What if Clippy Would Criticize Your Code?

Yuriy Tymchuk
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—Modern development tools can aid in software quality assessment by using static analysis. Some of these tools are integrated into a development environment while others work outside of it. Although both approaches have their pros and cons there is no clear evidence whether one of them ensures a better quality of a software system.**

In order to understand which kind of tools performs better, we surveyed the developers of one open source community about their experience with both kinds of tools. The results show that developers may prefer integrated tools over external ones.

I. Introduction

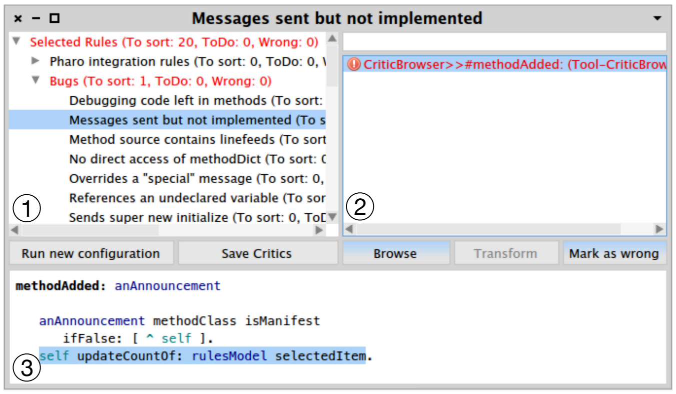
Quality of code in modern software systems is as important as their functionality. Bad code quality may result in additional costs required to maintain and evolve the software system. One of the most common practices to ensure a good quality of software is a *code review* [1]. Manually reviewing code is time-consuming [2]. To reduce the reviewers effort, static analysis tools can be used to automatically detect bad patterns in source code or to validate a system against some metrics [3], [4]. One of the most popular static analyzers is the FindBugs tool which automatically detects potential bugs in Java code [5].

Many static analyzers are standalone tools that can be run on a source code and produce a report about the violations of some coding rules. On the other hand there are some analyzers which are integrated into development tools. For example IntelliJ IDEA[1] is a Java integrated development environment (IDE) which provides a live statical analyzer feedback directly in the code editor. Another example is the Tricorder tool which provides static analysis feedback during a pre-commit code review [6].

Having a static analyzer integrated into your development environment can be distracting, as it will interrupt you with every critic that appears in your code. This may be frustrating as often during development there are incomplete pieces of implementation that can become a source of critics. People usually reference Clippy — the infamous Microsoft Office Assistant during such situations, as it was causing more distraction than providing aid. On the other hand running external tools usually takes more time, and developers tend to ignore this practice. Maybe having a live feedback about your code may motivate developers to keep the quality at a good level.

We want to investigate whether integrated quality analyzers perform better than external ones. For this purpose we are working with Pharo[2] a dynamic object-oriented language and IDE inspired by Smalltalk [7]. We use Pharo because it is easy

[1]https://www.jetbrains.com/idea/
[2]http://pharo.org/



Selected Rules (To sort: 20, ToDo: 0, Wrong: 0)
Pharo integration rules (To sort: 0, ToDo: 0, \
Bugs (To sort: 1, ToDo: 0, Wrong: 0)
Debugging code left in methods (To sort:
Messages sent but not implemented (To s
Method source contains linefeeds (To sort
No direct access of methodDict (To sort: (
Overrides a "special" message (To sort: 0,
References an undeclared variable (To sor
Sends super new initialize (To sort: 0, ToD

CriticBrowser>>#methodAdded: (Tool-CriticBrow

```
methodAdded: anAnnouncement

  anAnnouncement methodClass isManifest
     ifFalse: [ ^ self ].
  self updateCountOf: rulesModel selectedItem.
```

Fig. 1: Critic Browser panes: ① quality rules; ② rule's critics; ③ criticized source code.

