ViDI: The Visual Design Inspector

Yuriy Tymchuk, Andrea Mocci, Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—We present ViDI (Visual Design Inspector), a novel code review tool which focuses on quality concerns and design inspection as its cornerstones. It leverages visualization techniques to represent the reviewed software and augments the visualization with the results of quality analysis tools. To effectively understand the contribution of a reviewer in terms of the impact of her changes on the overall system quality, ViDI supports the recording and further inspection of reviewing sessions. ViDI is an advanced prototype which we will soon release to the Pharo open-source community.**

Video URL: http://youtu.be/EtdkcNBJAec

I. Introduction

Modern software development often integrates code review in its process, both in large software companies and open source communities [1]. Modern code review is supported by dedicated tools. Examples include Gerrit[1] by Google, Crucible[2] by Atlassian and ReviewBoard[3]. Such tools provide a common core of features, like a diff view of the changes to be reviewed, the ability to comment and discuss parts of code, and mark a contributed patch as reviewed.

The main goal of code review is to anticipate the detection of defects, to improve code quality, and to share code knowledge among developers [1], [2], [3]. Code review is usually performed by a small number of developers on a contribution patch just before its integration into the software system. This style on review is called *peer-review* [2]. The code quality improvement is assumed to emerge from the experience of developers through their review contributions.

The popularity and potentials of code review adoption motivated Bacchelli and Bird [4] to study the expectations of developers and the difficulties they encounter when performing a review. They discovered that the main motivation for code review is to identify defects in code, and as a consequence to improve the code written by others.

Conversely, the main difficulty of code review is to comprehend the reason of a change to be reviewed. This difficulty hinders the review process, causing reviewers to focus on code style problems, which are easier to spot. Thus, reviewers are not able to effectively tackle software defects, and the ultimate goal of improving code quality is hindered.

To overcome this limitation, we adopted a novel point of view in the way code review tackles its expected core concern, without simply relying on the expertise of reviewers, but by considering the review of code as an integral and dedicated process which takes a system form one quality state to another.

[1]https://code.google.com/p/gerrit/
[2]https://www.atlassian.com/software/crucible/overview
[3]https://www.reviewboard.org

We present a tool called Visual Design Inspector (ViDI), which augments code review by integrating software quality evaluation, and more general design assessment, as a first class citizen and as the core concern of code review. It leverages visualization to drive the quality assessment of the reviewed system, exploiting data obtained through static code analysis. ViDI enables intuitive and easy defect fixing, personalized annotations, and review session recording.

A. ViDI Concepts

ViDI is implemented in *Pharo*[4], 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 [5] to support quality analysis and obtain reports about issues concerning coding style and design heuristics [6], [7]. 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 (*e.g.,* classes or methods). A rule can be checked against an entity, and its violation is called a *critic*. Some critics store the selection of code that violates a rule.

The system to be reviewed 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-evaluates the quality assessment, to keep the reviewer updated about the current system state.

Most review tools focus on a *patch* of code to be reviewed before being integrated into the code base. We focused ViDI on a wider context, namely the one of continuous assessment of the quality of a software system. While dedicated patch review is left as a future work, ViDI now tackles the need for a tool where quality concerns become an integral part of the development process. While ideally such a quality control should be performed continuously, for the time being we designed ViDI to operate in a session-based approach, where developers verify the quality of a system in dedicated sessions.

ViDI is thus rooted in the concept of a *review session*, that can focus on a package or a set of packages. During the review session, all changes made by reviewer are recorded. Sessions can be stopped, and the session-related data can be archived for future usages. Each session can be visually *inspected* at any time to understand the impact of the review, in terms of the amount of changes and how the quality of the system under review improved.

