JIT Feedback — what Experienced Developers like about Static Analysis

Yuriy Tymchuk
Swisscom, Switzerland
http://yuriy.tymch.uk

Mohammad Ghafari
University of Bern, Switzerland
ghafari@inf.unibe.ch

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

ABSTRACT

Although software developers are usually reluctant to use static analysis to detect issues in their source code, our automatic just-in-time static analysis assistant was integrated into an Integrated Development Environment, and was evaluated positively by its users. We conducted interviews to understand the impact of the tool on experienced developers, and how it performs in comparison with other static analyzers.

We learned that the availability of our tool as a default IDE feature and its automatic execution are the main reasons for its adoption. Moreover, the fact that immediate feedback is provided directly in the related development context is essential to keeping developers satisfied, although in certain cases feedback delivered later was deemed more useful. We also discovered that static analyzers can play an educational role, especially in combination with domain-specific rules.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Static analysis, just-in-time feedback, software quality

ACM Reference Format:
Yuriy Tymchuk, Mohammad Ghafari, and Oscar Nierstrasz. 2018. JIT Feedback — what Experienced Developers like about Static Analysis. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3196321.3196327

1 INTRODUCTION

The assessment of internal software quality attributes mostly relies on the human perception of source code [15]. Manual code review is usually the main approach adopted to ensure software quality, but it is both error-prone and time-consuming [10]. Various static analysis tools exist to automatically find software bugs (*e.g.,* security code smells [17]), however, according to recent research, programmers do not fully benefit from such tools [5, 23].

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196327

We realized that most static analysis tools targeted by recent research are not available in commonly-used development environments, and have to be manually installed and executed. However, many developers are too busy to run a static analyzer on a regular basis [2], so having static analysis integrated into the development workflow can ensure a more systematic use. For example a continuous integration (CI) job can validate code before integration [5]. At the same time, this introduces several other issues such as efforts for setting up a CI server, as well as consensus at a management level, *etc.* [28].

We developed the hypothesis that just-in-time (JIT) feedback was an essential ingredient missing from static analysis tools, necessary for them to gain acceptance by developers. We decided to explore this hypothesis by enriching an existing development environment with JIT feedback for a static analysis tool. We integrated this as a default feature in the base distribution of the Pharo[1] development environment to immediately provide feedback about the code that a software developer is working on. We realized that our JIT analyzer is perceived as being useful by a large number of users [30], and, as confirmed by recent research [12, 16], such a tool considerably improves the performance of developers.

The goal of our study is to investigate the adoption of JIT feedback. To understand how developers use our tool in particular, and to identify the pros and cons of such tools in general, we interviewed 14 Pharo developers IDE who had an average of twelve years of experience in several programming languages, and who were also familiar with static analysis tools in Pharo. The interviewees identified the default availability of the static analysis in their IDEs and its automatic execution as being very important for adoption of the tool. They found the JIT feedback to be a key feature that alerts them immediately within the relevant development context where an issue arises, and eases the identification of false positives. Moreover, novice Pharo developers benefited from our tool by learning about relevant programming patterns and idioms. Nevertheless, we identified certain cases where immediate feedback was not considered to be useful *e.g.,* when developers were warned about unused variables in code that was still under development. We also received conflicting feedback about our user interface design. We therefore conclude that offering customization both for the time frame to report selected analysis results as well as for certain user interface features may well enhance the user experience.

This study provides important insights for communities interested in integrating automatic static analysis into their workflows. In summary, our paper makes the following contributions:

(1) a user experience report on a JIT static analysis tool that is integrated into an ecosystem with a previous long-lived static analysis support;

[1] http://pharo.org

(2) a number of lessons learned about designing a JIT static analyzer based on our findings.

In Section 2 we provide an overview of the current state of the art in the assessment of static analysis tools. Section 3 briefly describes the tool that we are evaluating. In Section 4 we describe the interview setup, and we provide the obtained results and discuss findings in Section 5. We discuss threats to validity in Section 6. Section 7 concludes our paper.

2 RELATED WORK

Johnson *et al.* interviewed software developers to understand why they do or do not use static analysis tools [23]. They quantitatively measured their attitudes towards the following five groups of static analysis features: (1) Tool Output, (2) Supporting Teamwork, (3) User Input and Customizability, (4) Result Understandability, and (5) Developer Workflows. For each group, negative feedback exceeded positive feedback. They found that the most impactful reasons for not using static analysis tools are the high number and poor organization of false positives, weak support for teamwork and customizability, and poor understandability of the tool output. The participants of this study also expressed a need to be informed about issues in their code as soon as they appear. While this study summarizes the shortcomings of various static analysis tools, we identify both positive and negative features, especially based on a static analysis tool that has been successfully adopted by software developers. We confirmed that users require better customizability, and suffer from poor understandability of static analysis reports. In contrast, we discovered that JIT feedback positively coexists with development workflows and helps developers to comprehend the tool output more quickly.

In a later study Christakis and Bird [9] surveyed Microsoft developers to identify the desiderata of static program analyzers. They discovered that a high number of unwanted reports, bad descriptions, and slow execution are the main negative aspects of static analyzers. The authors also learned that developers are mostly interested in static analyzers that capture security issues and violations of best practices. While Christakis and Bird focused on diverse teams of the same company that define their own static analysis practices we focused on diverse individuals that use an IDE with an intrusive static analyzer. Additionally, some of our interviewees are maintainers of open source projects, where code quality assurance and knowledge distribution may be harder to achieve in comparison with products developed by a single team. The authors identified that many developers do not use static analyzers because they do not meet their team's policy. On the other hand we investigate how an IDE can promote static analysis among its users.

