The False False Positives of Static Analysis

Yuriy Tymchuk

SCG @ Institute of Informatics - University of Bern, Switzerland

Abstract

Static analysis tools may produce false positive results, which negatively impact the overall usability of these tools. However, even a correct static analysis report is sometimes classified as a false positive if a developer does not understand it or does not agree with it. Lately developers' classification of false positives is treated on a par with the actual static analysis performance which may distort the knowledge about the real state of static analysis.

In this paper we discuss various use cases where a false positive report is not false and the issue is caused by another aspects of static analysis. We provide an in-depth explanation of the issue for each use case followed by recommendations on how to solve it, and thus exemplify the importance of careful false positive classification.

1 Introduction

Static code analysis can aid software developers to detect bugs by analyzing source code without executing it [1]. Static analysis tools have diverse detection algorithms also known as *rules*, that can detect certain known anomalies in a software project. Because of the lack of information these algorithms may produce incorrect detections (false positives). In case of a high false positive ratio, a developer who inspects static analysis results may spend significant amount of time trying to address the incorrect detections. Thus a high number of false positives can decrease the overall usefulness of a static analysis tool.

False positives are a critical issue for static analysis users. Bessey *et al.* suggest to keep the ratio of false positives under 20-30% to make a tool acceptable by users [2]. Google went even further and built their static analysis environment in a way that it automatically suppresses every rule that has more than 10% of false positive reports based on the user feedback [3]. However, the definition for false positives in static analysis quite often diverges from the common sense. According to the Merriam-Webster dictionary[1] a false positive is *"a result that shows something is present when it really is not"*. On the other hand, some users of static analysis classify reports as false positives if they do not understand the rationale behind the reports, or simply do not think that the reports are important for them [2]. No one can forbid people to express their opinion that true positives are false, and such cases should not be ignored. But we believe that the community around static analysis should not confuse the false positives identified by users with the real ones. First of all this will distort the false positive ratio and secondly this will mask real issues, such as poor understandability of a report. The reality is quite the opposite, for example Sadowski *et al.* state that for their study "developers will decide what a false positive is" [3]. In other words: when a person sick with tuberculosis says that his tuberculosis test is false positive, this neither cures the tuberculosis, nor makes the test incorrect. But if a software developer identifies a static analysis report as a false positive, then the anti-patterns in her code are not anti-patterns any more, and the false positive ratio of the detection rule suddenly increases.

We believe that the poor acceptance of static analysis reports is partially caused by the naming used in such tools. For example a developer starts to use a tool called FindBugs, and she expects that (as the name suggests) the tool will find bugs. Then the tool detects a bad practice, for example that a class has no comment. But the project still works despite the detection. The developer makes a conclusion that this is not a bug and thus the report has to be a false positive. To better understand the issue with the falsely identified false positives let us consider a different scenario. Originally static analysis applications came as full-fledged tools that have their own algorithms to check source code and their own user interface (UI) to provide results. Lately static analysis applications moved towards

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017 (sattose.org).
07-09 June 2017, Madrid, Spain.

[1] `https://www.merriam-webster.com/dictionary/false%20positive`

tools with a dedicated approach to provide reports. These tools reuse detection algorithms provided by other static analysis applications and aggregate the results in a single feedback. Some notable tools of this kind are Tricorder [3] that aggregates results of 11 analyzers into a pre-commit review report and UAV [4] that aggregates results of 3 analyzers into a tree map visualization. Our experience comes from QualityAssistant [5] — an extensible live feedback system that ideally should act as an artificial pair programmer. One of QualityAssistant's rules detects whether there is a temporary variable that is declared, but never read nor written. The rule can not produce false results, as all the information about temporary variables is available in method definitions. We believe that in a similar manner a pair programmer would point out the fact that a temporary variable is declared but never used. We are also aware about developers who do not like to see the information about unused variables. For example, a developer may want to suppress an information about unused variables on a piece of code under development, because he has not written statements that use newly defined variables yet. Although a developer can identify the reports described in this example as false positives, *the static analysis community has to focus on the problem with the tool output or its timing, and not with false positives*.

We developed QualityAssistant and the static analysis around it for 3 years in Pharo [6] — an object-oriented language inspired by Smalltalk. During that time we obtained multiple issue reports and plenty of mailing list discussions, and we conducted a series of interviews to learn how developers are using our static analysis support. In this paper we provide a few examples of the use cases in which a developer could identify an issue as a false positive but it is not so. We explain how falsely identified false positives mask real issues with tool design, communication between developers, and unclear design guidelines.