[4]http://pharo.org

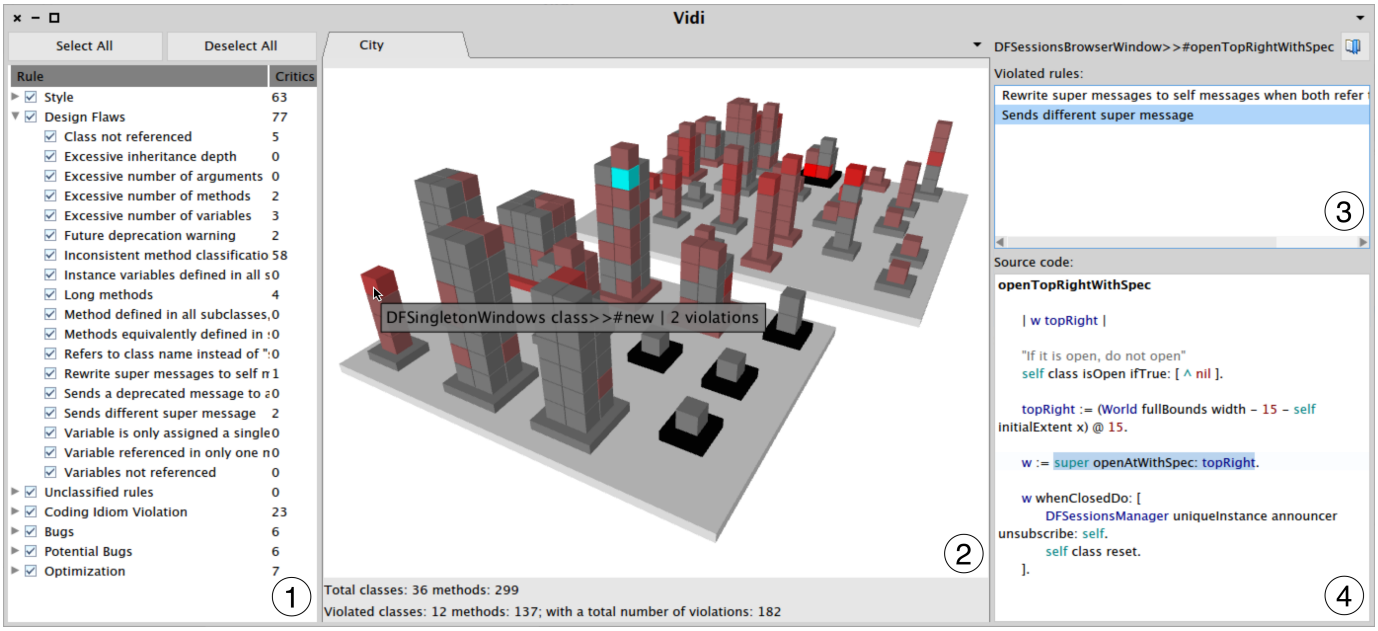


Fig. 1: ViDI main window, composed of 1) quality rules pane; 2) system overview pane; 3) critics of the selected entity; 4) source code of selected entity.

II. ViDI in a Nutshell

We illustrate ViDI through its main user interfaces: Its quality assessment window (Section II-A) and the session review window (Section II-B).

A. Quality Assessment Window

The main window of ViDI is depicted in Figure 1. It is composed of three horizontal panes, which respectively frame i) a list of categorized critics, ii) an overview of the system, and iii) detailed information about a selected entity.

The **Critics List** provides a categorized overview of the critics in the system. It provides two columns containing the name of the rule and the number of critics occurrences. Rules are hierarchically organized into predefined categories. Each rule and category can be deselected with a checkbox next to it. This removes the critics related to this rule (or entire category) from the other panes of the tool. By default, all categories are selected.

The **System overview** is the core visualization of ViDI supporting the quality and design assessment. It consists of a city-based code visualization [8], [9], depicting classes as bases on which their methods are stacked forming together a visual representation of a building. We plan to investigate complementary and alternative visualizations to the city-based one. The status bar provides a short system summary, containing information about the classes and methods under review, those which have critics, and the total number of critics on the system. The system overview pane supports immediate understanding of the quality of the system under review, relating its structure and organization with the distribution of critics. In this view, method and classes are colored depending on the amount of critics. Elements with no critics are colored in gray. The higher the amount of critics, the brighter is the red coloring of the entity. Hovering over the elements of the city displays a popup with the name of the element and the number of critics. Clicking on an element selects it, coloring it in cyan, and allowing further inspection in the rightmost pane of ViDI, the selection pane.

