When QualityAssistant Meets Pharo

Enforced Code Critiques Motivate More Valuable Rules

Yuriy Tymchuk, Mohammad Ghafari, Oscar Nierstrasz
SCG @ Institute of Informatics - University of Bern, Switzerland

Abstract

Static analysis tools can aid in software quality assessment, but are rarely used by software developers. Poor usage of quality analysis tools not only means missed opportunities for the quality of software systems, but also results in little feedback, which in turn slows the improvements of the quality rules themselves.

We introduced a set of intrusive quality plugins and integrated them into the Pharo IDE. This not only triggered a feedback loop that led to improvements of the existing rules, but also encouraged removal of some rules and integration of new ones. Our analysis of changes to the rules suggests that precise rules capturing a domain-specific logic are more valuable than general ones.

1. Introduction

Quality of software systems is based not only on their functionality, but also on maintainability aspects [3]. The latter can be ensured by manually reviewing source code or can be aided by static analysis tools. Such tools analyze source code and detect suspicious patterns, relations or metrics. However, just like any piece of software, quality analysis tools may have defects, miss important features or provide disturbing functionality that obstructs the effectiveness of these tools. While even detailed planning and rigorous testing cannot ensure a perfect software project, it is common practice to collect defect reports and usage experience feedback from users to improve the future versions of a project. Usually feedback loops and changes in quality analysis tools are not common. Previous studies have shown that developers need a way to customize how the tools work, or to change which rules are used for source code analysis [8, 15]. Moreover, many developers are not aware of the existence of quality analysis tools, and those who know about the tools prefer to have them run automatically in the background [15].

Our main goal was to foster the usage of quality analysis tools during the development of the fifth version of the Pharo project[1], an object-oriented programming language and a full-fledged integrated development environment (IDE) [2]. SmallLint [11] is a quality analysis engine that is shipped with the IDE since Pharo's initial release a few years ago. At the beginning of the development cycle SmallLint had 120 rules defining quality concerns in 6 categories such as *bugs*, *style*, *optimization*, *etc.* A violation of a rule by some piece of code is called a *critic*[2].

From the first release of Pharo and until the beginning of the fifth development cycle there was only one quality analysis tool with a graphical user interface, namely CriticBrowser. We distinguish *on-demand* tools, which must be explicitly activated by the developer, from *intrusive* tools that continuously perform quality checks and present critics to developers. CriticBrowser is an on-demand tool that validates a set of source code packages using a set of SmallLint rules to obtain the critics. We introduced QualityAssistant: a series of plugins for development tools that use the available SmallLint rules to provide an intrusive feedback about the quality of the source code that is being browsed. In the previous study we have analyzed the reception of CriticBrowser and QualityAssistant by Pharo developers [14]. The results have shown that the on-demand tool CriticBrowser was rarely used, while QualityAssistant, which intrusively reports critics about browsed code, obtained almost completely positive feedback from the developers.

Positive acceptance of QualityAssistant in combination with ease of updating SmallLint led to many changes in the quality rules. In the latest development cycle, over 60 SmallLint-related bug reports, feature requests and enhancement suggestions were opened which is twice as many as the average in previous years.

In this paper we summarize the changes introduced to the rules after QualityAssistant was integrated into the Pharo

CONF 'yy Month d–d, 20yy, City, ST, Country

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00
DOI: http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] `http://pharo.org/`

[2] Although the term "critique" would make more sense in English, tools in the image have adopted the term "critic".

IDE and demonstrate how promotion of a quality analysis tool together with a feedback loop can help in shaping the quality rules themselves. We also analyze 3 rules that were removed from the system and 15 new rules integrated into Pharo or related projects. The relationships between them show that developers prefer rules that are easy to understand and capture important violations to those that express general programming practices.

2. Changes to the Rules

QualityAssistant is a set of plugins for tools that are used during the development process in Pharo. The main goal of the plugins is to provide contemporary critics about the code on which a developer is working. One of the most important plugins resides in the code browser: a tool where developers browse and edit code. This way developers are always aware of the critics related to the code that they are working on, which encourages them to report false critics and express their ideas about enhancements.

