Treating Software Quality as a First-Class Entity

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

***Abstract*—Quality is a crucial property of any software system and consists of many aspects. On the one hand, quality measures how well a piece of software satisfies its functional requirements. On the other hand, it captures how easy it is to understand, test and modify a software system. While functional requirements are provided by the product owner, maintainability of software is often underestimated. Currently software quality is either assessed by experts, or presented as a list of rule violations reported by some kind of static analyzer. Both these approaches are working with a sense of quality outside of the software itself.**

We envision quality as a first-class entity of a software system, a concept that similarly to the functionality is persistent within the software itself. We believe that each entity or a group of software entities should be able to tell about its quality, reasons of bad smells and ways to resolve them. This concept will allow to build quality aware tools for each step of the software development lifecycle.

On our way to the concept of quality as a first class entity, we have created a code review approach where software quality is the main concern. A reviewer makes decisions and takes actions based on the quality of the reviewed system. We plan to continue our research by integrating advanced quality rules into our tools and devising new approaches to represent quality and integrate it into everyday workflow. We started to develop a layer on top of a software model responsible for the quality feedback and allowing to develop quality-aware IDE plugins.

I. Introduction

People always tried to assign some meaning of quality to things that matter to them. Software is not an exception. Boehm *et al.* [1] define two main categories of software quality: *Usability* and *Maintainability*. The former concerns how a software complies to its functional requirements. The latter reflects how the software itself is designed, how complex is understanding, testing and modifying the software. We focus on maintainability, which is directly connected with software architecture design and coding style. Well-designed software requires less effort to maintain, while common good coding style can improve the readability of code.

Designing high-quality software requires decision making at different levels of abstraction: From style used in code writing, to architecture of the whole software system and its components. For example, Riel [2] describes heuristics that should be followed to achieve a good quality object-oriented design. Gamma *et al.* [3] provide most frequently used design patterns in object-oriented languages with examples in C++ and Smalltalk. Kent Beck [4] describes the best coding practices specific for Smalltalk programming language.

Software development is a complex process, and even good quality practices cannot guarantee that through all its life cycle the project will not accumulate bad design decisions. It is important to be able to detect and improve low quality parts of a system. For this purpose Martin Fowler [5] describes bad code smells and how to solve them through refactoring techniques. A person possessing knowledge of these guidelines can perform a review of the software system, by manually assessing its quality and improving the parts that in his opinion have low quality. This practice is formally called a *code review*. Manually reviewing code is time-consuming [6]. To reduce a reviewers' effort static analysis tools can be used to automatically detect bad patterns in source code or to validate a system against some metrics. For example, Lanza and Marinescu [7] define rules for composite metrics that can be used to detect complex design smells by aggregating values of simpler metrics.

Currently software quality is either assessed by the reviewer who follows the guidelines, or by a tool which generates a list of issues. In both cases the quality assessment happens outside of the software system and remains outside of it. Our goal is to create a rich model of a quality concept that can be used by various development tools. We believe that the quality model should 1) provide cost and value information about critics 2) allow adaptation rules must to a given project 3) is aware of software scopes and development lifecycle.

II. Related Work

State of the art of software maintainability can be split in two categories. One of them consists of tools and algorithms designed to help *detecting quality issues*. Some of them automatically identify faulty software parts, others provide an alternative overview of the software system in order to improve comprehension. In the second category we assign approaches that try to embed automated issue detections into the code review process.

A. Detecting Quality Issues

Common bad programming practices can be detected automatically with *static code analysis* [8], [9]. At the moment there are many static code analysis tools focusing on different languages and different categories of quality issues. Lint [10] is one of the pioneering tools and it can be used to check C programs. FindBugs [11] is the most popular static analysis tool for Java. CheckStyle[1] and PMD[2] are also often used to find defects in Java code, although the latter one has a support for different programming languages. All these tools have common features. They run on static data, either source code or compiled byte code. They have predefined rules that are used to find defects. Most tools categorize quality issues.