Beller *et al.* conducted a large-scale evaluation of static analysis usage in open source projects [5]. Their findings confirm the discoveries of the previous related work. The authors determined that static analysis tools are commonly employed in open-source projects, but only few projects actually integrate such tools into their workflows. Beller *et al.* suggest that static analysis is fully beneficial only if it is integrated into the development workflow, and they suggest to perform integration at least at the level of a CI server. We realized that our interviewees appreciate the integration and JIT feedback of the static analysis. Thus we see IDEs as promising tools for static analysis distribution.

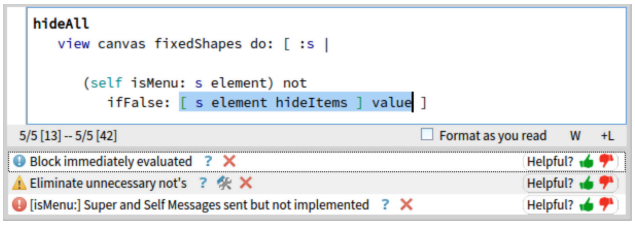

Figure 1: Code area of Pharo's code editor with QualityAssistant in the lower part.

Ayewah *et al.* analyzed the usage of FindBugs at Google [2]. They discovered that while the tool can detect important issues, developers rarely use it mostly because they are overwhelmed by development tasks. The authors discovered that only few developers use static analysis in an automated setup and usually there are no policies in place for using static analysis in development teams.

Do *et al.* introduced a JIT static taint analysis as an Eclipse plugin [12]. The tool was evaluated against an equivalent tool that works in batch mode both by performance tests and by a user study. Cheetah proved to work much faster than the batch-mode tool, and the developers using it were able to cope with their tasks twice as fast. Ganea *et al.* introduced a JIT static analyzer that detects code smells and assists in their refactoring. The tool operates as an Eclipse plugin and thus was evaluated against the capabilities of a default Eclipse distribution. The study has shown that developers who used the JIT static analysis managed to complete twice as many refactoring tasks in the same amount of time, and create code of better quality. Our interviewees also mentioned that JIT feedback increases their performance to react to static analysis results. There exist proprietary IDEs and their extensions such as IntelliJ IDEA, ReSharper, Xcode that provide JIT static analysis feedback with refactoring suggestions and can benefit from our findings about the importance of custom static analysis rules and their adaptability.

Based on interviews with professional software developers [31] Yamashita and Moonen discovered that many of them do not know about code smells [14] or anti-patterns [7]. Those who do however, care about having their code free from the bad practices. The authors identified that developers use technical blogs, programmer forums, colleagues, and industry seminars as their main sources of information. We discovered that our interviewees learned about best development practices directly from the JIT static analysis feedback.

3 QUALITYASSISTANT

We built QualityAssistant for Pharo [13], which supports a dynamically typed object-oriented language (Smalltalk) through a dedicated IDE. It has an active user community that rapidly adopts new releases of the product. This helped us to easily distribute and ensure the use of QualityAssistant. Pharo has a rich history of static analysis support, and included CriticBrowser — an on-demand static analyzer — as a part of the IDE. CriticBrowser used 135 SmallLint [25] quality rules spanning twelve categories such as *potential bugs*, *optimization*, *code style*, *etc.* In general CriticBrowser and SmallLint

are comparable to FindBugs [1], CheckStyle[2] or PMD[3] — the tools often used in related research. Usage of CriticBrowser is also similar to those tools as half of the developers run it on monthly basis or rarer. Based on the described similarities we believe that our discoveries can be generalized to other tools and programming languages as well. To introduce as few differences as possible, QualityAssistant operates with the same set of SmallLint rules used by CriticBrowser.

Currently QualityAssistant is completely integrated as a code editor feature of Pharo. The code editor is based on the Smalltalk *system browser* design and so displays code about only one class or method at a time [18]. Figure 1 displays the code area section of Pharo's code editor. At the bottom of the code editor, QualityAssistant displays a small list of critiques (*i.e.,* static analysis reports) that are present in the active entity such as package, class, method, *etc.* Each list entry refers to one issue detected by a quality rule. Clicking on an entry highlights the relevant part in code. A list entry starts with an icon that symbolizes the severity of the issue based on the corresponding property of the SmallLint rule. The severity icon is followed by a short description of an issue that may be prefixed by a tiny text hint surrounded by square brackets. For example the critique of `isMenu:` in the last line in Figure 1 hints at which method is not implemented. The short description is followed by actions that can be defined by the quality rules themselves. Three common actions for all the rules are: *view rationale*, *ban critique*, and *apply auto fix*. When viewing the rationale of a rule a developer is presented with a longer and more detailed description of the quality violation. By banning a critique a developer can prevent critiques resulting from the same rule from appearing again within the scope of the method where a critique is banned, its class or its package. About 15% of the rules provide a possibility to automatically resolve the issue. The auto fix can be any code transformation implementable as a composite refactory change [26]. Before an auto fix is applied, the developer is presented with the proposed changes. Finally, on the right hand side of each critique entry the "thumbs up" or "thumbs down" buttons allow users to provide feedback with an optional comment about the usefulness of a critique.