According to our investigations the changes to the rules after QualityAssistant was integrated can be naturally divided into 3 categories: bug fixes and usability improvements, removal of the rules or a part of their functionality, and creation of new rules. In this section we describe the most interesting use cases in each category.

2.1 Bug Fixes and Usability Improvements

QualityAssistant greatly increased the number of critics that Pharo developers encounter during programming sessions. This provoked complaints and bug reports about the critics that provide false information. For example one rule checked whether the category of a method is the same as the category of the overridden method from a superclass. However it also detected cases where a method has a category while the overridden method is uncategorized. In other words the rule suggested to remove the criticized method from its category to match the status of the overridden method, which is nonsense and violates another rule that checks whether all methods are categorized.

Some bugs had more severe consequences. For example a rule named "*Modifies collection while iterating over it*" modified abstract syntax trees during source code validation. This is completely unexpected and unacceptable behavior as the validation process destroyed parts of the system, much like a virus. The bug was detected only after the integration of QualityAssistant using which developers noticed a strange behavior of the methods that they worked with. As the bug was in a SmallLint rule — it also potentially affected all methods validated by CriticBrowser. The bug was not detected before although the rule is not new, which suggests that CriticBrowser was seldom used.

Another group of rule changes is related to usability improvements. Most of them are caused by the inability of a rule to explain a violation. For example the rule for detecting usage of a "soon-to-be deprecated API" detected a method, but did not specify which part exactly violates the rule. Another rule suggests replacing `detect:ifNone:` with `anySatisfy:`, but also reports a critic if `contains:` is used. This happens because at some point the rule's functionality was updated, but its description was left unchanged. While many usability problems were caused by poorly developed rules, there were some issues related to a completely wrong design of a rule. For example one rule detected whether an abstract class has references. While a developer gets a feedback about the class, the exact piece of code that references it is unknown. This issue was resolved by shifting validation from detecting abstract classes that are referenced to detecting methods that reference an abstract class and highlighting the source code where the abstract class is accessed.

In some cases we went even farther than just enhancing the explanations of critics. For example one rule detected a class that has all subclasses with the same-signature methods, but does not itself define an abstract method with that signature. The rule did not specify which method is missing in the class. Not only did we add the method name to the critic explanation, but also we created an action that automatically creates the required abstract method. Automated resolutions of critics contribute to the usability of the rules by making them clearly actionable, as well as offering better explanations.

In total over the course of development of Pharo 5 around 20 integration requests were related to improvements in functionality or usability of quality rules.

2.2 Rule Removals

During the evolution of Pharo 5, three quality rules were completely removed. All of them had the same properties, namely they brought attention to common beginner mistakes, but were annoying to experienced developers. Additionally we allowed developers to react to the critics by sending positive or negative feedback. The critics of all these rules received a large amount of negative feedback. Next we provide the description of each use case.

The most hated rule checked whether a `yourself` message is present at the end of a message cascade [6]. The rule is very important if a person does not know how a cascade works, because one may expect to obtain the receiver of a cascade as a result while a cascade actually returns the result of the last message. If a developer is aware of this, there is little benefit in being reminded that there is a cascade without a `yourself` message at the end. There are many cases where this message is not necessary. Moreover, one may want to actually obtain the result of the last message in the cascade instead of the receiver. In this case `yourself` has to be omitted on purpose.

The second rule detected whether the messages like `ifTrue:` have a block as their argument. This is another useful rule for beginners that annoys experienced develop-

ers. None of the users of Pharo 5 found it useful, and the rule reported critics only about special cases that worked well without blocks. For example:

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

is a perfectly valid piece of code that is easier to read compared to the version with blocks, and executes faster as there is no need to unwind block contents.