The **Selection Pane** supports inspection and modification of an entity (*i.e.,* package, class or method) selected in the system overview. The selected entity name is displayed on top of the pane, which is in turn vertically split in two parts. The top half of the pane displays the list of all visible critics about the selected element, while the bottom part displays the code of it. Clicking on one of such critics highlights the problematic parts in the source code. Source code is also editable: The reviewer can make changes to fix an issue and save them. When an element is changed, all the critics are re-evaluated on it. Furthermore, we have implemented the possibility to fix some critics automatically, and this option can be triggered from the context menu of a critic. Figure 2 shows such scenario.

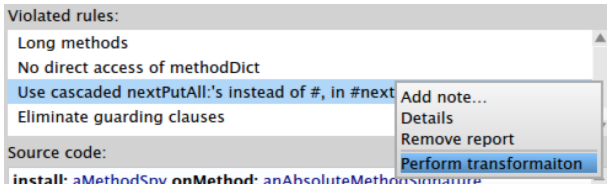


Fig. 2: Automatically fixing a critic

Another useful option is the inspection of the rationale of a critic and further details. Finally, another fundamental option is the possibility to add a note, the purpose of which is to leave

a reviewer comment related to the specific critic, propose a solution, or details on its rationale. Figure 3 shows a specific example of this scenario.

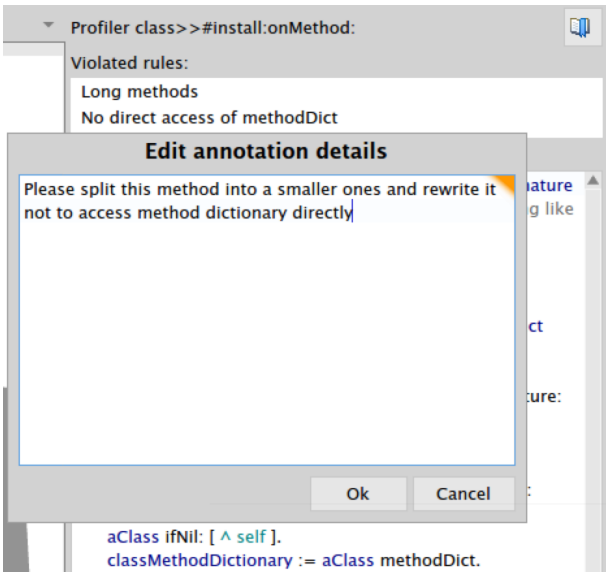


Fig. 3: Adding a note in ViDI

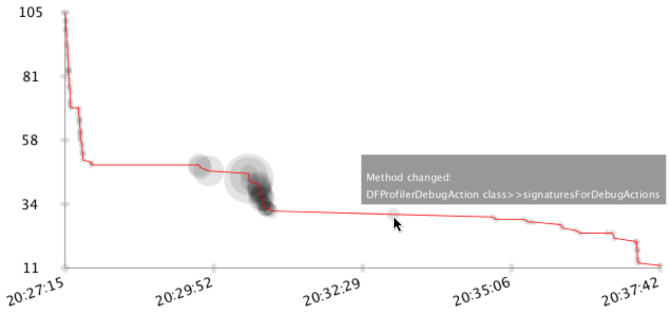
Notes are essentially considered as custom critics by the reviewer: Notes are stored alongside entity critics and they are fundamental for the purpose of evaluating a system's quality. The principle is that reviewer comments are at same level of automatically generated critics. After a note is added, it is displayed in the list of critics.