JIT static analysis is more complicated to implement than the on-demand batch-processing analogues. The strategies that we followed to solve three major implementation issues are as follows:

Scoping. JIT feedback has to be related only to the current context a developer is working on, thus one has to decide on which scope to perform the analysis. In the Pharo code editor a developer can browse only a single class or method definition at a time and we used the currently browsed entity as the scope of our static analysis. This makes it easy for developers to analyze and consider static analysis results, but it allows only local analyses (for example, a clone between two entities cannot be found in our analysis).

Responsiveness. Static analysis takes time to compute. Although reducing the scope significantly reduces the duration of computation, this often is not enough to provide a live experience. Asynchronous computation with update callbacks for detected violations may improve the responsiveness of the system. In Pharo each class and method is recompiled upon modification, thus the complete system is always compiled. As a result we could always query the bytecode of any method in case a rule requires that data. Large systems that require rich analysis may need to use more complicated approaches such as incremental validation [21].

Feedback Loop. After the integration of QualityAssistant into Pharo, developers started to encounter bugs in the static analysis rules that were present in the system for several years. Previously, the developers simply ignored the bugs by rarely using the on-demand static analyzer. In contrast, we were asked to assist in bug fixing quality rules a few weeks after the AnonymizedTool integration.

Previous research has also described approaches that one can follow to design a JIT static analyzer [12, 16].

4 INTERVIEW SETUP

Our main goal was to identify how our JIT static analysis influences the productivity of software developers. We followed a *sequential exploratory research design* [11] and started with a quantitative survey on the usefulness of various features of QualityAssistant. Details about the survey has been published in an earlier paper [30]. The survey was motivated by the enthusiasm of the first QualityAssistant adopters who contacted us by email to express gratitude, suggest improvements, and report bugs. We invited Smalltalk developers to participate by posting a call on relevant mailing lists and Twitter. In the main question of the survey we asked developers to evaluate the usefulness of the JIT static analysis through a 7-point Likert scale: *very useful, useful, sometimes useful, not influential, sometimes disturbing, disturbing, or very disturbing*. A total of 29 developers responded. The responses are presented in Figure 2. More than 90% of developers find QualityAssistant to be useful to some extent, and almost half of all survey participants find the JIT feedback to be very useful. To investigate the reason for such high acceptance of QualityAssistant and to identify advantages and shortcomings of JIT static analysis we designed an interview with open-ended questions that are designed to answer the following research questions:

RQ1 — *what are the positive and negative features of QualityAssistant?* We wanted to learn to what extent each feature can accommodate developers needs.

RQ2 — *what are the good and bad quality rules?* QualityAssistant motivated adding a few domain-specific rules, and removing some rules in the original quality rule base of Pharo. We wanted to see which rules developers like and dislike to understand if we are moving in the right direction.

RQ3 — *what is the impact of a static analysis tool like QualityAssistant on its users?* We conjectured that constantly reacting to the analysis reports will improve developers' coding skills over time.

We wanted to interview developers who are experienced, work in various organizations, and actively use Pharo and so are familiar with QualityAssistant. We therefore conducted most of the interviews with participants of the Pharo Days conference[4] which attracts experienced Pharo users. We recorded 14 interviews of an average duration of 14 minutes.

[2] http://checkstyle.sourceforge.net
[3] https://pmd.github.io
[4] https://medium.com/concerning-pharo/pharo-days-2016-c52fe4d7caf

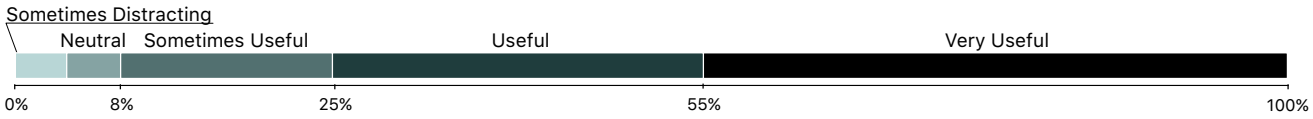

Figure 2: Perceived usefulness of QualityAssistant according to surveyed developers.

We performed semi-structured interviews with these software developers based on the template questions presented in Table 1. We used a couple of test interviews to fine tune the template, however during the interviews we dynamically added or removed some questions to obtain the most information possible. The first two groups of questions solicit the interviewees background by assessing their programming experience and knowledge of code quality concepts. The remaining three groups of questions are designed to evaluate the usefulness of QualityAssistant, the static analysis rules, and the overall impact of the JIT feedback on the interviewees' daily development tasks.

Table 1: Questions for structuring the interviews.

Experience	What is your name and occupation? How much of programming experience do you have in Pharo and other languages?
Quality	What is good code? What are code smells? Which smell-detection tools did you use?
QualityAssistant	How do you use QualityAssistant? What is helpful? What is distracting?
Rules	Which critiques are helpful and which are not?
Impact	Did the way you program changed because of QualityAssistant? Did you learn something because of QualityAssistant?

5 INTERVIEW RESULTS

A single person manually transcribed and coded the interview recordings, following Gordon's guidelines by assigning codes to the transcribed information [19].[5] The interviews were relatively short and the groups of interview questions served well as initial coding categories. Later we added a couple of more categories such as "improvement suggestions" that emerged during the initial analysis. Then we reviewed the data related to each code and summarized each as a finding. The codes serve further in the paper as categories for bar-charts. In the following, we discuss the lessons learned from the interview responses and summarize them into a few categories.