The last rule detected methods that reference an abstract class. This rule was previously modified and so is also described in the previous subsection. In essence the rule raises the programmer's awareness of possible instantiations of an abstract class which can lead to a potential invocation of an abstract method. However this also includes the usage of utility methods on the class-side. Moreover abstract classes often provide factory methods [5] that return their subclasses. For example `UIManager default` will return an instance of a concrete `UIManager` subclass that is default for the current setup. It is also not possible to only focus on the `new` messages that are sent to the abstract classes, for example `String new` creates an instance of a concrete `ByteString` class which is a subclass of an abstract `String` class.

In all cases the probability of a critic being an actual issue was much lower than the negative impact that the issue would cause if it was present. However all the rules from this subsection had an educational emphasis which means that they may be useful for a teaching configuration of the system.

2.3 Rule Addition

As developers became more aware of the critics in their code, new rules were introduced to inform developers about the violations. Most of the rules were requested by the Pharo community or the developers of different frameworks and we ourselves implemented many of them. Originally all the rules were packaged together with the SmallLint engine. We tried to put each new rule in the same package that contains the code related to that rule. For example if a rule describes how a testing framework should be used, we package it with that framework: this way if someone uses the framework he or she will also get critiques from the dedicated rules.

Only a few rules were placed in the SmallLint package. Two of them are rules related to Pharo core functionality. One of them suggests to use `ifNil:` and `ifEmpty:` instead of `ifNilDo:` and `ifEmptyDo:` as they will be deprecated soon. Another one suggests to replace `Smalltalk at:` with `Smalltalk globals at:`, because of changes in the core API. The other group of rules captures new architectural constraints of SmallLint rule classes and so was packaged in the SmallLint package.

All the other rules are packaged together with their related projects. For example a rule that warns about a usage of special `BoxedFloat64` and `SmallFloat64` classes was shipped with the Kernel package. Another rule that suggests to use `assert: a equals: b` instead of `assert: a = b` was added to the SUnit package. First of all this enhances modularity of the system: one can simply unload the SUnit package and load another testing framework. In this case the SUnit-specific rules will be uninstalled together with the package. Additionally this packaging strategy encourages the maintainers of the package to also maintain the rules that are related to their code.

Rules were introduced to three projects that are being developed in parallel with Pharo and have their own repositories. Two of these projects, Rubric and Roassal [1], required rules to check the order of method invocations. Both projects make use of builder pattern [5] where the order in which the builder methods are invoked impacts the final result. For this purpose a special type of rule was introduced that allowed developers to easily specify the required order.

Rules for the third project, *Glamorous Toolkit*[3], were different by their nature. One of the rules detects classes that have extension methods coming from the toolkit but do not define special methods that are used to show examples about these classes. Other rules relate to performance issues that could arise during the definition of the extensions. In particular computation of the values can be delayed by passing expressions wrapped in blocks and allowing a builder to lazily evaluate the block as late as possible.

The 15 new rules can be grouped into five distinct categories based on their nature, impact of their critics, and ways to resolve them.

Migration rules describe a transformation from one API to another and can automatically rewrite code. For example usage of `ifNil:` instead of `ifNilDo:`. If violations of these rules are ignored, in the future the code may fail because of the removal of the old API.

Private access rules warn about usage of functionalities that are not meant for public access. For example `SmallFloat64` is a system-specific class that is present to ensure good performance of floating point calculations on 64-bit systems, but it should not be used directly in the code. Violations of these rules can be ignored only under very special circumstances, such as deliberate low-level programming, but this may lead to unexpected errors, non-portable code, or even issues in the future as a private functionality evolves more frequently.

Invocation order rules detect if the order of message sends makes sense. For example if the `edges:` message is sent to a graph builder before the `nodes:` method is used to specify nodes, the edges will not be initialized. If critics of these rules are ignored the desired result will not be obtained.

Class structure rules capture design guidelines that should be followed while subclassing or extending classes. They

[3] `http://gtoolkit.org`

check whether certain methods are overridden, or that particular methods are present if certain conditions are met. The rule that checks if the `hash` method is overridden together with the "=" method belongs to this category. If left unchanged the violations will either result in an unexpected functionality, or cause a rejection of integration as the project's design guidelines are not followed.

