Code Review: Veni, ViDI, Vici

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

***Abstract*—Modern software development sees code review as a crucial part of the process, because not only does it facilitate the sharing of knowledge about the system at hand, but it may also lead to the early detection of defects, ultimately improving the quality of the produced software. Although supported by numerous approaches and tools, code review is still in its infancy, and indeed researchers have pointed out a number of shortcomings in the state of the art.**

We present a critical analysis of the state of the art of code review tools and techniques, extracting a set of desired features that code review tools should possess. We then present our vision and initial implementation of a novel code review approach named Visual Design Inspection (ViDI), illustrated through a set of usage scenarios. ViDI is based on a combination of visualization techniques, design heuristics, and static code analysis techniques.

I. Introduction

Code review is a common software engineering practice used by organizations and open-source communities to improve the quality of source code [1]. During this activity the code is reviewed by software developers to check whether it complies with guidelines, to find defects, and to improve faulty parts. The most common form of code review is *peer review*, a semi-structured approach that can be more easily adopted (and adapted) for the specific needs of development teams [2]. During peer review, the changes to source code are reviewed by a small number of developers just before being integrated.

Modern code review is supported by dedicated tools. Popular examples include Gerrit and Rietveld by Google, Code Flow by Microsoft, Phabricator by Facebook, Crucible by Atlassian, etc. Most tools provide a set of common core features, such as a diff view of the changes to be reviewed, the ability to comment parts of code, discuss changes, and mark a code patch as reviewed.

Ideally, code review is an efficient way to improve code quality, detect critical bugs and share code ownership [1], [2], [3]. This effectiveness motivated Bacchelli and Bird [4] to study the expectations of developers and the difficulties they encounter when performing a review. They found that the main reason to perform the code review is to find defects in code and to improve the code written by others. The main difficulty of code review is understanding the reason of a change that one has to review. As a side effect of this problem, reviewers start to focus on the easier to detect code style problems, in essence going for the low hanging fruits. A natural consequence of this is that reviewers are not able to effectively tackle software defects, and the ultimate goal of improving code quality is hindered. We believe one underlying reason for this *status quo* is to be sought in the way code review tools are implemented and in the features they offer to reviewers.

We conducted a critical analysis of the state of the art of code review tools, shedding light on a number of facts. For example many of them leverage static code analysis techniques, like the ones provided by FindBugs [5], to spot implementation problems. However, the results from such techniques are poorly integrated in a code review process, as we will see later.

We propose an approach to augment code review by integrating software quality evaluation, and more general design assessment, not only as a first class citizen, but as the core concern of code review. Our approach, called Visual Design Inspection (ViDI), uses visualization techniques 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. While we implemented a running prototype as an open source MIT-licensed tool,[1] ViDI has a long-term vision that is beyond the scope of this paper. Thus, a central contribution of this paper is also a discussion on its open issues, its potential and the ideas that are part of our future work. We provide detailed showcase scenarios that illustrate the benefits and potential of ViDI and our vision. The showcase scenarios also clarify and exemplify the actual shortcomings that need further investigation and research.

This paper makes the following contributions:

- A critical analysis of the current state of the art in code review;
- ViDI, a new approach to integrate software quality assessment in a tool that visually supports the design inspection of software systems;
- Case studies to illustrate and assess the applicability and promising value of ViDI;
- A prototype tool implementation based on our approach.

Structure of the Paper. Section II provides a critical analysis of the current state of art on code review tools and approaches of static analysis to evaluate code quality. In Section III we describe the ViDI approach with its core features and its long term vision. In Section IV we assess ViDI through two illustrating case studies. Section V discusses the current status and limitations, and outlines future work. Section VI concludes the paper.

II. Critical Analysis of the State of the Art

In this section we analyze major existing code review tools, extracting and organizing their main features. We first describe how we identified the existing approaches, and we give a short overview of each tool (Section II-A). Then, we extract core features of the tools on which we compare them (Section II-B). Finally, we identify the set of desired features of an ideal code review approach, and we frame the actual contributions of ViDI (Section II-C).