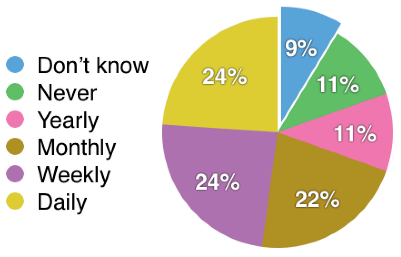
to integrate new features in the IDE, the community around it is active, diverse and has experienced software engineers. Pharo developers use a static analyzer called SmallLint [8]. At this moment it has 120 quality rules grouped into 6 categories such as bugs, style and optimization. A violation of a quality rule by some piece of code is called a *critic*. While SmallLint simply defines a model of rules and an infrastructure to run them, it is more convenient to use a graphical user interface (GUI) to work with critics. Before beginning our study the only available GUI tool for SmallLint in Pharo was Critic Browser. It is an external tool, which means that in order to use Critic Browser a developer has to leave the place where he or she is developing code. Not having any static analyzer for Pharo with live feedback can provide us with valuable data of how the developers were treating code quality before having an integrated live–feedback tool. Later we plan to compare it with the experience of using such a tool.

II. Critic Browser

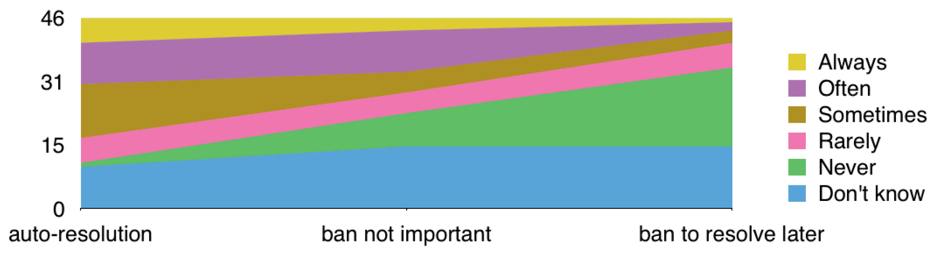
Critic Browser is a graphical tool that allows a developer to select a subset of SmallLint quality rules that will be used to analyze a selected list of software packages. The result of this analysis is presented in a window on Figure 1.

In the **quality rules** pane the rules are grouped into categories and presented as tree list. When a rule is selected, all the critics produced by these rules are displayed in the critics pane. At the same time the rule's rationale is displayed in the source code pane.

The **critics** pane contains a list of classes and methods that violate the selected rule. Selecting one of them will show its source code, and allow one to mark the critic as false positive.

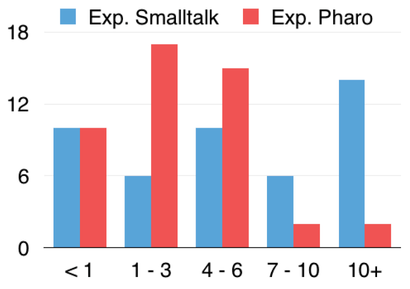


(a) Usage of Critic Browser

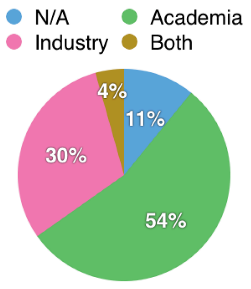


(b) Usage of Critic Browser features. The features are situated on X axis. Y axis corresponds to the number of responses

Fig. 2: Usage of Critic Browser and its features



(a) Experience in years



(b) Experience source

Fig. 3: Development experience of Critic Browser survey participants

Certain rules allow automated resolution of the critic by source code rewriting.

The **source code** pane displays the source code of a selected entity and highlights a section detected by the quality rule. A developer can modify the code in place and save it. For the rules that allow an automatic resolution, a unified diff of proposed changes is presented.

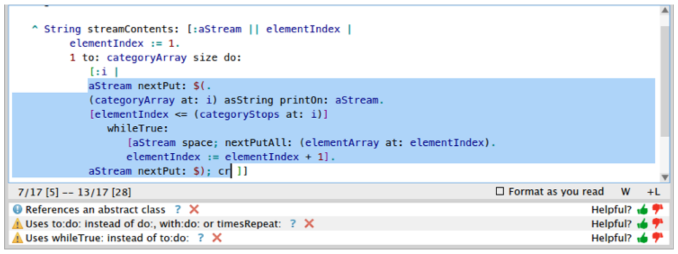
III. Critics Browser Survey

In order to understand how Pharo developers work with static analysis we conducted a survey while Critic Browser was the only static analysis tool available in Pharo.

In total 46 developers participated in the survey. They could identify their programming experience in years both for Pharo and Smalltalk in general. Also participants could provide the source of their experience as either academia or industry. The summary of participants' experience is shown on the Figure 3. Most of participants' experience in Pharo is almost uniformly distributed on a range from 0 to 6 years. Also the participants have a diverse experience in Smalltalk development, with a large group of developing in Smalltalk for more than 10 years. The number of participants from academia is almost twice as high as the number of participants from industry.