[1]http://checkstyle.sourceforge.net
[2]http://pmd.sourceforge.net

Some tools assign severity value to each issue, and allow to extend a set of quality rules by manually entering new ones. Johnson *et al.* [12] investigated why software developers do not use static analysis tools to find bugs. They surveyed 20 developers that used FindBugs, Lint, CheckStyle, PMD and other similar tools. All survey participants felt that use of static analysis tools is beneficial, but false positives and the way in which the warnings are presented discouraged usage of the tools.

Another category of analyzers relies on software metrics, thresholds and fuzzy logic to detect more complex issues in software architecture. Marinescu [13] implemented the IPLASMA tool which uses a metric-based approach to detect code smells. Its detection strategies capture deviations from good design principles and aggregate metrics and compare their values against absolute and relative thresholds. Moha *et al.* [14] present DECOR, a methodology that defines all the steps needed to specify and detect code design smells. They also present DETEX, a tool which implements the DECOR approach. Khomh *et al.* [15] developed a bayesian approach which calculates the probability of an entity violating a certain design rule. Despite giving a better overview of the status of the underlying system, these rules are still presented as a list of issues, which does not provide a good way of understanding which parts of the system have more issues and need attention in the first place. These analyzers rely on a deviation from the average values in a system, whereas we believe that developers should be able to define the thresholds themselves.

Caserta and Zendra [16] provide an exhaustive survey of source code visualization techniques. These techniques aim to increase the awareness of a person reviewing the code, by mapping different metrics on a visualization. While these tools indeed improve understanding of the underlying code, they miss an automated analyzer, that could tell a developer some information about the quality of the system and the parts that compose it.

Static code analysis tools are often embedded into Integrated Development Environments (IDEs). This allows to inform the developer about quality issues in the same environment that he or she uses to program. Some IDEs provide live feedback about coding practices violations as-you-type. Popular Java IDEs Eclipse[3] and NetBeans[4] use FindBugs reports. IntelliJ IDEA[5] uses a custom defect detection subsystem. Some tools also allow automatic fixes for certain issues. These tools are not used while the developer reviews his changes before committing a new version of code. In the case of the code quality concept, a developer could be provided not only with quantitative changes, but also with some estimation whether they improve general quality, or decrease it.

Static analysis can be also used during continuous integration (CI) [17]: After each integration of a change, the software system is automatically built and validated with respect to different criteria. One of the most popular systems used for this practice is SonarQube [18]. It defines seven axis of quality as follows: *Architecture and Design*, *Duplications*, *Coding Rules*, *Comments*, *Unit tests*, *Potential bugs* and *Complexity*.

[3]http://www.eclipse.org

[4]https://netbeans.org

[5]https://www.jetbrains.com/idea/

B. Reviewing Code

Static code analysis can aid in detection of low quality software parts. Another approach used to ensure and improve the quality of a project is *code review* [19]. During this activity developers are reading the code that does not belong to them in order to familiarize with it and find possible defects. Modern code review is usually performed on a patch before it is integrated into the system, in order to find and fix faulty parts on the early stage.

While code review provides good results, Bacchelli and Bird [20] discovered that reviewers spend a lot of time to detect trivial issues instead of focusing on the important ones. These trivial issues are usually related with the coding guidelines, and possibly can be detected automatically to save the time of reviewers. During a code review session, tools like Review Bot [21] use static analysis reports to improve reviewers experience by pointing out parts that violate some rules. However as stated before, there is no way to understand whether a change improved something, or if it contains any violations that are strictly forbidden to be integrated.