2 The True False Positives

First of all, we want to specify what we treat as the true false positives. As a false positive of a critique[2] we identify a rule violation report that does not conform to the rationale of the rule. For example there was a new rule introduced into Pharo which suggested to assert test results with `self assert: value equals: expected` instead of `self assert: (value = expected)`. The former expression provides a more descriptive output. However, the `assert:` method is defined on the top level of the class hierarchy and it is a common practice to make assertions in your code to express contracts (*i.e.,* preconditions, invariants and postconditions). For example one can have the following assertion in an algorithm:

```
self assert: (aCollection size = 1)
```

[2]a single violation of a certain static analysis rule

The rule also detected assertions outside of the test classes and such critiques were false positives as the `assert:equals:` method is defined only for tests and cannot be applied outside of the testing framework. This false positive is caused by a bug in the detection rule. The rule can easily check if the assertion is performed in a test class and eliminate previously discussed false positives.

Another more classic example of a true false positive is caused by a lack of information. One useful newly introduced rule detects issues related to the use of the Roassal [7] framework. When building a graph with this framework you are expected to specify nodes of the graph before defining edges. Thus the rule checks if an `edges` message[3] is preceded by a `nodes:` message. However Pharo is dynamically typed and the rule has no way to make sure that the receiver of the checked messages is a graph builder. This ambiguity will result in false positives as soon as there is another interface with `nodes:` and `edges` methods. This issue cannot be solved easily, but there are strategies that the rule may follow, to reduce the false positive ratio. For example the detection algorithm can use type inference [8] or dynamic analysis [9] to obtain the type of the messages receiver. Alternatively the rule can detect only the cases where a instance of the graph builder is created and immediately initialized. This will greatly lower the recall of this rule because it will miss all the cases where the graph builder is passed as an argument to a method or returned by another object. On the other hand, such change will increase the precision, as the rule will be sure about the type.

In both cases false positives were caused by the issues in the algorithm. They can be simple bugs or more complicated limitations of the environment where the algorithm is executed. In case of the true false positives it is the responsibility of the rule designer to act in that situation.

3 The False False Positives

While analyzing developers' preferences about quality rules, we discovered that there are rules that are not favored by some developers although they detect exactly what they intend to. Some of these rules are general best practices like a warning about a declared but unused variable, uncommented class, or a debugging statements left in source code. All these critiques are indeed bad practices which should not be present in a final version of an application and will not be integrated in the Pharo code base. However, some developers do not want to be bothered with such critiques while they develop, and would rather focus on them when they are about to commit the final version. This suggest that such rules should be applied in a pre-commit phase and not continuously while a developer is programming. This reasoning can be done only if we acknowledge

[3]the term "message" originates from Smalltalk, where one "sends a message" to an object, which then looks up a "method" for responding to it.

```
background ifNil: [ ^ true ].
(background isColor and: [ background isTranslucentButNotTransparent ]) ifTrue: [ ^ true ].
(border isColor and: [ border isTranslucentButNotTransparent ]) ifTrue: [ ^ true ].
^ false
```

Listing 1: The "quick return" approach

```
^ background isNil or: [
(background isColor and: [ background isTranslucentButNotTransparent ]) or: [
border isColor and: [ border isTranslucentButNotTransparent ] ] ]
```

Listing 2: The compound boolean logic

that there is an issue which is not a false positive, because it cannot be solved by updating the rule, but can only be solved by rethinking the static analysis tooling.

One more complicated rule was checking if there are multiple `if`-statements that were returning a boolean literal from a method and suggested to replace them with a compound conditional expression. For example in Listing 1 the conditional expressions on the first three lines check if some conditions are met and then return `true` from the method. In case the execution does not trigger the conditional expressions, the rest of the method is executed *i.e.,* `false` is returned.

The rule suggests to use the implementation demonstrated in Listing 2. This way all the conditions are incorporated into a single compound boolean expression. According to one of the developers this is a bad rule as it suggest a less readable code. A more detailed mailing list discussion revealed that most of the developers also find the implementation in Listing 1 easier to comprehend than the one in Listing 2. Moreover, no one from the Pharo community knows who implemented the rule, and many developers suggest to remove it completely. The rule cannot be improved in any way, as this is not a false positive, although it does not help developers. Blind removal of the rule based on the developer requests will eliminate its useless critiques, but will not answer the question of why the rule was created. We believe that in this case the community has to discuss the design guidelines and maybe replace the rule with an antipodal one that will detect constructs similar to Listing 2 and suggest to implement them as in Listing 1.