Lazy evaluation rules in our case were represented by a single pair of rules in Glamorous Toolkit. They detect expressions that could be evaluated lazily to exploit the design of Glamorous Toolkit. Violations of these rules will result in poor performance and will complicate exception handling, if left unchanged.

All of the added rules target exact violations such as a wrong API usage, violation of design conventions or inefficient code. The violations are related to the project where the rules belong rather than some general OOP practices. Moreover, most of the violations will certainly result in defects and so cannot be ignored.

Another important aspect worth mentioning is the implementation of the rules based on their type. Migration rules can be easily expressed with the refactoring browser [12] rewrite engine. For example use of the old SUnit API can be easily detected with the expression:

```
self should: [ ``@object ]
```

and changed to the new API with:

```
self assert: ``@object
```

Private access rules can simply detect access to entities with a certain annotations, which is implemented in many languages through access modifiers. To specify invocation order a special DSL like *usage contracts* [9] can be used. Class structure can be easily checked by directly manipulating class objects *i.e.,* by obtaining a list of methods defined in a class, and validating whether they adhere to certain requirements. To support lazy evaluation rules we worked with raw AST nodes: we identified builder message nodes that should receive blocks as arguments and verified whether the argument nodes are literals or blocks. If the arguments are represented buy other kinds of expression nodes we generate a critic. We believe that there is no single approach or DSL to define quality rules but rather many different ones that target a certain type of violation and provide a different level of actionability.

2.4 Structural Changes

The introduction of QualityAssistant triggered some minor structural changes in SmallLint rules. Some duplicated functionality was removed from certain rules. For example one rule checked whether `= 0` is used instead of `isZero` and `= nil` is used instead of `isNil`. But another rule also included the latter case and additionally checked if `~= nil` is used instead of `notNil`. Duplication became easy to detect by shifting the critics from the rule-centric view in the Critic Browser to an entity-centric view in the QualityAssistant. For example previously a developer was using a browser that was displaying violations per each rule making it complicated to notice among many critiques that the same violation is reported by two rules. On the other hand if a developer browses a method that contains `= nil`, critics from both rules inform that `isNil` should be used instead, making the duplication evident.

Some rules were split into multiple ones to allow better filtering or performance decisions. For example a rule that checks whether a sent message is not implemented was split into two: a fast one that checks whether a message sent to self or super is not implemented in the hierarchy, and a slow one detecting whether a method with the selector of a message is implemented anywhere in the system. This way the slow rule may be excluded from the live feedback of QualityAssistant to avoid lengthy delays during development process, while the fast rule will still detect a subset of critiques.

3. Related Work

Sadowski *et al.* [13] reported a study with a similar setup by introducing the *Tricorder* quality analysis tool into Google's development process. Tricorder is also an intrusive tool that reports critics during a pre-commit review [10]. However, in their experiment Tricorder was the first tool that was introduced in the development process while we are able to make a comparison with CriticBrowser, which was already present in Pharo before. While the authors focus on the principles of the feedback loop and the rules were refined to minimize negative feedback, they do not provide information about the changes themselves.

Hora *et al.* [7] conducted an experiment to determine whether domain specific rules detect more bugs than generic ones. They analyzed critics of the historical data of a project, and correlated them with bugs from the project's bug tracker. The experiment has shown that more bugs were related to domain specific rules in comparison with generic rules. Similarly our results show higher values of domain-specific rules, as the removed ones are generic object-oriented rules, while the added ones are related to a specific domain.

Lozano *et al.* [9] developed the *uContracts* language to specify structural regularities in software systems. The authors motivate their decisions with a previous experience with the *SOUL* [4] language: developers were reluctant to define structural regularities in SOUL as they had to learn a new programming language. For this reason the authors analyzed the most common structural regularities and designed uContracts semantics to be very similar to the programing language that developers use. Many rules from our use case conform to the structural regularity types defined by uContracts. This correlation can be investigated even further

by encouraging developers to use uContract to develop new rules.