[1] https://github.com/Uko/Vidi

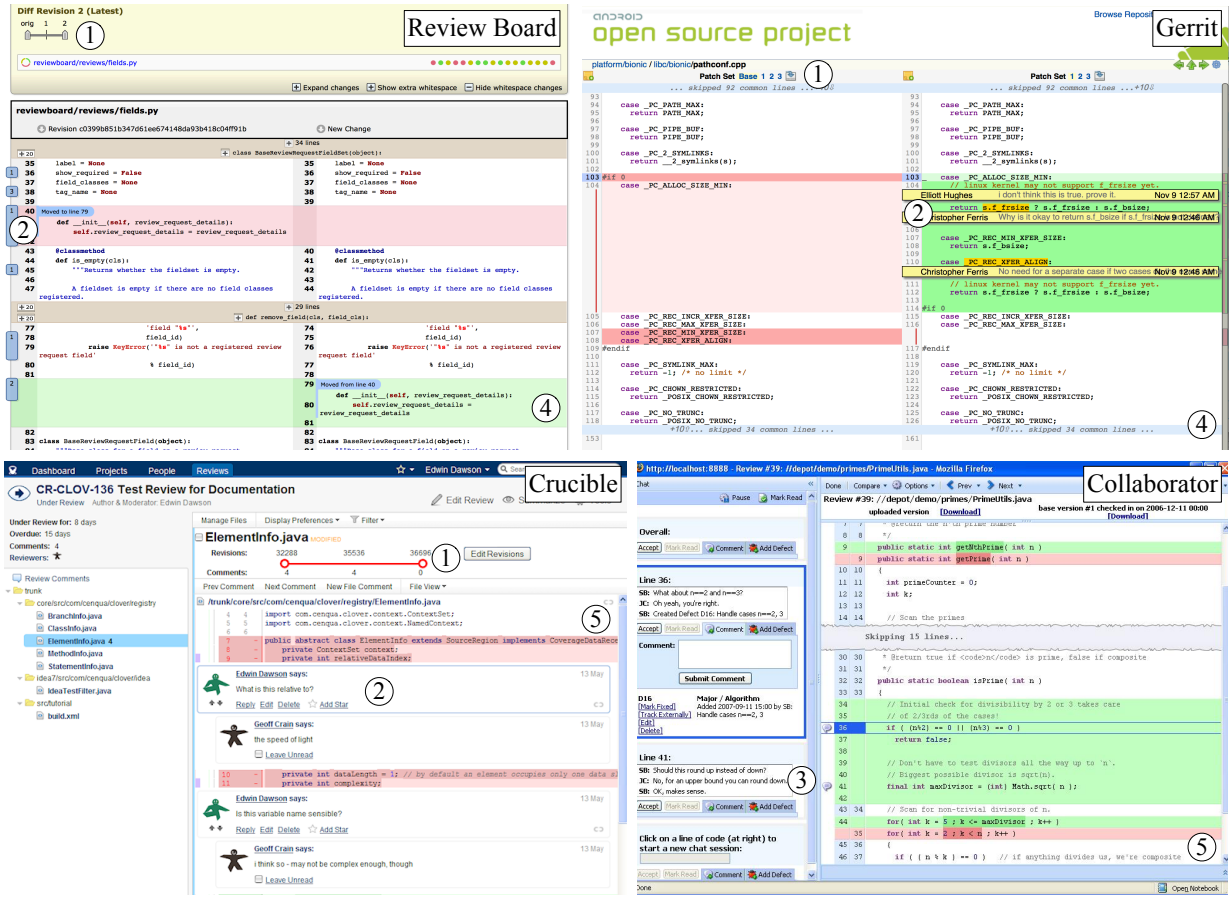


Fig. 1: Code review features: ① version breakdown; ② inline comments; ③ comment feed; ④ side-by-side diff; ⑤ unified diff.

A. *Tools Overview*

After surveying the related work in this area we identified a number of code review tools, listed in Table I.

TABLE I: State of the Art Code Review Approaches

Name	License	Website
CodeFlow	Proprietary	http://goo.gl/jH2YkP
Collaborator	Proprietary	http://goo.gl/C6oqI8
Crucible	Proprietary	https://www.atlassian.com/software/crucible
Differential	Apache v2	http://phabricator.org/applications/differential
Gerrit	Apache v2	https://code.google.com/p/gerrit/
GitHub	Multiple	https://www.github.com
Review Board	MIT	https://www.reviewboard.org
Upsource	Proprietary	https://www.jetbrains.com/upsource

CodeFlow is the tool on which Bacchelli and Bird [4] mainly focused on. Differential by Facebook is also mentioned by the same authors. Gerrit and Collaborator were used, together with CodeFlow, in the work of Rigby and Bird [3] for the analysis on contemporary software review practices. Balachandran [6] introduced Review Board and extensions for it. We also included the tools released by two important software development companies: Crucible by Atlassian and Upsource by JetBrains. Last but not least, we also reviewed the features of GitHub, since researchers have compared the mechanism of pull requests to the review process [7]. Figure 1 provides a set of screenshots coming from some code review tools, with some core features highlighted.

CodeFlow is a code review approach and tool developed by Microsoft. CodeFlow is partially integrated into the last public release of Visual Studio; i.e., CodeFlow uses file comparison (diff) capabilities provided by this IDE. The normal usage of CodeFlow is for pre-commit patches, which means that it is normally used to review code before commits to the versioning system are made. CodeFlow allows one to leave comments on any selected piece of code, which are then reported together with general review comments in a unique separate feed. It provides a context of review sessions for each person, but there seems to be no support for a review workflow. In particular, there is no mention to support review history; e.g., after a review, there is no way to link the reworked contribution to the original submitted code. CodeFlow supports review patches generated by Visual Studio, and it appears to be Software Configuration Management (SCM)-agnostic.

Gerrit is an open-source tool developed at Google for the Android project. It is a fork of Rietveld, the successor of the Mondrian review tool. It provides both side-by-side and unified diffs. It also provides a way to select which of the reviewed patch versions to get a diff of. Gerrit allows to leave comments on any code selection and they are shown between the lines just under the selection. This tool can be easily integrated with the Jenkins[2] continuous integration service.