5.1 Developers' Background

We asked the interviewees to estimate their programming experience. Many of the interview participants claimed to have been "programming since high school" but we considered only their post-university experience. A high proportion of participants (*i.e.,* five out of fourteen) have more than *twenty years* of programming experience, and three participants have more than *ten years* of experience. Only two interviewees have programmed for less than *five years*. We believe that the high number of experienced developers is beneficial for the interview, as they should provide more reliable feedback.

Only two participants have used Pharo for less than *three years* and three interviewees were experts who used Pharo for more than *six years*. The remaining nine are fairly experienced developers who have developed in Pharo from *three to six years*.

Java is the most popular language with eleven interview participants being experienced in it. *Python* and *C++* share the second place, each being used by four interviewees, and each of the *C*, *Javascript* and *Lisp* programming languages were used by three developers.

To identify to which extent the developers are familiar with the concept of code quality and best development practices we asked the interviewees what is *good code* in their opinion. Figure 3 summarizes the most common aspects of good code according to the answers. The criteria are not mutually exclusive because we wanted to record the exact opinions. Over two thirds of the interview participants stated that good code is *easy to read* and *easy to understand*. More precisely, three developers identified that good code uses *good names and abstractions* and two of them said that it is *modular*. According to three interviewees, another important feature of good code is the *absence of any additional complexity* besides that which is essential. Availability of good *documentation* is another prerequisite of good code according to three developers. The interviewees also mentioned properties such as: extensible, concise, well-tested, continuously integrated, maintainable, and clean. According to two participants, the definition of good code is context-dependent. For example efficient bit-shifting operations may be hard to understand by reading the code itself, but could be well-documented.

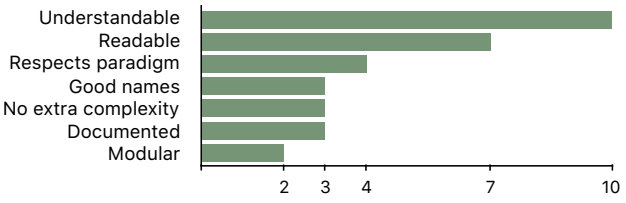

Figure 3: Top aspects of good code according to the interviewees.

We conclude that the interviewees are quite familiar with the notion of code quality as their responses highly correlate with the maintainability aspect of the ISO software quality standard [22]. They knew about code smells, and except for one person who had only used QualityAssistant, others had also experienced other static analysis tools to detect code quality issues. Such a high percentage of developers knowing about code smells is uncommon when compared to related research [31], and may result from the high number of experienced engineers who participated in our interview. About

[5]The study data are available online: http://scg.unibe.ch/research/QualityAssistant-study

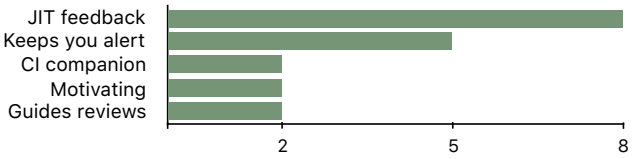

Figure 4: Top positive feedback aspects of QualityAssistant.

half of the interviewees used CriticBrowser [30], the static analysis tool available for the last decade in many Smalltalk environments including Pharo. There was no other particular tool that was used by a significant number of participants, but they mentioned using *Java tools* like Checkstyle and PMD, *lint tools* like PyLint and JSLint, *IDEs with quality reports* like IntelliJ, PHPStorm, Eclipse, CodeBlocks and the SonarCube at an integration server.

5.2 QualityAssistant Feedback

This subsection discusses the responses of the interviewees and highlights important lessons related to RQ1 which is about the positive and negative features of QualityAssistant. All but one of the interview participants provided some kind of positive feedback summarized in Figure 4.

According to four developers, an integrated tool that runs automatically without a special setup was crucial for their static analysis adoption.

> *Lesson 1.* Integration and out-of-the-box functionality of static analysis significantly improves its adoption.
>
> ***
>
> Previous research identified that the lack of developer discipline and weak organization rules result in underuse of static analyzers [2, 9]. Integration of a JIT static analysis into the main development tool can promote its systematic use.

Three interviewees identified that QualityAssistant operates on a limited scope, which speeds up the static analysis execution and shortens the time between changes made to code and detection of an issue. For example, a developer mentioned that running static analysis on the complete project he is working on takes about ten minutes, while QualityAssistant can provide an immediate feedback about the code he is currently working on. The interview participants found a scoped feedback easier to process, as it is limited to a small part of the code that they are currently focusing on, and consequently the number of critiques decreases proportionally to the scope.

> *Lesson 2.* The scoped nature of a JIT static analysis comes with execution speedups and smaller reports that a developer can comprehend more quickly.
>
> ***
>
> Previous research has identified that static analysis during code integration yields reports that are easier to comprehend because developers recently worked on the validated code [8]. We build on this finding and show how easy it is to process the JIT analysis feedback.