In this survey we wanted to obtain the answers for the following questions:

1) How often do developers use Critic Browser?
2) How often do developers automatically resolve critics?
3) How often do developers mark a critic as a false positive because critic is not important?



```
  ^ String streamContents: [:aStream || elementIndex |
      elementIndex := 1.
      1 to: categoryArray size do:
         [:i |
         aStream nextPut: $(.
         (categoryArray at: i) asString printOn: aStream.
         [elementIndex <= (categoryStops at: i)]
            whileTrue:
               [aStream space; nextPutAll: (elementArray at: elementIndex).
               elementIndex := elementIndex + 1].
         aStream nextPut: $); cr ]]
```

7/17 [5] -- 13/17 [28]   ☐ Format as you read   W   +L

References an abstract class ? ✕ Helpful?
Uses to:do: instead of do:, with:do: or timesRepeat: ? ✕ Helpful?
Uses whileTrue: instead of to:do: ? ✕ Helpful?

Fig. 4: Coding area of Nautilus with QualityAssistant critics below.

4) How often do developers mark a critic as a false positive because they don't have time to resolve it now?

For the first question we have asked the participants to provide an answer on 5-point Likert [9] scale: 1) daily 2) weekly 3) monthly 4) yearly 5) never. The responses that also include participants who are not familiar with Critic Browser are shown on Figure 2a. About 50% of developers are using Critic Browser less often than once a week. Usually a week of programming can change the software dramatically. Also 25% of developers use the tool less often that once a month, which can be treaded as not using the tool at all.

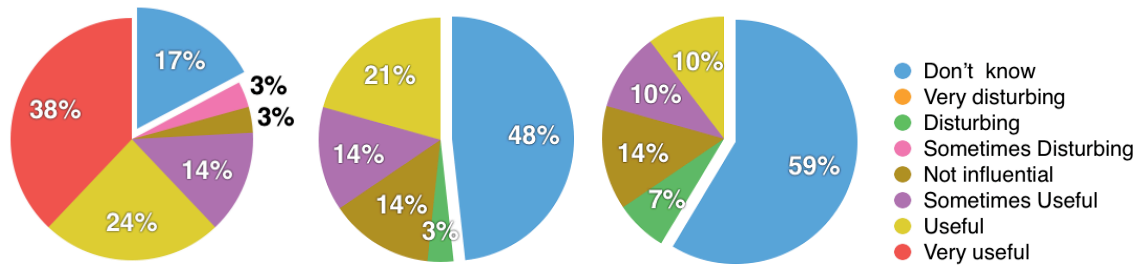
We also asked developers whether they are familiar with each feature (the last two questions share a feature which is marking critic as false-positive). For these questions we used a 5-point Likert scale: 1) never 2) rarely 3) sometimes 4) often 5) always. The responses are shown as a stacked area chart on Figure 2b. The features are situated on the X axis and the number of responses is represented on Y axis. Each area corresponds to a different answer option.

IV. QualityAssistant

To understand how developers react to a live feedback about their code we have developed QualityAssistant (QA). It has a live quality feedback engine based on SmallLint, and plugins for the tools commonly used in Pharo development process.

The main plugin of QualityAssistant works with Nautilus, the code browser and editor most commonly used for Pharo development. This plugin appears as a list of critics below the coding area, as can be seen in Figure 4.

Each critic has a designated severity icon which can be one of: information, warning or error. QualityAssistant relies on SmallLint and so provides features similar to Critic Browser.