Another false false positive use case comes from an analysis of the QualityAssistant's impact. The integration of QualityAssistant into Pharo caused certain changes to the static analysis rules themselves [10]. Developers started to see critiques more often and this motivated them to fix incorrect rules or remove the ones that they found absolutely useless. When analyzing the rules that were removed from the Pharo ecosystem, we discovered a rule which was accused of having too many false positives. A more detailed investigation revealed that the critiques reported as false positives are not clearly false. In Smalltalk branching of a control flow is implemented in a functional style with the help of lexical closures. Listing 3 contains an example of a conditional expression. The expression `denominator = 0` will be evaluated to a boolean object, and depending on the object itself either the true block[4] or the false block will be evaluated.

```
(denominator = 0)
    ifTrue:  [ Float infinity ]
    ifFalse: [ numerator / denominator ]
```

Listing 3: Smalltalk conditional expression

```
size = 1 ifTrue: ':' ifFalse: 's:'
```

Listing 4: Conditional expression without blocks

The rule detected whether the conditional messages have a block as their argument. This rule is especially useful for novices, as they can forget to wrap their conditional expression in the square brackets, of confuse them with parentheses that create an ordinary expression instead of a block. In most of the cases the overall expression will still work, because any other object evaluated as a block will return itself. However this is not recommended, as the expressions will be instantly evaluated which will slow down execution, may change the state of the program or even result in an exceptional situation. For example if the snippet in Listing 3 did not have square brackets, both expressions `Float infinity` and `numerator / denominator` will evaluate on each execution, including the one where `denominator` is zero, which will cause a zero division exception. On the other hand, in certain cases developers prefer to omit blocks if they contain only a single literal as demonstrated in Listing 4. Further analysis showed that the reports about false positives came from experienced developers who are familiar with the implementation of the conditional expression and do not want to see warnings when they omit blocks. The precision of the rule could be improved to ignore the cases where literals are used as the arguments of conditional expressions. Nevertheless, we argue that the rule in its current state brings more value than the burden caused by the false positive critiques. First of all, novices can learn about the design of the conditional

[4]Block is the Smalltalk term for closure expression. Block definition is surrounded by square brackets.

expressions and fix their code as soon as they forget to wrap parameters of the conditional expression with blocks. On the other hand it is not hard for experienced developers to simply ignore the critiques if they omit blocks because such hacks are not common. Furthermore, originally in Smalltalk other objects were not polymorphic with the evaluation protocol of the block class, which means that a conditional expression without blocks will not run in all Smalltalk dialects.

Another especially irritating rule that developers did not like was detecting "cascading messages" that did not end with the `yourself` message. Cascades are a concept specific to Smalltalk and a real example of a message cascade from Pharo is presented in Listing 5. On the first line an instance of `ToolDockingBarMorph` is created. The rest of the lines separated by semicolons contain message sends to the same object (a newly created instance). This construct is very useful for initializing newly created objects with desired values and not having to retype the variable each time in front of a message. However, the result of the whole expression is equal to the value returned by the last method evaluated by the cascade (in our case `yourself`). This means that if `adoptMenuModel:` would be the last message and would return and adopted model the whole expression would return it, while the desired result is the instance of `ToolDockingBarMorph`. To avoid this kind of problem, one of the rules suggested to end cascades with the `yourself` message as shown in the example. This message simply returns the receiver *i.e.,* the the instance of `ToolDockingBarMorph` in our example. This rule is a good suggestion for novices who are not aware about the pitfalls of Smalltalk cascades, it can be absolutely annoying for experienced developers who want a different last message on purpose. While this rule is most often mentioned when developers list false positives or bad rules it cannot be clearly labeled with a negative tag. One of the interviewed developers admitted that maybe the rule is not that bad after all because when he rewrites his code to avoid such critiques, the code becomes more understandable.

```
1 ^ ToolDockingBarMorph new
2     hResizing: #shrinkWrap;
3     vResizing: #spaceFill;
4     adoptMenuModel: aModel;
5     yourself
```

Listing 5: Smalltalk cascade example

The use case with the "missing yourself" rule shows one more situation where a rule that was removed could help novices to learn how the programming language works. Additionally, there is a small evidence that the rule many suggest a better coding style. While we cannot claim the importance of the rule with respect to design guidelines, we can definitely conclude that instead of discussing the readability aspect that this rule promotes, the rule was simply deleted due to a false positives claim.