About a quarter of the developers noted that QualityAssistant can be used as a *sidekick of a CI server*. They said that sometimes you need to wait a significant amount of time while a CI server is validating the project, just to find out that you have made a mistake. Then you have to fix that mistake and start the integration process from the beginning. On the other hand QualityAssistant can point out the same mistake just when it is introduced. Thus JIT static analysis acts as a personal assistant that quickly highlights suspicious parts in code, while the CI server ensures certain standards for all contributions. We realized that this synergy is not limited to analyzers that operate during continuous integration, but also extends to other batch analysis tools. For example, one developer stated that he rarely uses a standalone tool that summarizes all the issues of interest for a whole project (or its part) to understand the overall state of the software, but he enjoys checking QualityAssistant reports during programming sessions.

> *Lesson 3.* Any type of static analysis tool could be useful, but JIT feedback occupies an independent niche. It acts as a quick development guidance, while a CI server ensures global rule adherence.
>
> ***
>
> Previous research treats a static analyzer as a single independent tool that is either used directly by developers [23], or is launched by an integration server [27]. In contrast, our interviewees found that besides the available validation infrastructure on a CI server, immediate feedback has its own features that not only are useful, but also complement the analysis done on the CI server.

More than one third of the participants noticed that QualityAssistant *keeps them alert* and has saved them a few times when they made a mistake, which is hard to spot immediately, but may cause unexpected behavior in the future. For example, bugs can occur when the result of a boolean expression is compared with a variable instead of being assigned to it. These developers said that while traditional debugging could take them some time to find the error, QualityAssistant pointed out the issue immediately. One developer mentioned that QualityAssistant saved him time when refactoring a core graphics framework, since an error could break the UI *including the debugger*, thus disabling safe recovery. According to two developers, QualityAssistant *makes them think* more about the code and in this way *motivates* them to make their code better. This applies to the warnings about missing documentation, use of low-level and reflective APIs and use of side effects. Two developers mentioned that sometimes critiques produced by QualityAssistant are annoying because their impact is not immediately perceivable, yet they adapted suggestions to deliver a higher-quality code. The senior developers who are part of the Pharo patch review process said that QualityAssistant *guides code reviews* by highlighting questionable pieces of code while allowing developers to focus on the actual functionality changes brought by the patch. Moreover, during code inspection developers can instantly react to certain critiques as they are already focused on the related piece of code.

> *Lesson 4.* JIT feedback motivates developers to write better code and keeps them alert about the possible mistakes or highlights suspicious parts of code during a review.
>
> ***
>
> Similar to previous research [3, 4], the interviewees confirmed that QualityAssistant draws their attention to suspicious

code during common development tasks, and they can focus more on other important concerns during code review.

A few common points were raised regarding the negative feedback. These issues are presented in Figure 5. According to the interviewees *false positives* were one of the most important problems concerning four developers. The interview participants acknowledged that false positives are common in static analysis, but they are much more distracting in a standalone setup or CI server, where they may need to review hundreds of reports many of which are false positives. However in a JIT feedback setup they had to focus only on a few reports related to their current context, which significantly eases identification of false positives.

Lesson 5. JIT static analysis also suffers from false positive detections. However, the small scope and related context of the reports make it easier for developers to deal with them.

***

While employing advanced analysis techniques could decrease the number of false positives, the usual practice is to manually inspect the code to find false positives. Our tool eases inspection by its JIT capabilities. This finding complements that of lesson 2.

Figure 5: Top negative feedback topics.

Four interviewees complained about *annoying rules*. Some of the critiques are treated as annoying because they cannot be addressed easily. For example, if a developer has to perform a complicated refactoring to reduce the depth of a hierarchy, the critique will be most likely ignored. For certain rules some developers simply do not agree with the rationale. For example, developers did not see the significance of the rule that checked wether each class is documented. We focus on the details of good and bad rules in the following subsection 5.3.

We noticed that the attitude towards false positives and annoying rules greatly depends on the experience of interviewees and their attitudes. According to the three participants represented by *experienced developers* who manage teams or review contributions, it is worth *spending extra time* and paying attention to static analysis reports. Senior developers stated that it is ok if you see a critique that you cannot act upon because it still motivates you to rethink the current implementation. On the other hand, other three participants who have fewer years of programming experience tend to claim that their code is well written and they do not care about some rules especially if they are the single developer on the project. For instance, one acknowledged agreeing that excessive usage of meta-programming is not recommended, but that he has to use it due to certain application requirements, and gets annoyed by critiques that warn against it.

Lesson 6. Experienced developers tend to react well to static analysis feedback more than less-experienced developers who may be too self-confident to accept criticism.

Unclear explanation of critiques is the most severe problem for four developers. The problem itself has two parts: the explanation of the critique and localization of the issue in the code. The former arises when the rationale behind a quality rule is not clear and developers cannot understand why their code is bad. The latter issue occurs when the critique does not provide enough information to understand what exactly triggered the rule and how to fix it. For example an interviewee mentioned a rule designed to detect deprecated method invocations. However, critiques produced by that rule reported a complete method that was invoking a deprecated method rather than identifying the specific invocation.

Lesson 7. Critiques may fail either to explain why a detection is a violation, or to specify which piece of code violates the rule. A feedback loop from static analysis users to developers of rules is very helpful in such cases.

***

The negative impact of unclear critique explanations is previously discussed in the literature [6]. We use a feedback loop, similarly to that of Tricorder [27], that allows users to quickly notify us about such critiques. Within six months of the integration of QualityAssistant we fixed around 20 rules.