[2]http://jenkins-ci.org

It can also benefit from the sputnik[3] project, which integrates comments on the review diff with the reports produced by quality checkers like Checkstyle[4], PMD[5] and FindBugs [5]. Gerrit can notify other people about a review, which is similar to the "watcher" feature of other tools. Gerrit works only with `git`, and this enables the so-called *gerrit-style* code review. It relies on a process where each new contribution is versioned in a new `git` branch [8]. Actual code review occurs when the merge request is issued, and in case of approval, the contribution branch will be merged into the main trunk of the project. This makes Gerrit essentially a post-commit review tool, that is, reviews are performed on code that has been already committed to a repository. Reviewers must also rate the patch when they conclude the review. Thus, based on the average review rating, a decision is made whether to allow or forbid the integration of a contribution. Having contribution patches available as commits in the source code repositories makes it possible to automatically run unit test before the review. It is also possible to integrate the review with additional data like static code analysis reports. Some extensions integrate Gerrit with issue trackers, but they are not yet mature.

Collaborator is the review tool by SmartBear, available either as a web client or as a plugin for IDEs like VisualStudio and Eclipse. It provides both a unified and side-by-side diff for patches. Collaborator is also a document review tool, as it provides special diffs for filetypes like Word documents and PDFs. Since the review process may consist of several iterations of changes, Collaborator allows to select which changes to see in the diff. The tool allows to leave comments per line of code and they appear in the general feed together with global review-related comments. As Collaborator can be used to review non-source code, it allows to leave comments on "push pins" that are associated to the coordinates of a reviewed file. The tool does not allow to run test and quality checks automatically, but SmartBear provides a guide on how to run test on Jenkins, and how to integrate quality analysis tools results into the review. Collaborator provides the ability to create reviewer pools that allow authors to select a whole pool for participation in a review. Besides reviewers, authors can invite observers who do not impact the review itself, but can spectate to familiarize with changes. This tool provides a review session for each person and integrates in into the global session called *workflow*. Workflows are customizable, as many other features of Collaborator. It can be used to perform both pre- and post-commit reviews: The main workflow is based on sequential patch submission and review, but it also supports the gerrit-style code review. Optionally, an issue tracker such as Bugzilla[6] can be integrated, allowing to open or associate an issue with the comment on a code review. Collaborator supports many version control systems, from free software like `svn` and `git`, to proprietary ones like IBM Relational Team Concert and Microsoft Team Foundation Server. Collaborator also has a lightweight version called CodeReviewer.

Crucible is part of Atlassian's family of development tools. This allows Crucible to be easily integrated with the issue tracker JIRA and other development tools of the company.

[3]https://github.com/TouK/sputnik

[4]http://checkstyle.sourceforge.net

[5]http://pmd.sourceforge.net

[6]http://www.bugzilla.org

Crucible supports only unified diff views, with no special support for non-source files. On the other hand, it provides a timeline view of review iterations and allows to select a time window of which it will generate the diff. Crucible allows to leave comments per line of code, and the comments are displayed just after each line. The tool provides a way to select groups of reviewers as well as suggest individual reviewers based on i) contributions to code, using total lines of code for the files currently under review; ii) availability, using number of outstanding reviews; iii) randomness, with two random users added to the list to help get fresh eyes on the code, and to spread the effort for code reviews. Crucible can be integrated with Bamboo continuous integration server, enabling unit tests to run automatically. At the moment, there is no straightforward way to integrate quality analyzers. Crucible does not provide dedicated support for observers, but allows to send emails with snippets of the review, which can be used for knowledge sharing. It also allows to mention users from the review comments, and it can create issues in JIRA issue tracker from the reviews. Integration with JIRA enables to link the issue to the related review. Crucible supports a dedicated workflow for reviewers to vote for integration or rejection and ultimately reach a consensus; it also supports moderators who take the full responsibility for the decision. Moreover, reviews are performed inside recorded sessions, collecting data about the time spent by the reviewer and percentage of code reviewed. Crucible mostly focuses on post-commit reviews, but it is also possible to use it for pre-commit reviews. The tool supports `cvs`, `svn`, `git`, Hg and Perforce SCMs.

Differential is part of the Phabricator development suite originally developed internally at Facebook. It is now an open source Apache-licensed project maintained by Phacility, Inc. Differential has side-by-side diff for code review. A review report page contains a list of all versions related to the review and allows to select which ones on which the diff should be based. The tool allows to comment multiple lines at once and comments are displayed between the code lines. Another tool currently being developed, called Harbormaster, should provide continuos integration for the tool suite. Command Line Interface (CLI) Arcanist allows to enrich code review with feedback obtained from static quality analysis tools. Another useful tool called Herald allows to define rules that automate reviewer selection. For example, it can assign a specific reviewer to each contribution committed by new interns. Finally, Phacility announced a new tool called Pholio, which will be used to review design sets. Differential allows one to add watchers to the review. It also supports different kinds of workflow, and reviewers should "approve" reviews in order to close it. Differential can integrate with Maniphest, the issue tracker of the Phabricator suite. As stated in its documentation, Differential focuses mainly on pre-commit reviews, but we managed to uncover only workflows with gerrit-style code review, and so we rather categorize it as post-commit review tool. Phabricator integrates with `git`, `svn`, and Mercurial.