B. Session Review

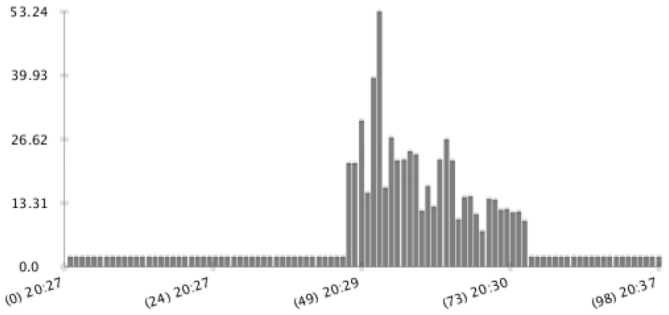
ViDI provides two effective visualizations to support review inspection and reflection.

The **Critics evolution view** displays the evolution of the total amount of critics during a review. Figure 4a shows the typical example of an effective review, where the graph is decreasing with some steep drops caused by automated batch fixes. With this visualization, the reviewer can immediately see that the session removed, in around 10 minutes, a significant amount of issues (*i.e.,* 94 critics). Moreover, the visualization displays the impact of each change as dark semitransparent circles, whose radii correspond to the *change impact*, a simple metric of how the change impacted the reviewed code. As a preliminary metric we quantified the impact as the number of edited characters in the source code. We plan to study alternatives as future work, for example metrics that take into account the nature of changes, like refactoring choices.

The **Change impact view** shows instead a histogram of changes made during the session. It allows to reason on the amount of changed code that corresponds to the number of resolved critics. The x axis contains the sequence of changes in the code, while the y axis shows the change impact. In both views, hovering over an entity shows a popup with information about the change, while clicking on it opens a dedicated diff view of a change.



(a) Critics evolution during a review session



(b) Impact of changes made during a review session

Fig. 4: Reviewing a Review Session

III. A Use Case: DFlow

By means of a brief use case, we discuss the effectiveness of ViDI on assessing and improving the quality of DFlow [10], a profiler for the Pharo IDE that records fine-grained user interface interaction data. DFlow consists of 8 packages, 176 classes and 1,987 methods. We reviewed a core package of DFlow which consists of 23 classes and 119 methods. The package makes heavy use of reflection mechanisms and *meta programming* [11] to instrument the IDE. Ensuring the quality of a core package is fundamental, as defects on it can cause crashes in the IDE. We illustrate a single session, performed by both the author of DFlow and the one of ViDI as a reviewer, the starting point of which is depicted in Figure 5.

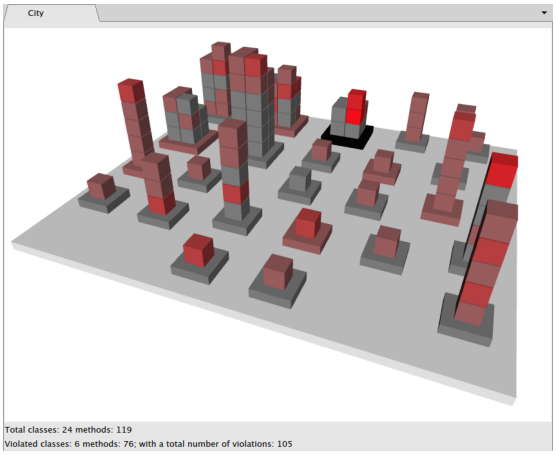


Fig. 5: Initial quality status of DFlow.

The system overview pane shows a relatively distributed number of critics. The two categories with the largest number of critics are "Unclassified methods" and "Inconsistent method classification", which are Smalltalk-specific critics related to the way the methods are organized within the classes. We decide to exploit automatic fixing for them. The resulting view gives us a clearer image to focus on more serious issues (Figure 6).