Finally, almost one third of developers complained about *user experience*. The interviewees did not like the implementation used for banning critiques. It reuses the strategy of SmallLint where the source code is modified with certain annotations. The developers did not like the changes introduced into their source code just because they do not want to see a critique again. However, this issue is an artifact of the SmallLint engine and thus we do not discuss it further in this paper.

QualityAssistant's user interface was not efficient for four interviewees. One of them did not like that it takes up some space from the code editor and asked if it would be possible to simply highlight criticized code chunks. In contrast, another developer did not like the fact that only up to three quality critiques are visible at a time. He stated that the initially presented critiques may not be as important as the following ones which are hidden at the bottom in the list. In subsection 5.5 we provide a scenario where the current design supports quick comprehension of critiques.

5.3 Rule Usefulness

Despite the high ratio of positive feedback, most of the interview participants struggled to identify the rules that are useful for them. One third of them agreed that good rules are related to a *specific context*, such as a certain project. One developer went into detail and told us that the rules of Glamorous Toolkit[6] were extremely useful for him. Glamorous Toolkit is a project developed externally but integrated and shipped with Pharo. It has its own domain-specific language (DSL) for scripting user interfaces. It also provides SmallLint rules that check the DSL and suggest transformations to enable lazy initialization and so improve the UI performance. By providing

[6]http://gtoolkit.org

its own rules the project educates unfamiliar developers about the best way to use the UI framework.

Consider the example code snippet in Listing 1. Two lines were intended to represent two statements, but the period character that serves as the statement delimiter in Smalltalk is omitted from the end of the first line. This results in a single statement where the result of an expression is assigned to the `aLink` variable. Furthermore the expression consists of a message[7] chain `asLink self isEmpty ifTrue:` sent to the `aLinkOrObject` variable. The exception will be raised when the result of `aLinkOrObject asLink` receives the `self` message, as that method is not implemented. However, there is a rule that checks for messages with the `self` selector and warns that they look suspicious.

```
1 aLink := aLinkOrObject asLink
2 self isEmpty ifTrue: [lastLink := aLink].
```

Listing 1: Missing statement separation

A summary of the most common rules identified as "bad" is shown in Figure 6. We identified three groups of such rules and each of them was mentioned by about a quarter of the interview participants. The first category consists of rules that are *based on metrics* such as classes with too many methods, methods with too many lines of code, *etc.* Developers reported that most of these issues require significant effort to resolve and usually do not make sense in the setup of QualityAssistant, where you want to have issues that just appear and can be quickly resolved.

Figure 6: Negative aspects of rules according to the interviewees

The second category contains useful rules that are *not really just-in-time friendly*. In fact, these rules (*e.g.,* check for unused variables, uncommented classes or debugging code present in the project) are valid but not in the JIT way of displaying issues. For example, ideally your code should not contain debugging statements, but two participants stated that the report was disturbing during a debugging session. On the other hand, three developers liked such reports and treated them as a todo list: you have created a variable, now you have a task to use it. Therefore, we decided to add the possibility of flagging a report as a todo in the next release.

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

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

Listing 2: Smalltalk cascade example

Lesson 8. Immediate feedback is just-in-time for most of the critiques, yet certain critiques are distracting when they are instantly reported.

***

We suggest not to search for the perfect time to report static analysis results, but rather to try to identify in which contexts do certain rules excel, or how a developer can choose in which context she wants to see certain critiques. We hypothesize that the critiques that some developers do not like in the JIT feedback can still be useful to them in other contexts like pre-commit validation.

The third group refers to *known Pharo idioms*. A rule found to be especially irritating by developers detected "cascading messages" that did not end with the `yourself` message. Cascades are a concept specific to Smalltalk, as illustrated in Listing 2. On the first line an instance of `ToolDockingBarMorph` is created. The remaining 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, thus avoiding the need 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, its return value (the adopted model) will be also returned by the whole expression, while the desired result is the instance of `ToolDockingBarMorph`. To avoid this kind of problem, one of the rules recommends to always end cascades with the `yourself` message. This message simply returns the receiver *i.e.,* the the instance of `ToolDockingBarMorph` in our example. While this rule offers a useful warning for novices who are not aware of the pitfalls of Smalltalk cascades, it can be absolutely annoying for experienced developers who want to use a different last message on purpose. On the other hand, one interviewee stated that when he rewrites his code to avoid the critique, the code becomes more understandable.

Another idiomatic rule mentioned by the interviewees detects use of the reflective API, such as checking the type of an object. Similarly to the previous rule this one may explain that there are other more appropriate ways to solve general problems without the support of reflective API, but if the developer knows what he or she is doing, then the rule is simply distracting. We cannot draw a definite conclusion on this rule as one of the core Pharo developers told us that this rule is useful because it draws attention to suspicious code during a review.

Regarding RQ2 which asks about the good and bad quality rules, we can say that the developers like domain-specific rules, especially if they come from other projects. Developers especially dislike rules producing a high number of false positive or annoying critiques. In more detail, three developers did not like the critiques that cannot

be easily acted upon, but we believe that they can still benefit from those critiques if they are reported in a different context. We realised that the developers also require a possibility to disable certain rules for a specific domain or a project. The developers also identified metric-based critiques that require a significant refactoring effort as a special group of critiques that do not work well in the JIT setup. Finally, three experienced developers did not like the critiques about common programming idioms.