(a) Codding browser (b) Inspector (c) Spotter

Fig. 5: Usefullness of QualityAssistant plugins

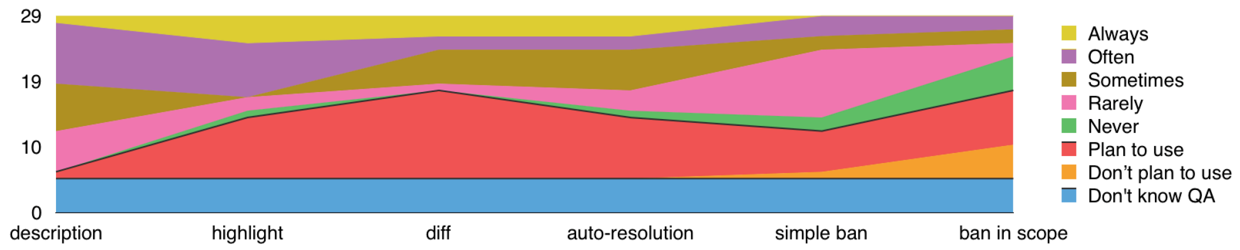
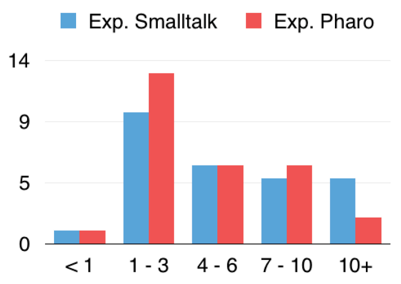
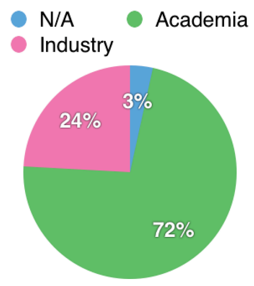


Fig. 6: Usage of QualityAssistant features. The features are situated on X axis. Y axis corresponds to the number of responses



(a) Experience in years



(b) Experience source

Fig. 7: Development experience of QualityAssistant survey participants

Clicking on a list item which represents a critic will highlight a relevant piece of code. Also for each critic a user is able to read the description and to ban the critic. QualityAssistant also allows one to ban a critic for a higher scope *i.e.,* a critic of a method can be banned on a level of a class and so it will not be applied anymore to the methods of that class. For the rules which provide automated resolution the user can preview the proposed code changes, and apply them if desired.

For each critic a user can press "thumbs up" or "thumbs down" buttons situated on the right side of the list. By doing this he or she can send us a feedback of whether the critic was helpful to him (her) or not. Optionally a textual description can be also provided with the feedback. By collecting this data we are able to quickly identify issues in the quality rules and detect which rules are not welcomed by the developers.

QualityAssistant also provides plugins for the Inspector and Spotter tools. Inspector is a tool that allows developers to inspect objects in Pharo [10]. Objects may have different representations, and a user can select an object from a presentation and inspect it. This enables continuous inspection of objects which is useful during software development.In Pharo everything is an object, including methods and classes. This allowed us to create a special inspector presentation for method and class objects which displays critics about them. Moreover developers can select and inspect a critic to obtain more information about it.

Spotter is a unified search interface that spans many scopes [11]. One can use spotter to search for tools, or to search for a class, dive into a class to see its components (*e.g.,* methods) and perform a search among these components. As Spotter allows various extensions, QualityAssistant adds critics as components of a class or method. This way if a developer dives into a method or a class, he encounters related critics.

V. QualityAssistant Survey

Almost two months after QualityAssistant was integrated into the development version of Pharo, we conducted a second survey to understand how developers were using it. The survey had similar structure to the one described in Section III. 29 developers participated in this survey, and their development experience summary is shown in Figure 7. In this survey most of the participants have from 1 to 3 years of development experience both in Pharo and Smalltalk. Also there are almost no participants with less than 1 year of experience. The number of participants from academia is three times as large as the number of participants from industry.

In this survey we asked developers to evaluate usefulness of all three plugins of QualityAssistant. Participants had to grade the main coding browser plugin on a 7-point Likert scale: 1) very useful 2) useful 3) sometimes useful 4) not influential 5) sometimes disturbing 6) disturbing 7) very disturbing. For the inspector and spotter plugins we used similar 5-point Likert scale that did not include extreme options 1) and 7). The results of these questions are shown in Figure 5.

As opposed to Critic Browser, developers are using QualityAssistant all the time that they spend while programming. According to the responses about 75% of the developers find the main coding browser plugin useful in some way. More than a half of participants does not know about the inspector and spotter plugins, but the majority of those who know tend to find them useful to some extent.

We have also selected six features that the coding browser plugin provides:

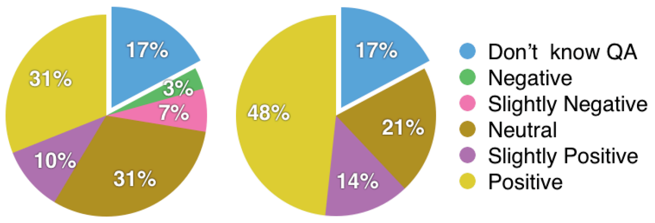
1) Display the description of a rule.
2) Highlight the part of the code that a critic refers to.
3) View the diff of a proposed automated critic resolution.
4) Apply an automated critic resolution.
5) Ban a critic for the entity it refers to.
6) Ban a critic for a more general scope.