Yamashita and Moonen [15] surveyed professional software developers to identify the current status and requirements of static quality analysis. The authors discovered that almost one third of the surveyed developers did not know about the existence of quality critics. We addressed this issue with QualityAssistant, which in turn triggered numerous changes to the rules. One of the highly-ranked requirements was to be able to "define and customize detection strategies". SmallLint provides a possibility to add new rules and in our case more than a dozen rules were added. However most of the rules were implemented by us following the requests of developers. This leaves us with a question on how to help developers to create rules themselves.

4. Conclusions and Future Work

We have introduced a set of intrusive plugins called QualityAssistant into the IDE used by Pharo developers. The plugins educated developers about the the existence of quality rules and issues in their code. At the same time QualityAssistant started a feedback loop that triggered many changes to SmallLint rules.

The analysis of the changes made after integration of QualityAssistant shows that many important bugs in rules were detected and fixed. Unproductive or annoying rules were removed, while new helpful rules were added. The analysis of the added and removed rules helps us to identify the features that are important for developers in the rules, namely:

1. clear explanation of a critic that exposes the source of violation;
2. suggestion of a solution or automated resolution of an issue;
3. high impact of the critic *i.e.,* critics that detect guaranteed bugs are valued more than the ones that warn about hypothetical problems.

Moreover the new rules introduced into the system capture domain-specific properties of the projects that they represent. This suggests two distinct types of rules:

Internal rules are specific to a team, company or community and focus more on the general questions. These rules may define style, focus on common metrics like number of methods in a class or describe an architecture of a project. Their value comes from capturing agreements that are already present in the community. To change them one has to negotiate the change of the principles in the community.

External rules come together with a library or framework and act as documentation. This type of rule supports migration to a new version of API and detects improper or inefficient usage of a library. The value of these rules comes from automatically providing crucial information about possible bugs and their solutions as early as possible. Maintainers of projects should be in charge of changing the related rules similarly to how documentation is changed.

In the near future we expect to obtain feedback about the rules from even more users which should allow us to enhance the understanding of the needs regarding quality rules. This includes the feedback about the new rules as at the moment we cannot be sure if all of them are well-received.

Another important question to investigate is why developers are not implementing the rules themselves, as most of the new rules were developed by us. We conjecture that first of all developers are not implementing their own rules because they are not used to it. We also believe that the main obstacle is the means to define the rules themselves. Usually they are very complicated and each new way to define rules tries to be better than all the other ones. By analyzing the rules we created, we can see that different approaches may be suitable for different kinds of rules. Investigating more rules and the difficulties that developers encounter while defining them will help us to draw more concrete conclusions.

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] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013. ISBN 978-3-9523341-6-4.

[2] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009. ISBN 978-3-9523341-4-0. URL `http://pharobyexample.org`.

[3] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of 2nd International Conference on Software Engineering*, pages 592 – 605. IEEE Computer Society Press, 1976.

[4] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The soul tool suite for querying programs in symbiosis with eclipse. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 71–80, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. doi: 10.1145/2093157.2093168. URL `http://doi.acm.org/10.1145/2093157.2093168`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995. ISBN 978-0201633610.

[6] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley,

Reading, Mass., May 1983. ISBN 0-201-13688-0. URL http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[7] A. Hora, N. Anquetil, S. Ducasse, and S. Allier. Domain specific warnings: Are they any better? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 441–450, Sept 2012. doi: 10.1109/ICSM.2012.6405305.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681. IEEE Press, 2013. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486877.

[9] A. Lozano, K. Mens, and A. Kellens. Usage contracts: Offering immediate feedback on violations of structural source-code regularities. *Science of Computer Programming*, 105:73 – 91, 2015. ISSN 0167-6423. doi: 10.1016/j.scico.2015.01.004. URL http://www.sciencedirect.com/science/article/pii/S016764231500012X.

[10] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491444. URL http://doi.acm.org/10.1145/2491411.2491444.

[11] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[12] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[13] 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*, ICSE '15, pages 598–608, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818828.

[14] 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. URL http://yuriy.tymch.uk/papers/benevol15.pdf.

[15] A. Yamashita and L. Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, Oct 2013. doi: 10.1109/WCRE.2013.6671299.