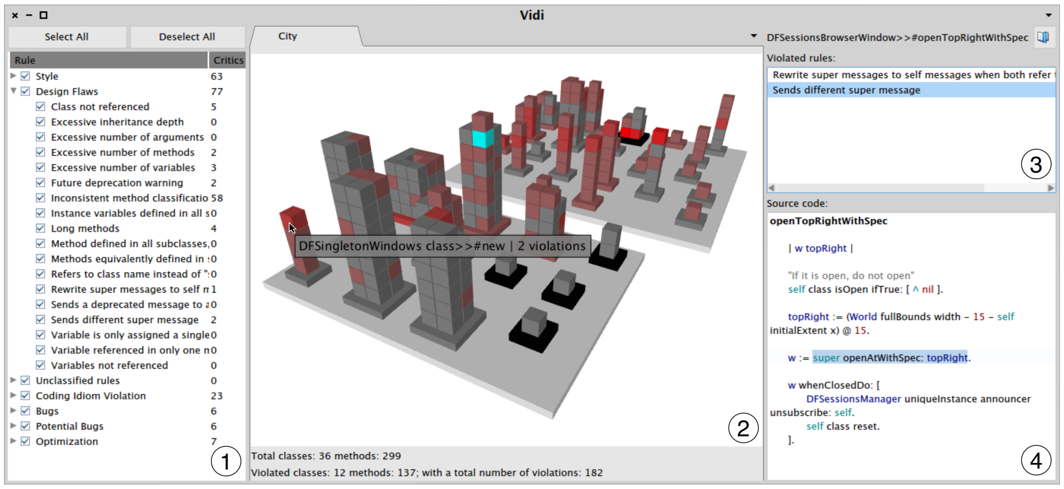
The Tricorder project adds quality reports to the code review tool used by Google [22]. Reviewers can mark quality critics as false positives and provide an optional textual feedback that will be sent to the quality rule developer. Usage data is collected and the Tricoder team pursues the goal to maintain the amount of false positives under 10%. Issues that are postponed for future resolution are also treated as false positives. And while the low percentage of ignored critics results in high tool acceptance, some of the quality rules are removed as they are not trivial, and require longer time to resolve them.

III. Modeling Quality of Software Systems

We want to address the limitations of current approaches to define, detect and ensure software quality. We envision quality as a first-class entity of a software system. This concept allows us to focus on quality itself as opposed pragmatic rule violations, and it is required to compare entities either in one project or across projects, localize parts with higher rating of severity, or summarize the quality of an entire system. In this way, we will be able to build quality-aware tools that provide better understanding of a software system. These tools should be used throughout the entire development cycle and provide an aid in writing and reviewing code.

A. ViDI

As a first milestone towards a full fledged concept of software quality we developed an approach called Visual Design Inspection (ViDI) [23]. It augments code review with software quality evaluation, and more general design assessment, as a first class citizen and as the core concern of code review. It uses visualization to drive the quality assessment of the



Rule	Critics
☑ Style	63
☑ Design Flaws	77
☑ Class not referenced	5
☑ Excessive inheritance depth	0
☑ Excessive number of arguments	0
☑ Excessive number of methods	2
☑ Excessive number of variables	3
☑ Future deprecation warning	2
☑ Inconsistent method classificatio	58
☑ Instance variables defined in all s	0
☑ Long methods	4
☑ Method defined in all subclasses,	0
☑ Methods equivalently defined in :	0
☑ Refers to class name instead of ":	0
☑ Rewrite super messages to self m	1
☑ Sends a deprecated message to a	0
☑ Sends different super message	2
☑ Variable is only assigned a single	0
☑ Variable referenced in only one n	0
☑ Variables not referenced	0
☑ Unclassified rules	0
☑ Coding Idiom Violation	23
☑ Bugs	6
☑ Potential Bugs	6
☑ Optimization	7

Total classes: 36 methods: 299

Violated classes: 12 methods: 137; with a total number of violations: 182

DFSessionsBrowserWindow>>#openTopRightWithSpec

Violated rules:

Rewrite super messages to self messages when both refer

Sends different super message

Source code:

```
openTopRightWithSpec

    | w topRight |

    "If it is open, do not open"
    self class isOpen ifTrue: [ ^ nil ].

    topRight := (World fullBounds width - 15 - self
initialExtent x) @ 15.

    w := super openAtWithSpec: topRight.

    w whenClosedDo: [
        DFSessionsManager uniqueInstance announcer
unsubscribe: self.
        self class reset.
    ].
```

Fig. 1: ViDI main window, composed of 1) quality rules pane; 2) system overview pane; 3) critics of the selected entity; 4) source code of selected entity.

reviewed system, exploiting data obtained through static code analysis. Alongside the approach we have also built a tool with the same name [24].

