Null Check Analysis

Haidar Osman

Software Composition Group
University of Bern
osman@iam.unibe.ch

Abstract

Null dereferencing is one of the most frequent bugs in Java systems causing programs to crash due to the uncaught NullPointerException. Developers often fix this bug by introducing a guard (*i.e.*, null check) on the potentially-null objects before using them.

In this paper we investigate the null checks in 717 open-source Java systems to understand when and why developers introduce null checks. We find that 35% of the *if*-statements are null checks. A deeper investigation shows that 71% of the checked-for-null objects are returned from method calls. This indicates that null checks have a serious impact on performance and that developers introduce null checks when they use methods that return null.

1 Introduction

In a previous work [1], we analyzed the bug-fix code changes in a software corpora that consists of 717 popular (> 5 stars) and large (> 100 KB) Java projects cloned from Github [1]. We found that a considerable number of the bug fixes are recurrent, confirming literature findings [2][3][4]. However, an interesting finding was that missing null check is the most recurrent bug arising across the projects in both corpora.

In this paper, we analyze the null checks in Java systems to gain a better understanding of the missing null check bug pattern. Surprisingly, our study shows that 35% of the conditional statements in our corpus are null checks. This result suggests that the null checks have a serious negative impact on the performance of Java systems. Also we find that 25% of the checked objects are member variables, 24% are parameters, and 51% are local variables. Unsurprisingly, 71% of the checked objects come from method calls. In other words, developers mostly insert null checks when they use methods that return null.

2 Extracting Syntactic Bug-Fix Patterns

We analyzed the evolution history of each project in our corpora to extract bug-fix code changes. Our approach follows these steps:

1. Extract the bug fixing commits.
2. Extract method bodies before and after the commit.
3. Diff the text of every changed method.
4. Anonymise the changed code:

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2015 (sattose.org), Mons, Belgium, 06-08 July 2016, published at http://ceur-ws.org

(a) Every number is replaced by N.

(b) Words are anonymized using numbered letters (T0, T1, T2, *etc.*).

(c) Every whitespace is removed.

For example, following these steps, we can end up with a bug-fix code change such as:

```
((View) getParent(1)).invalidate()    --->    View parent = (View) getParent(1);
                                              if (parent != null) parent.invalidate();
```

Anonymizing the previous bug-fix code change results in the bug-fix syntactic pattern:

```
((T0)T1(N)).T2()    --->    T0T3=(T0)T1(N);
                            if(T3!=null)T3.T2();
```

After investigating the results, we found that "missing null check" is the most frequent bug category [1]. This bug is usually fixed by adding a null check like in the following patterns:

```
[]    --->    if(T0==null)return;
[]    --->    if(T0==null)return null;
[]    --->    if(T0!=null)T0.T1();
[]    --->    if(T0==null)throw new T1();
[]    --->    if(T0==null)return T0;
```

3 Null Check Analysis

The severity of the missing null check bug pattern comes from the fact that it causes software systems to crash due to the often uncaught NullPointerException in Java. A manual inspection of 200 instance of the missing null check pattern shows that 70% of the checked-for-null objects come from method calls. In other words, when a method "may" return null, the returned object should be checked before it is dereferenced.

In this study we aim to better understand the checked-for-null objects. More specifically, we want to understand the types of these objects and how they are usually initialized. So, we developed a tool that analyzes Java source code and extract information about the null checks and the checked-for-null objects. Our tool analyze each Java source file as follows:

1. It parses the Java source file and extracts the null check (conditional statements).

2. It extracts the expression that is checked against null.

3. It parses the expression and determines its type (name expression, method call expression, array access expression, *etc.*).

4. When the expression is a name expression, the tool determines its type (member variable, local variable, or parameter).

5. When the expression is a name expression, the tool also extract all the assignment statements that appear textually before the null check and parses them to extract the type of the assigned expression (method call, value, *etc.*)

4 Where Does This Null Come From?

We ran the null check analysis on our corpus. We found that 35% of the conditional statements are null checks. Similarly, but from another point of view, Kimura *et al.* found that the density of null checks are from one to four per 100 lines of code [5]. These results combined pose serious doubts regarding the effect of the null checks on performance, but this is outside the scope of this paper.

We also found that 25% of the checked-for-null objects are member variables, 24% are parameters, and 51% are local variables. The analysis of the assignments of these objects show that 71% of the time they are assigned the results of method invocations.

5 Implications

The results of this study imply that null checks are mostly applied on objects coming 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. This motivates the development of a tool that checks whether the returned object is checked before it is put in use, when the invoked method may return null.

Also the fact that parameters get checked against null often, indicates poor API design. We argue that null should not be passed as a parameter to methods. Method overloading should be used to prevent such scenarios.

References

[1] 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*, pp. 343–347, Feb. 2014.

[2] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, pp. 286–315, June 2009.

[3] 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*, ICSE '10, (New York, NY, USA), pp. 315–324, ACM, 2010.

[4] 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.

[5] 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*, pp. 244–253, Feb. 2014.