We asked participants whether they are familiar with each feature. In case they were familiar we asked them to specify how often they were using a feature out of 5-point Likert scale: 1) always 2) often 3) sometimes 4) rarely 5) never. If the participants did not know about a feature we asked them whether they plan to use it. As can be seen in Figure 6, many participants do not know about QualityAssistant's features, but are planning to use them. The only two features that some developers don't want to use are related to banning rules, which shows that we have to pay more attention while designing this kind of features.

Integration of QualityAssistant into the development version of Pharo facilitated changes in SmallLint rules. We decided to choose two major diverse changes and ask whether developers like them. For the first change we selected a complete removal of *"Probably missing yourself"* rule. The rule checked whether a certain method is called in the end of method cascade, because in some cases it could be useful. In reality however this rule was generating many false–positive critics and was distracting developers more than aiding them. For the second change we have selected an introduction of a new rule that enforces to use `ifNotEmpty:` and `ifNotNil:` methods instead of `ifNotEmptyDo:` and `ifNotNilDo:` respectively. This rule describes API usage, and originated because methods `ifNotEmptyDo:` and `ifNotNilDo:` should not be used, but are kept in the system for compatibility reasons. For each change we have asked developers to rate the change on the 5-point Likert scale: 1) positive 2) slightly positive 3) neutral 4) slightly negative 5) negative. As can be seen in Figure 8, developers are mostly positive about the changes made to the SmallLint rules. This shows that QualityAssistant indeed drew attention to the quality rules and positively influenced changes.

VI. Survey Analysis and Future Work

Both informal observation and surveys show that QualityAssistant is well accepted by developers. About 75% of them find the main coding browser plugin useful in some way. This also amounts to more than 90% from the developers that are aware



(a) Removal of *"missing yourself"* rule (b) Addition of *"use* `ifNotEmpty:` *method instead of* `ifNotEmptyDo:`*"* rule

Fig. 8: Reaction to the changes in rules

of QualityAssistant. This is a very high positive feedback ratio which shows that developers like QualityAssistant.

We still need to verify whether the tools like QualityAssistant really impact the quality of the developed software more than the tools like Critic Browser.

We have almost integrated a usage recorder that will record developers' interaction with QualityAssistant and send the collected data to our server. This functionality will be activated only if the developer agrees to share the usage data with us. We plan to use the data for further investigation on whether live-feedback integrated static analysis tools are more useful than the external ones. Also we plan to use this data to improve QualityAssistant and SmallLint rules themselves.

We also plan to perform a controlled experiment to see how QualityAssistant performs code quality tasks in comparison with Critic Browser.

Acknowledgments

I want to thank Michele Lanza and Andrea Mocci for providing a valuable advice during this research.

References

[1] M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, Sep. 1976.
[2] J. Cohen, *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.
[3] A. G. Bardas, "Static code analysis," *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99 – 107, 2010.
[4] P. Louridas, "Static code analysis," *Software, IEEE*, vol. 23, no. 4, pp. 58 – 61, July 2006.
[5] N. Ayewah, W. Pugh, D. Hovemeyer, D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22 – 29, Sept 2008.
[6] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of 37th IEEE/ACM International Conference on Software Engineering*. IEEE, 2015, pp. 598 – 608.
[7] A. P. Black, O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by example*. Lulu, 2010.
[8] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theor. Pract. Object Syst.*, vol. 3, no. 4, pp. 253 – 263, Oct. 1997.
[9] A. N. Oppenheim, *Questionnaire design, interviewing and attitude measurement*. Bloomsbury Publishing, 2000.
[10] A. Chis, O. Nierstrasz, and T. Grba, "The moldable inspector: a framework for domain-specific object inspection," in *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, 2014.
[11] A. Syrel1, A. Chis, T. Girba, J. Kubelka, O. Nierstrasz, and S. Reichhart, "Spotter: Towards a unified search interface in ides," in *Proceedings of the Companion Publication of the 2015 ACM SIG- PLAN Conference on Systems, Programming, and Applications: Software for Humanity*. ACM, 2015, p. to be published.