ViDI is implemented in *Pharo*[6], a modern Smalltalk-inspired programming language and full-fledged object-oriented development environment. ViDI is available as an MIT-licensed free software at http://vidi.inf.usi.ch. ViDI uses SmallLint [25] to support quality analysis and obtain reports about issues concerning coding style and design heuristics. The version of SmallLint that we use has 115 rules organized into 7 different categories, from simple style checks to more complex design flaws. Rules concern specific code entities like packages, classes or methods. A violation of a rule by an entity, is called a *critic* about that entity. For example, elements with no critics are colored in gray. The higher the amount of critics, the brighter is the red coloring of the entity.

The reviewed system is presented in a visual environment augmented with automatically generated quality reports. The environment is self-contained: The reviewer can navigate, inspect and improve the system from inside ViDI. As a system can be changed during the review session, ViDI automatically re-runs the static analysis tool, to keep the reviewer updated about the current system state. The main window of ViDI is depicted in Figure 1.

While reviewing a software system with ViDI, the reviewer is presented with a visual representation of the system, as well as the summary about size of the reviewed system and the number of present issues. In our first prototype we used a city-based code visualization [26], [27], depicting classes as platforms on which their methods are represented as blocks and stacked together forming a visual representation of a building. ViDI imbues the visualization with the quality critics making it easy to understand where are the bad parts, and how severe the critics are. The reviewer can inspect each single entity to see the detailed critic descriptions, and the source code. Some of the critics can be resolved automatically, by code transformation, or the reviewer can manually edit the code critics will be re-evaluated to take into account the latest changes. The reviewer can also filter which critics he wants to see and which ones will be hidden either by a quality rule or by a group of rules.

B. Ongoing Work and Future Work

We have performed a couple of ViDI use-case studies in our group. The analysis of them has shown that the current static analysis tools used by ViDI are lacking important features needed to detect real issues.

Fuzzy rules. One of the limitations of current tools is the boolean nature of validation results. An entity can either violate a rule or not. To solve this we are currently investigating the use of *fuzzy logic* [28] to implement rules. While boolean variables may only be true or false, fuzzy logic variables have a truth value that ranges between 0 and 1.

Advanced rules. Some parts of a system may have different thresholds, require specific rules, or should not be validated with certain rules. We believe that rules should support parameters and localization for parts of a system. We plan to add rules based on the detection strategies proposed by Marinescu [13] in order to be able to identify issues on a global scope. Another kind of rules that is missing are architectural and API usage rules. During its lifetime a project is designed with some architecture, often they are described in a plain text and validated manually. Having this design encoded as rules and distributed with the source of a project will not only save time of integrators, but will also help developers to introduce architectural disorders on the early stage. Finally, having a high amount of diverse rules requires a dedicated approach of providing a feedback about the system.

Realtime feedback. Recently we developed a tool called QualityAssistant. As opposed to ViDI, it does not require a developer to explicitly open a separate window in order to see the quality issues of a system. QualityAssistant is a realtime

[6]http://pharo.org

engine that keeps track of the code quality reports and allows the developer to react. It provides plugins to the most popular tools that Pharo developers use to read or write code. This approach significantly reduces the feedback loop and not only allows developers react to quality issues more quickly, but should also allow us to collect fine grained data about the quality rule effectiveness.

IV. Evaluation Method and Results

The first contribution of our Ph.D. research is the Visual Design Inspection approach and the complementing ViDI tool. This tool was presented to Pharo community and used in a couple of use-case studies in our group. This allowed us to understand the main shortcomings of our approach.