GitHub is an online hosting service for `git` repositories. This analysis covers also GitHub Enterprise, which is a paid version for private development, and GitLab[7], the free analogue of GitHub Enterprise. Both unified and side-by-side diffs are provided. GitHub allows to compare changes made

[7]http://gitlabcontrol.com

TABLE II: Comparison of code review tools

Tool	Diff				Comments		Integration		Reviewers Selection	Review process			Code Navigation
	Unified	Side-by-side	Non-source	Versions	Global	Selection	Tests	Quality		Watchers	Workflow	Issues	
CodeFlow	✓	✓	—	?	✓	S	—	—	S	—	✓*	—	—
Collaborator	✓	✓	✓	✓*	✓	L+	✓	✓	S, G	✓	✓	✓	—
Crucible	✓	—	—	✓	✓	L	✓	—	S, G, A	—	✓	✓	—
Differential	?	✓	?	✓	✓	ML	✓	✓	S, R	✓	✓	✓	—
Gerrit	✓	✓	?	✓	✓	S	✓	✓	S	✓	✓	✓*	—
GitHub	✓	✓	✓	—	✓	L	✓	—	—	—	—	✓	—
Review Board	✓	✓	—	✓	✓	ML+	✓	✓	S, G, A	—	—	✓*	—
Upsource	✓	✓	—	✓*	✓	S	✓	✓	S, I	✓	—	✓*	✓

Legend: (?) unknown, (—) no support, (✓) full support, (✓*) partial support, (*N/A*) not applicable

to images in three different ways, renders changes in 3D STL files and uses special diff for prose documents. Github has a pull-based code review style similar to Gerrit [7]: It provides implicit review during pull requests. Users can discuss code and the review itself. This allows maintainers of open-source projects to review changes before integrating them into the project. A pull request theoretically ends by being integrated or closed, but after this users can still use the code review features. Moreover, GitHub allows to mention other users in order to attract them to review. Travis CI[8] can be easily integrated and provides feedback of whether the integration commit succeeded or not. Github works only with `git`, but lately `svn` access has been added to the repositories.

Review Board provides unified and side-by-side diff views. It also provides a timeline overview of the versions of a patch and the ability to select a time window upon which the diff will be generated. Review Board allows comments on multiple lines at a time. Commented line numbers are emphasized and clicking on them opens the comments in a popup window. Review Board also supports reviewing images, allowing to leave comments on a rectangular selection of the image. This tool has been influenced by VMware; Balachandran [6] introduced Review Bot as an extension for Review Board, which improves code quality understanding and helps with reviewers selection. Review Bot acts as virtual reviewer, and runs FindBugs, CheckStyle, and PMD to add review comments with their feedback. This functionality was later improved to automatically fix some detected defects [9]. To assist with reviewers selection, Review Bot analyzes the history of changes and suggests developers that worked on the same parts of code. This algorithm was improved by Thongtanunam *et al.* [10]. Review Board allows one to select predefined groups of developers for review. During a review session, comments can be transformed into issues that have to be explicitly resolved or dismissed. It does not provide any per-reviewer sessions, but provides an option of a special comment type called "Ship it". When this comment is used, the review is marked as positive. This does not restrict others to participate in the review, but this action cannot be undone. Review Board can be used for both pre- and post-commit reviews and supports Bazaar, ClearCase, `cvs`, `git`, `svn`, Mercurial, and Perforce SCMs.

Upsource, developed by JetBrains, is part of a family of tools such as IntelliJ IDEA, TeamCity, YouTrack, and others. Upsource provides unified and side-by-side diff views. It also provides the possibility to switch between the diffs of the versions included in the review, but this feature is significantly limited compared to other approaches. Comments can be placed over any selection of the source code and are displayed between the selected lines. Upsource will be soon integrated with TeamCity and YouTrack which are respectively the CI server and the issue tracker developed by JetBrains. Upsource provides a unique approach for code understanding and quality assessment. While browsing a patch diff in Upsource, the reviewer is able to take a peek into the JavaDoc [11] of an entity or jump to its definition in the same revision as you would do in the IDE. Upsource also uses the JetBrains static code quality analysis and visualizes it in the same way as IntelliJ IDEA. This tool is also a repository browser. At the moment it is supposed be used for free-form reviews on the available commits. Upsource does not provide any way to conclude review by the reviewers, but has the possibility to invite watchers to the review. Upsource provides a way to mention people from text and thus invite them to review. The tool provides post-commit review and supports `git`, `svn`, Mercurial, and Perforce SCMs.

[8]https://travis-ci.org

B. Features

Table II provides a catalog of core features for the code review approaches we previously selected and analyzed. We now proceed to explain each category of features in detail.

Diff. This is one of the basic concerns of a peer review, as it provides the main visualization of the changes that must be reviewed. Diff can be implemented in different styles and can support different formats. We found that code review tools support the following variations of diff support:

- *Unified:* diff is displayed in a single view, highlighting the changed lines to identify what was added/removed.
- *Side-by-side:* diff is represented as the previous and current version positioned side-by-side, which improves understanding of a change.
- *Non-source diff:* it can be special diff for either text files or binary files like PDFs or images.
- *Version breakdown:* a diff can span across multiple versions. We differentiate between full or partial support of this feature. Full support represents dedicated support to navigate a timeline of commits or iterations and select a timespan one wants to examine. Partial support represents the possibility to check only a pair of versions at a given time.

Comments. As modern code reviews are usually performed asynchronously, the communication between the reviewers

is accomplished by commenting the review, which can be implemented as follows:

- *Global:* represents the ability to leave a general comment regarding the whole review.
- *Selection:* enables commenting specific code entities; selection can be either a single line of code (*L*), multiple lines of code (*ML*), or any code selection (*S*). The plus sign (+) indicates that the tool provides dedicated selection and commenting in non-source code files.

