springer, 2005, pp. 49–69.

[33] "Haskel 98 Report." [Online]. Available: https://www.haskell.org/onlinereport/index98.html

[34] M. Odersky, "Scala language specification v. 2.4," École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, Tech. Rep., Mar. 2007.

[35] R.-G. Urma, "Tired of null pointer exceptions? Consider using Java SE 8's optional!" Oracle, Tech. Rep., Mar. 2014. [Online]. Available: http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html