5.4 Impact on individuals

Half of the interview participants believe that their programming habits changed because of QualityAssistant, which is an important finding for RQ3. Some developers simply became motivated by QualityAssistant and started to follow the suggested best programming practices, like writing class comments. For those carrying out code reviews, it detects simple issues, and in the second pass the developers can focus on what the tool missed. In general, many developers mentioned that their habits changed to quickly react to issues as soon as they are reported.

Furthermore, we observed that JIT static analysis also possesses a teaching capability which is useful for developers at any level. Novice developers learned about optimization techniques and some Pharo-specific idioms. For instance, one instructor stated that many of his students write code to evaluate the equality of a boolean expression and `true` or `false` literals and such a tool could educate them that this is a bad practice. (`(b == true)` is the same as `b`.) Senior developers mostly learned about common style guidelines and approaches to make their code more portable across different Smalltalk dialects. For example, a number of them learned that instead of using `expression1 & expression2` they can use `expression1 and: [ expression2 ]` so the second expression will be evaluated lazily in case the first one returns true. Experienced Pharo developers also learned about API changes to Pharo and other frameworks that were used in their project. For instance, the rules in Glamorous Toolkit suggested some developers how to use the DSL more efficiently. Another given example was the API change of the SUnit testing framework that provides more informative output if one uses:

```
self assert: actualValue equals: expectedValue
```

instead of:

```
self assert: (actualValue = expectedValue)
```

Lesson 9. JIT static analysis can serve as documentation of best practices relevant to the exact context where you need them.

***

The classic approach is to read the documentation or the change log. We noticed a JIT static analysis tool can provide domain specific knowledge which perfectly serves as live documentation that relieves developers from consulting external resources.

5.5 Usefulness for Novices

Both the initial survey and the interviews showed that most developers find QualityAssistant playing the role of an artificial pair programmer to be more helpful than distracting. One of the interviewees stated that *"We (experts) are so used to jump over things while trying to understand code, that one or two lines of the critiques do not impose much more distraction. I hypothesize that QualityAssistant can be distracting for novice programmers, as they are not used to quickly skimming over a large amount of information."* Moreover, nine of the interviewees stated that when developers are presented only with critiques about the code that they are working on, it reduces the information pressure. Finally, static analysis tools for Java detect on average 40 issues for every thousand lines of code [20] while an average Java or C++ method has ten lines of code [24]. This suggests that while working on a method a developer would encounter on average 0.4 critiques.

An open question raised by the interviews is whether the JIT feedback is also useful for novice developers, or whether they find it distracting. To shed more light onto this issue, we carried out a preliminary survey of Masters students of a Software Modeling and Analysis course to identify whether QualityAssistant was useful to them. For most of the students this was their first encounter with Pharo, thus we find the selected group of participants to well represent the *novice developers* category. To avoid bias, we did not offer any reward for the survey participation, the participation was voluntary, and we allowed students to stay anonymous. Seven students participated in the survey and only one student knew Pharo for half a year before the course. Five of the students had an average of $1.8 \pm 1M \pm SD$ years of industrial development experience. Two of the students had previous experience with JIT static analyzers in the IntelliJ IDEA and ReSharper. We asked students to evaluate the usefulness of QualityAssistant on a 7-point Likert scale: *very useful, useful, sometimes useful, not influential, sometimes disturbing, disturbing, or very disturbing*.

The responses are presented in Figure 7. It is worth noticing that all the students claimed that QualityAssistant was useful for them. In the freeform feedback they specified that QualityAssistant taught them about the functionality that they did not know before as well as some programming concepts of Pharo. On the negative side a student reported the user interface to be user-unfriendly, and found some critique explanations hard to understand.

Figure 7: Usefulness of QualityAssistant from students perspective.

6 THREATS TO VALIDITY

The semi-structured interviews were performed by the developer of QualityAssistant, which may have biased the interviewees to report positively about the tool. In addition, the interviewees may be more positive because they are early adopters and because the Pharo community is generally accepting towards new tools, and tolerant to issues that arise in research or beta tools. The interviewer strove not to lead or influence the interviewees. Besides, we expect the interviewees to be honest in their responses as they are not evaluating a random tool that they can decide not to use, but an internal feature in their IDEs that can influence the future of their development environment. Moreover, to minimize the effects of this threat we tried to focus more on the stories that interviewees told about their experience with QualityAssistant rather than qualitative

feedback where they tried to assess whether something is good or bad. Only a single person coded the interview results. It is possible that a second person may have identified different codes.

While Pharo is similar to other IDEs for object-oriented languages such as Java, Python or Objective-C, there are certain differences, particularly related to its support of live programming, that may not be generalizable. Similarly, QualityAssistant does not have many conceptual differences to other similar tools. However, we expect that there may be bias on the "community level." For example the Smalltalk community in general expects there to be a dedicated IDE that will assist in debugging, refactoring, *etc.* On the other hand, there are communities that would rather take a basic but extensible editor like EMACS [29] and add other tools on their own. We believe that our interviewees had significantly high programming experience, and worked with different technologies through their career to provide reasonable feedback. On the other hand, the high degree of interviewees' development experience may introduce bias when generalizing our findings to less experienced developers.

All our participants were aware of the concept of code smells. This differs from the population considered by other research [6, 31], which means that our findings should not be directly compared with others.