Integration. This feature concerns integration with data coming from other development tools. We found the following supported integrations:

- *Tests:* this feature runs tests against a contribution to be reviewed. Since modified code may break some functionality, if a defect can be detected by automated test runs, the modification may be rejected automatically.
- *Quality analysis:* it provides integration with static code analysis tools like FindBugs or CheckStyle.

Reviewer selection. Code review tools may provide support to select who should review code. We found the following variations:

- *S:* the tool provides invitation for single reviewers;
- *G:* the tool has a notion of a group which can be invited for review;
- *A:* the approach provides an algorithm which suggests who should be invited;
- *R:* rules can be constructed to include specific reviewers automatically if certain conditions are met;
- *I:* identifies the possibility to invite reviewers by mentioning them in the text of a review.

Review process. This category identifies features supporting specializations of the review process.

- *Watchers:* it adds a new category of developers that just observe the review.
- *Workflow:* it forces a specific workflow of actions to reach the final state of the review, while some review tools follow a less structured process.
- *Issues:* determines whether the tool is integrated with an issue tracking system and thus can link to open a new issue.

Code navigation. This feature indicates support for IDE-like navigation of code and documentation during a review. As suggested by Bachelli and Bird, this feature can improve code understanding and thus improve the effectiveness of code reviews.

Summing up. As we can see from this overview, each tool was implemented with specific strengths and weaknesses, and there seems to be no common baseline requirements that such tools should fulfill. Moreover, such tools seem to not be an integral part of a software development process, but more as an afterthought. Last, most tools are dedicated to patch review, and not as general design assessment tools.

C. Desiderata of an Ideal Code Review Approach

What are the desiderata of an ideal code review approach? One could consider an idealized version of all the feature classes that we found in the analysis of the state of the art:

- **Precise Diff support**, to be able to effectively understand what changed in a given contribution to be reviewed;
- **Support for effective code commenting,** since it is mainly through comments that reviewers communicate their concerns and objections during a review session;
- **Integration with tools for testing and quality assessment,** to facilitate the understanding of the issues in a given contribution to be reviewed;
- **Optimal reviewer selection,** to identify the best people that could perform a given code review task;
- **Support for a clear, defined review process,** to facilitate integration with tools like the issue tracker;
- **Effective code navigation,** to support understanding of code to be reviewed and more easily identify possible issues related to code design and architecture.

However, these desiderata are abstract and general. Many features are implemented slightly differently in each tool supporting code review, and it is unclear what would be the best choice for each category of features in an ideal approach. A definite, finalized answer about all the desired features of an ideal code review approach, and how to realize them, is beyond the scope of this paper: Instead, after the analysis of the state of the art, we focus on the challenges identified by Bacchelli and Bird [4] as a starting point, from which we frame the contribution of ViDI. An interesting aspect is that CodeFlow is missing features if compared with the other tools (see Table II): it is missing integration with static code quality analyzers, which can be essential to easily spot defects of code to be reviewed. According to Bacchelli and Bird the top-ranked motivations of developers for code review are *finding defects* and *code improvement*. Moreover, authors found out that defects – while being the top motivation for code review – occur less frequently (*i.e.,* at rank four) among the actual results of code review. This is unsurprising: authors also discovered that finding defects is perceived as the activity which requires the highest level of code understanding among the expected outcomes of code review. The analysis of the state of the art backs up, as further evidence, the fact that the concern about software quality is relevant and not really addressed by the currently available approaches. Most of the existing approaches which support integration with code quality assessment tools like FindBugs are limited in the sense that they simply report their results as additional data to the review, and thus they are poorly integrated in the review process. Furthermore, there is a surprising lack of support for code navigation, which is essential to enable code understanding, a prerequisite to find defects and ultimately improve software quality.

We frame the focus of this paper, and the conceptual contribution of ViDI, to a single specific concern of code review, that is, software quality evaluation and more generally *design assessment*, escalating its importance as the core concern of core review. We do not intend to demean the other desiderata, but are confident that once the core concern of code review has been properly addressed, we could better address also the other concerns, which is part of our future work.