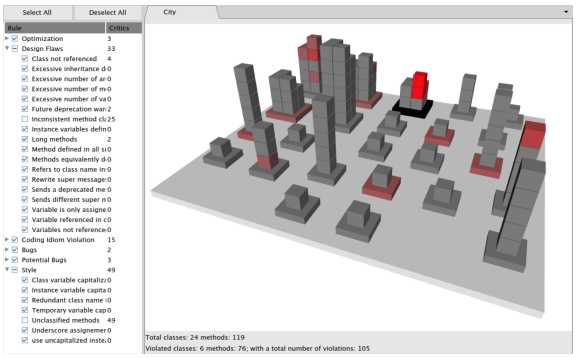


Fig. 6: Status after solving categorization issues.

An alternative would have been to deselect the entire category of critics without fixing them, or even to deselect all rules and then select just one or a few critics classes. This alternative choice would allow to focus on specific kinds of issues that may be more severe and important to a project.

At this point, the reviewer can again automatically resolve issues related code style and optimization. The remaining issues cannot be dismissed automatically. For example, there is a method violating two rules: the method is too long and it violates an issue related to reflection. The fact that these critics cannot be automatically fixed, and the fact that the reviewer is likely not the author of the method, leaves him in front of a choice. He could either manually fix the method or leave a note for future, further inspection. Figure 3, that we analyzed in the previous section, shows exactly this case: The note asks the author to split the method into shorter ones and remove direct access to internal class structure. The note is left as a full-fledged critic in the system, that the author will be able to inspect it when reviewing the system himself.

Figure 4a illustrated the critics evolution of this show case session, which was relatively productive: In a timespan of 10 minutes the number of critics went from 105 to 11. We can spot a couple of phenomena related to the way we designed ViDI. At the beginning, critics dropped under the mark of 58 critics because of the automated resolution of the first class of issues (*i.e.,* method classification). Then, after *20:29*, we can spot effective changes in the source code depicted as dark circles, corresponding to the fixing of style and optimization issues. The next change appears after *20:32:29*. This was a non-trivial issue that could not be automatically fixed. There is also a longer period without any change after the resolution in *signaturesForDebugActions*. This is because the reviewer was trying to understand how to solve the second issue, before writing another note.

A. Planned Studies

This was just a brief illustration of how ViDI is used. The user has a central, consistent view on the software to be reviewed, and then interacts with ViDI to inspect and fix problems of various nature (bad style, bad design, etc.). ViDI is still in a prototype status and needs to meet other desiderata of code review, like better process support and representation of critics. We plan to conduct the following future studies to assess the capabilities of ViDI:

- An *early release* of ViDI to the Pharo community to get early feedback from developers, supported by a *qualitative study* that will help refinement of ViDI to ensure it meets the expectations of reviewers;
- A solid *quantitative evaluation* aimed to measure the effectiveness of ViDI in terms of the numbers of defects that it helps to spot and solve;
- A user feedback system tied to a continuous release policy of ViDI, where feature requests are handled in a timely fashion;
- A full-fledged study aimed to *compare* ViDI with competitive approaches in the state of the art.

IV. Conclusion

We presented ViDI, a tool to support visual design inspection and code quality assessment as its core concern. It exploits reports generated by automatic static analysis to identify so-called critics to be reviewed. ViDI enables automated or manual fixing of critics, and allows reviewers to leave comments and notes that are elevated to the same status of full-fledged critics. Another feature of ViDI is that it records reviewing sessions that can be reviewed for further inspection, for example by highlighting how the system quality has improved, and enabling a real-time evaluation of the impact of changes on source code.

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] P. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, 2013, pp. 202–212.

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

[5] 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.

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

[7] 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.

[8] R. Wettel, "Software systems as cities," Ph.D. dissertation, University of Lugano, Switzerland, Sep. 2010.

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

[10] R. Minelli, L. Baracchi, A. Mocci, and M. Lanza, "Visual storytelling of development sessions," in *Proceedings of ICSME 2014*, 2014.

[11] N. M. N. Bouraqadi-Saâdani, T. Ledoux, and F. Rivard, "Safe metaclass programming," in *Proceedings of OOPSLA 1998 (13th International Conference on Object-Oriented Programming Systems, Languages and Applications)*. ACM, 1998, pp. 84–96.