7 CONCLUSION

In this paper we present a user experience study of a JIT static analysis tool integrated into an IDE based on 14 interviews with experienced developers. Our results show that JIT feedback is highly beneficial, as it brings up possible issues at the time when a programmer is looking at related code, and thus reduces the time needed to understand the context of a critique. Also, the integration and automatic execution of the static analyzer played a crucial role in its adoption.

Our recommendation is to integrate JIT feedback directly into the development environment, by providing analysis results that are *directly related to the software artifact* currently being manipulated by the developer. The JIT feedback should be *actionable* for the developer, and should provide *explanations* to justify the proposed actions. *False positives* should be minimized; a *feedback loop* allowing the developer to ban undesirable critiques and to inform the rule designer of shortcomings can be an effective way to reduce false positives. To maximize the value of JIT feedback, it should also be possible to add project-specific quality rules.

We also discovered that not all developers like to be immediately notified of all available critiques. Based on our use cases we believe that there should be multiple reporting tools throughout the development process that can use a unified static analysis model. This way a developer will be able to decide in which time frame she wants to see the critiques of a certain rule. We realized that developers at any level can learn from JIT static analysis. Novices learn basic programming guidelines and patterns of the programming language while more experienced developers learn about optimization tricks and portability guidelines.

We believe that the quality rules about third party projects are one major feature that is lacking in the QualityAssistant, and we plan to investigate the ways to boost the rule creation by ordinary developers in the future. Currently we have no insights into how JIT feedback actually impacts developer productivity or software quality. Possible future work would include an in-depth study to assess such impact.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

REFERENCES

[1] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *Software, IEEE* 25, 5 (Sept. 2008), 22–29. https://doi.org/10.1109/MS.2008.130

[2] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Experiences Using Static Analysis to Find Bugs. *IEEE Software* 25 (2008), 22–29. Special issue on software development tools, September/October (25:5).

[3] Nathaniel Ayewah and William Pugh. 2010. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 241–252. https://doi.org/10.1145/1831708.1831738

[4] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 712–721. http://dl.acm.org/citation.cfm?id=2486788.2486882

[5] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 470–481. https://doi.org/10.1109/SANER.2016.105

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press.

[8] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. *Moving Fast with Software Verification*. Springer International Publishing, 3–11. https://doi.org/10.1007/978-3-319-17524-9_1

[9] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[10] Jason Cohen, Eric Brown, Brandon DuRette, and Steven Teleki. 2006. *Best kept secrets of peer code review*. Smart Bear.

[11] John W. Creswell and Vicki. 2006. *Designing and Conducting Mixed Methods Research* (1 ed.). Sage Publications, Inc. http://www.worldcat.org/isbn/1412927927

[12] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, 307–317. https://doi.org/10.1145/3092703.3092705

[13] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, and Dimitris Chloupis. 2017. *Pharo by Example 5.0*. Square Bracket Associates. http://files.pharo.org/books/updated-pharo-by-example/

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

[15] Steve Freeman and Nat Pryce. 2009. *Growing Object-Oriented Software, Guided by Tests* (1st ed.). Addison-Wesley Professional.

[16] George Ganea, Ioana Verebi, and Radu Marinescu. 2017. Continuous quality assessment with inCode. *Science of Computer Programming* 134 (2017), 19–36. https://doi.org/10.1016/j.scico.2015.02.007

[17] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security Smells in Android. In *17th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 121–130. https://doi.org/10.1109/SCAM.2017.24

[18] Adele Goldberg and David Robson. 1983. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., Chapter 17. The Programming Interface, 291–328. http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[19] R.L. Gorden. 1998. *Basic Interviewing Skills*. Waveland PressInc.

[20] Sarah Heckman and Laurie Williams. 2008. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In

Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08). ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/1414004.1414013

[21] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. 2014. Teamscale: Software Quality Control in Real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 592–595. https://doi.org/10.1145/2591062.2591068

[22] ISO/IEC. 2010. ISO/IEC 25010 — Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. (2010).

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

[24] Michele Lanza and Radu Marinescu. 2006. *Object-Oriented Metrics in Practice*. Springer-Verlag. http://www.springer.com/de/book/9783540244295

[25] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. 1996. An Automated Refactoring Tool. In *Proceedings of ICAST '96, Chicago, IL*.

[26] Donald Bradley Roberts. 1999. *Practical Analysis for Refactoring*. Ph.D. Dissertation. University of Illinois. http://historical.ncstrl.org/tr/pdf/uiuc_cs/UIUCDCS-R-99-2092.pdf

[27] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 598–608. http://dl.acm.org/citation.cfm?id=2818754.2818828

[28] Flash Sheridan. 2012. Deploying Static Analysis. *Dr. Dobb's Journal* (Aug. 2012), 8–14. http://www.rahul.net/flash/Deploying_Static_Analysis.pdf

[29] Richard M. Stallman. 1981. EMACS the Extensible, Customizable Self-documenting Display Editor. *ACM SIGOA Newsletter* 2, 1-2 (April 1981), 147–156. https://doi.org/10.1145/1159890.806466

[30] Yuriy Tymchuk. 2015. What if Clippy Would Criticize Your Code?. In *BENEVOL'15: Proceedings of the 14th edition of the Belgian-Netherlands software evoLution seminar*. http://yuriy.tymch.uk/papers/benevol15.pdf

[31] Aiko Yamashita and Leon Moonen. 2013. Do developers care about code smells? An exploratory survey. *WCRE'13* (2013), 242–251. https://doi.org/10.1109/WCRE.2013.6671299