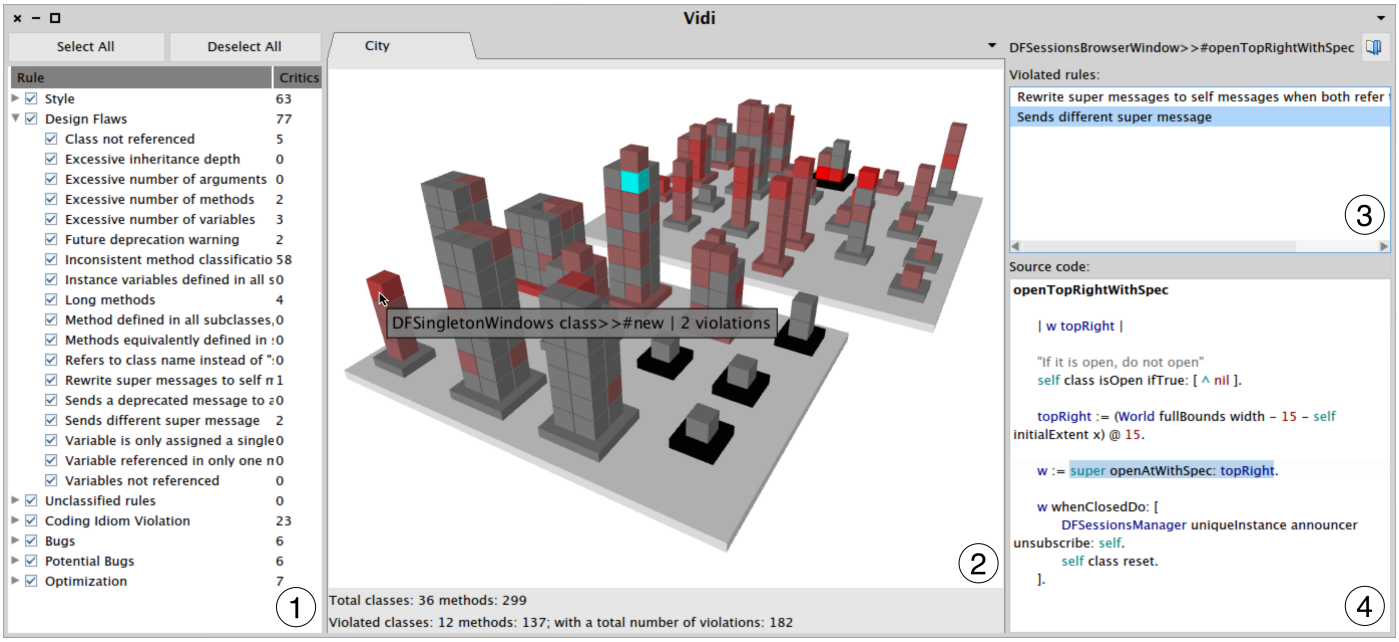


Fig. 2: ViDI main window, composed of ① quality rules pane; ② system overview pane; ③ critics of the selected entity; ④ source code of selected entity.

III. Visual Design Inspection

A. Philosophy and Core Concepts

As we saw in the previous section, most review tools focus on a specific context, the one of pieces of code (patches) that need to be reviewed before being allowed into the release version of the system code base. We argue that this is a specific scenario of a wider context, namely the one of continuous assessment of the quality of a software system. We believe there is the need for an approach where quality concerns are not reported only for patches, but become an integral part of the development process. In the ideal case such a quality control would be performed in real-time, but for the time being we opt for a session-based approach, where developers verify the quality of parts of a system (either old parts, or newly contributed parts, such as patches) in dedicated quality assessment 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 and can be accessed in the future. The system to be reviewed is presented in a dedicated visual environment augmented with automatically generated quality reports. The environment is self-contained: The reviewer can navigate, inspect and change the system from inside ViDI: ViDI supports both system understanding and improvement in an integrated environment. 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 state of the system. Sessions can be stopped, and the session-related data can be archived for further usage. Furthermore, the 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 system under review improved from the perspective of code and design quality.

ViDI is implemented in *Pharo*[9], a modern Smalltalk-inspired programming language and full-fledged object-oriented development environment. We use SmallLint [12] to support quality analysis and obtain reports about issues concerning coding style and design heuristics [13]. It is similar to other tools, like FindBugs [5], that exploit static analysis of code to identify defects. The version of Smalllint that we used has 115 rules that are organized into 7 different categories, which range from simple style checks to more complex design flaws. Each rule concerns specific code entities (*i.e.,* classes or methods), and can be checked against them to determine if the rule is violated or not. A violation of a rule is called a *critic* and means that the software entity does not satisfy the rule prescriptions. As opposed to the output of FindBugs, in SmallLint critics are full-fledged objects which can be manipulated, inspected, etc.

B. User Interface

The main window of ViDI is depicted on Figure 2. It consists of three horizontal panes, which respectively provide i) a list of categorized quality rules violations (critics), ii) an overview of the system, and iii) detailed information about a selected entity.

Critics List. This pane provides an organized overview of the occurrences of critics in the system. The list of critics provides two columns containing the name of the rule and the number of critics occurrences in the current system. 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 category) from the other panes of the tool. By default, all categories are selected.

[9]http://pharo.org

The **System overview** consists of a city-based code visualization [14], [15]. We depict classes as bases on which their methods are stacked forming together a visual representation of a building. A status bar provides a short summary about the system, 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 how critics are distributed over it. In this view, method and class colors also depend on the amount of critics. The element with the most critics is colored in bright red. This color gradually changes to gray as number of related critics decreases. Elements with no critics are colored in gray. The view considers only the critics and categories selected in the critics list. Hovering over the elements of the city displays a popup with the name of the element and the number of critics, if present. Clicking on an element selects it: When an element is selected, it is colored in cyan and can be inspected in the rightmost pane of ViDI.

The **Selection Pane** is dedicated to inspection and modification of an entity (*i.e.,* package, class or method) selected in the system overview. The name of the selected entity is displayed on top of the pane, while the rest of the pane is split in two parts. In the top half, the pane contains the list of all visible critics about this element. Selecting one of the critics highlights the problematic parts in the source code, which is displayed in the bottom part of the pane. The reviewer can make changes in the source code and save them. When an element is changed, the critics are re-evaluated.

ViDI extends some of SmallLint rules to provide automatic fixes. This option can be triggered from the context menu of a critic. For example, Figure 3 shows how by clicking "Perform transformation" in the context menu ViDI will automatically fix the problematic part of the method.