The final use case that we want to discuss is related to a warning against bad practices. Some developers do not like the rule that detects the usage of a reflective API, such as checking the type of an object. Similarly to the previous cases this rule may explain that there are other more appropriate ways to solve general problems without the support of reflectivity, but if the developer knows what she is doing, the rule is identified as distracting. On the other hand, senior developers think that the rule is always useful as it suggests not to use the reflective API during a programming session, and highlights questionable pieces of code during a code review. Once again this rule can be a candidate for removal due to a reasonable number of false positive claims from certain developers, but there are is also a evidence of the rule being useful for another group of developers. This means that either the rule should be applied on a personal basis or the should be a better communication to explain the importance of the rule.

The critiques mentioned in this section perform poorly to some extent. The main issues on these use cases are not caused by false positives but rather by vague design guidelines or poor tool design. In certain cases false positive are also present and the static analysis rules can be updated to eliminate some of the incorrect detections, or to provide a more detailed feedback. Nonetheless, to resolve the main issue a static analysis developer has to approach it from the non-false positive perspective.

4 Conclusion

False positive reports are one of the main issues of static analysis tools. However, sometimes even correct static analysis detections are classified as false positives. This not only skews the statistics of static analysis rules, but also masks the real problems. In this paper we provide examples where a static analysis report is not useful, but it is also not a false positive. Together with the examples we show the real issues that should be investigated. Static analysis community risks to miss such issues when labeling everything as a false positive when a developer does not like it.

The line between the false and the true false positives is very thin. For every true false positive one can add a "Possibly" prefix to the repot description and turn it into a false false positive. This way the critique stating "Possibly you should use `assert:equals:` instead of `assert:` and `=`" will never be false. But instead of playing with words we want to emphasize that there are issues with static analysis not related to false positives, and they have to be acknowledged separately. We suggest to identify false positives as the issues where critique cannot be easily identified because certain limitations. We also believe that there are cases where a rule suffers from both false positives and issues of a different kind.

At the moment there is much evidence that false positives have a negative impact on the acceptance of static

analysis tools. It is complicated to improve a false positive ratio, as usually it is caused by the lack of information in the source code. On the other hand we showed that some of the reported false positives are not really false detections, but rather issues of the understandability of quality rules, static analysis tool design, or inconsistency in guidelines. These issues can be easier to tackle and thus static analysis developers may improve the acceptance of their tools, by addressing the non-false positive issues first.

References

[1] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1109/MS.2006.114

[2] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374

[3] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 598–608. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818828

[4] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "UAV: Warnings from multiple automated static analysis tools at a glance," in *2017 IEEE 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2017, pp. 472–476.

[5] Y. Tymchuk, "What if clippy would criticize your code?" in *BENEVOL'15: Proceedings of the 14th edition of the Belgian-Netherlands software evoLution seminar*, Dec. 2015. [Online]. Available: http://yuriy.tymch.uk/papers/benevol15.pdf

[6] S. Ducasse, D. Zagidulin, N. Hess, and D. Chloupis, *Pharo by Example 5.0*. Square Bracket Associates, 2017. [Online]. Available: http://files.pharo.org/books/updated-pharo-by-example/

[7] A. Bergel, *Agile Visualization*. LULU Press, 2016. [Online]. Available: https://books.google.ch/books?id=lEk7vgAACAAJ

[8] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," in *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, vol. 26, Nov. 1991, pp. 146–161. [Online]. Available: http://www.cs.purdue.edu/homes/palsberg/publications.html

[9] T. Ball, "The concept of dynamic analysis," in *Proceedings of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC'99)*, ser. LNCS, no. 1687. Heidelberg: Springer Verlag, sep 1999, pp. 216–234.

[10] Y. Tymchuk, M. Ghafari, and O. Nierstrasz, "When QualityAssistant meets pharo: Enforced code critiques motivate more valuable rules," in *IWST '16: Proceedings of International Workshop on Smalltalk Technologies*, 2016, pp. 5:1–5:6. [Online]. Available: http://scg.unibe.ch/archive/papers/Tymc16b.pdf