Recently we have integrated a realtime quality feedback engine called QualityAssistant into the development version of Pharo. Just by releasing a beta version to the Pharo community, we obtained a feedback about the issues of current static analysis rules present in Pharo. We plan to continue using QualityAssistant as a source of feedback about upcoming improvements to quality rules and the way they are handled. To obtain fine-grained data we plan to setup a reporter, which will collect a data about how the developers are using the tool. This will provide us with information that can shed light on how developers react to bad quality reports, which rules are ignored more often, and whether developers are using the approach that we are devising.

V. Conclusions and Future Plan

During the first year we have created a prototype tool for visual design inspection called ViDI. After that we have made first steps towards modeling quality rules in a fuzzy and non-boolean way. Now we are constructing a real-time quality checker and instrumenting it in a way to bring a concept of quality into a standard development workflow.

The expected contributions of our Ph.D. research can be summarized as follows:

- A quality model that can be seamlessly integrated into development tools.
- ViDI: a code review tool leveraging visualization techniques and augmented with quality overview.
- QualityAssistant: a tool used both to give a realtime feedback about the code quality to a developer and to collect the usage data in order to improve quality rules.
- User studies to evaluate the usefulness of our approach and of its implementation.

VI. Acknowledgements

Yuriy Tymchuk thanks the Swiss National Science foundation for the support through SNF Project “ESSENTIALS”, No. 153129.

References

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

[2] A. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[4] K. Beck, *Smalltalk Best Practice Patterns. Volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ, 1997.

[5] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[6] J. Cohen, *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., 2006.

[7] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[8] P. Louridas, “Static code analysis,” *Software, IEEE*, vol. 23, no. 4, pp. 58 – 61, July 2006.

[9] A. G. Bardas, “Static code analysis,” *Journal of Information Systems & Operations Management*, vol. 4, no. 2, pp. 99 – 107, 2010.

[10] I. F. Darwin, *Checking C Programs with lint*. ” O’Reilly Media, Inc.”, 1991.

[11] 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.

[12] 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 35th International Conference on Software Engineering*, 2013, pp. 672 – 681.

[13] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” in *Proceedings of 20th IEEE International Conference on Software Maintenance*, 2004, pp. 350 – 359.

[14] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur, “Decor: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20 – 36, Jan 2010.

[15] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, “A bayesian approach for the detection of code and design smells,” in *Proceedings of 9th International Conference on Quality Software*, 2009, pp. 305 – 314.

[16] P. Caserta and O. Zendra, “Visualization of the static aspects of software: A survey,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 17, no. 7, pp. 913–933, July 2011.

[17] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[18] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Manning Publications Co., 2013.

[19] M. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Syst. J.*, vol. 15, no. 3, pp. 182 – 211, Sep. 1976.

[20] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *Proceedings of 35th ACM/IEEE International Conference on Software Engineering*, 2013, pp. 712 – 721.

[21] V. Balachandran, “Fix-it: An extensible code auto-fix component in review bot,” in *Proceedings of 13th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2013, pp. 167–172.

[22] 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.

[23] Y. Tymchuk, A. Mocci, and M. Lanza, “Code Review: Veni, ViDI, Vici,” in *Proceedings of 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015, pp. 151 – 160.

[24] ——, “Vidi: The visual design inspector,” in *Proceedings of 37th IEEE/ACM International Conference on Software Engineering, Tool Demo Track*. IEEE, 2015, pp. 653 – 656.

[25] 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.

[26] R. Wettel, “Software systems as cities,” Ph.D. dissertation, University of Lugano, Switzerland, Sep. 2010.

[27] R. Wettel, M. Lanza, and R. Robbes, “Software systems as cities: A controlled experiment,” in *Proceedings of 33rd International Conference on Software Engineeering*. ACM, 2011, pp. 551 – 560.

[28] G. Klir and B. Yuan, *Fuzzy sets and fuzzy logic*. Prentice Hall New Jersey, 1995, vol. 4.