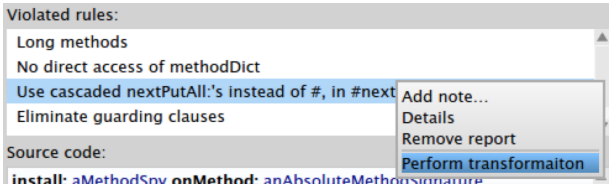


Fig. 3: Automatically fixing a critic

Another option offered by the context menu is the inspection of the details of a critic, that illustrate its rationale and further details. Finally, another option is to add a note, the purpose of which is for the reviewer to leave a comment related to the specific critic, propose a solution, or details on its rationale. Figure 4 shows a specific example of this scenario.

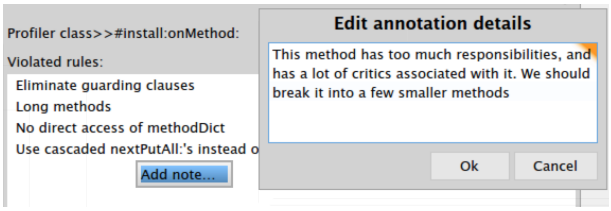
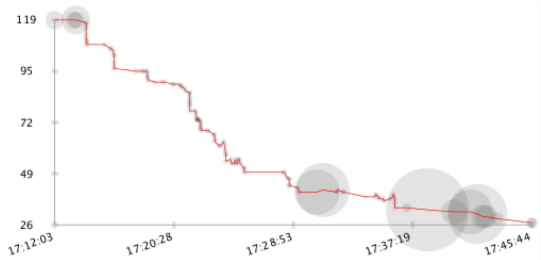


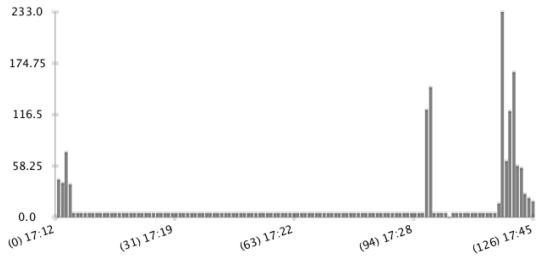
Fig. 4: Adding a note in ViDI

After a note is added, it is displayed in the list of critics: Such a note is essentially a custom critic by the reviewer. Notes have the same role and importance of critics: They are stored alongside entity critics and they are equally considered fundamental for the purpose of evaluating the quality of a system. The purpose is to elevate reviewer comments at the same level of automatically generated critics.

At the end of a session, or at any moment, the reviewer can reflect on the session itself and understand the effects of the review on the system. We designed and implemented two complementary views: the critics evolution view (Figure 5a), and the changes impact view (Figure 5b).



(a) Critics evolution during a review session



(b) Impact of changes made during a review session

Fig. 5: Reviewing a Review Session

The **Critics evolution view** displays how the amount of critics changes in time during a review. Figure 5a shows an example where the graph is monotonically decreasing, with minor exceptions (around 17:36). With this view, the reviewer can visualize the fact that the session decreased a significant amount of issues in the reviewed code, from 119 initial critics to 26 critics, in a timespan of around 35 minutes. The visualization also displays the impact of each change, displayed as dark semitransparent circles, whose radii correspond to the change impact.

The **Change impact view** shows a histogram of changes made during the session 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, the y axis shows the *change impact*, a simple metric of how the change impacted the reviewed code. As a preliminary metric we chose the number of changed characters in the source code. We plan to study alternatives that would take into account the nature of each change to code, like refactoring choices. 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.

IV. ViDI Demonstration

In this section we walk through two review sessions to demonstrate ViDI: The former is about ViDI on ViDI itself (Section IV-A), and the latter is on DFlow[10] [16], a profiler for developer's actions in the IDE (Section IV-B).

A. ViDI on ViDI

We developed ViDI following the principle of *dogfooding* [17], that is, by using ViDI on ViDI itself to continuously validate its capabilities. Once we reached a working prototype status, we started reviewing and developing ViDI with the support of ViDI itself. This allowed us to refine concepts and ideas, but it also helped us to upkeep the quality of ViDI's code, which is what we focus on here. At the moment, ViDI contains 23 classes with 201 methods and extends[11] 14 classes with 33 methods. Figure 5a and Figure 5b, that we discussed previously, show one of the many review sessions of ViDI. That session involved mainly method categorization. Such critics are Smalltalk specific: In Smalltalk, methods can be categorized, meaning that a developer can assign it a category representing the class of purpose of the method. ViDI helped to solve these critics, and others that were automatically fixable, many times during its own development. This ability is effective to focus on more important design issues, alleviating the burden of checking a long report originated by a static analysis tool. Moreover, by assigning an impact to changes, we could also more effectively review the more important changes we performed on ViDI. Figure 5b shows how most of the changes are in fact minor, automatically resolvable, issues of low impact. The changes with higher impact focus on three different moments of the session, in the beginning and in the end of a session, when the developer solved a number of style-related critics involving the use of conditionals, and other functional critics that required refactoring of methods. Unfortunately, a couple of rules solved in the end of the session suggested invalid changes to the code, and led to malfunctions of ViDI that were corrected in subsequent reviewing sessions. This suggests that SmallLint, the static code analysis technique that we currently use, has some shortcomings; in the long term, we plan to address them specifically, but at the current point, we assume that the critics we obtain can be generally trusted. Finally, the developer noticed some missing features from ViDI while reviewing a specific piece of code. In particular, he added two accessors to get the start and end time of a session. Even if the modification was not motivated by solving a critic, this is an interesting use case of ViDI that we plan to better support, for example by asking the reviewer a motivation for the change when it is not clear which was the issue he was trying to solve.

The current quality state of ViDI can be seen in Figure 6; it still contains few critics that we could not resolve. As we are working with Pharo, a dynamically typed language with no type inference [18], many rules are based on limited imprecise heuristics. This leads to misleading false positives. For example, since ViDI uses reflection, we found rules that identified bad style in references to abstract classes, which however is fine when using reflection methods.

[10]http://dflow.inf.usi.ch

[11]In Smalltalk it is possible to extend classes by adding methods in "ghost" representations of classes located in other packages.

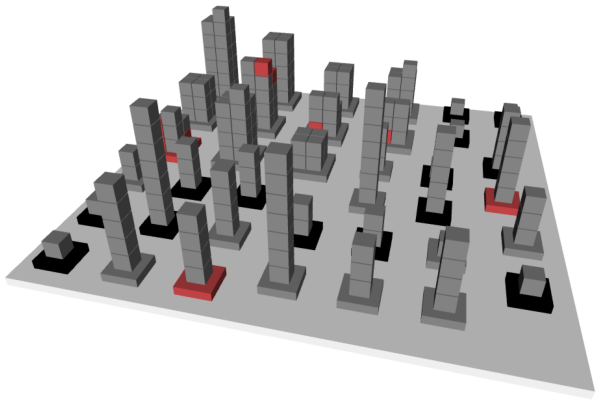


Fig. 6: Vidi quality status

This suggests either refinement of SmallLint critics or, more likely, improvements of ViDI to manage exceptions and the personalization and localization of rules.

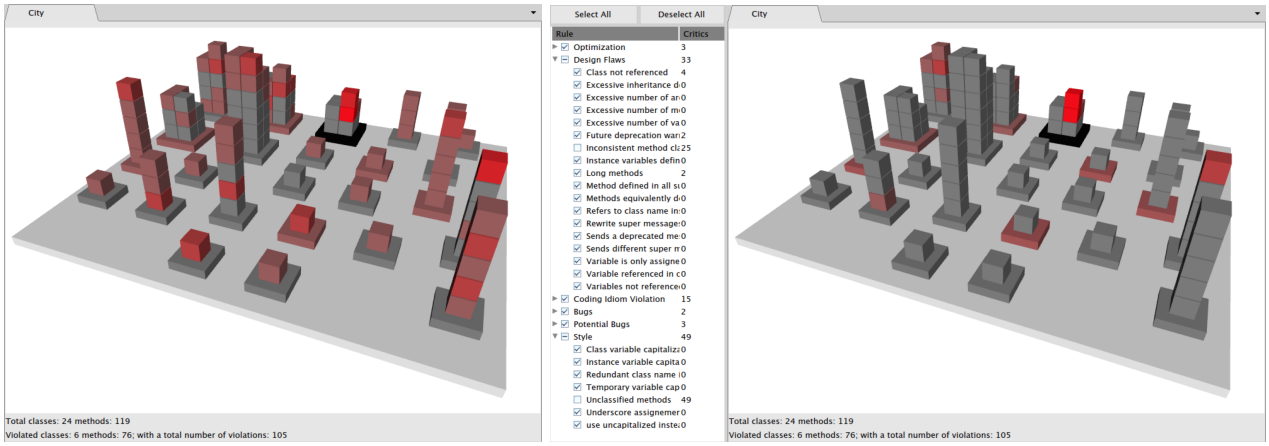
Another class of rules that we needed to ignore are rules that are general and conceptually fuzzy, like rules detecting overlong methods. For example, some specific api usages (*e.g.,* for visualization) tend to generate methods which contain many parameters to be set through method calls. This domain-specific API usage generates relatively long methods, which in turn generate critics by SmallLint. However, such critics are false positives because such long methods are due to domain-specific usages, and not because of method complexity. Again, this suggests the need for a specific management of exceptions.

B. ViDI on DFlow

DFlow consists of 8 packages, 176 classes and 1,987 methods. We reviewed a single package which consists of 23 classes and 119 methods. The package uses *meta programming* [19] to instrument and augment the Pharo IDE with additional functionalities. The quality of such a package is essential, as it can break the IDE and cause issues to development itself.

The starting point of the session is depicted in Figure 7a. 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". Critics point out that a method has no category, or that the category of the method is different from the one of the method that it overrides. As these violations are related to documentation, and they do not lead to real bugs, we can decide to omit them by deselecting the checkboxes next to related rules. The resulting view gives us a clearer image to focus to more serious issues (Figure 7b). Another way to assess the quality of the system is to deselect all rules and then select just one or a few them. This allows to focus on specific kinds of issues that may be more important to a project.

After filtering unimportant rules, a reviewer can automatically resolve issues related to code style and optimization. This leaves more complex issues that neither can be dismissed because they are important, nor can they be fixed automatically. For example there is a method violating 2 rules: the method is too long and it directly access a class methods structure, which is specific to the current implementation of Pharo.



(a) All critics visible.

(b) Unclassified and inconsistently classified methods critics hidden.

Fig. 7: Initial state of the review sessions.

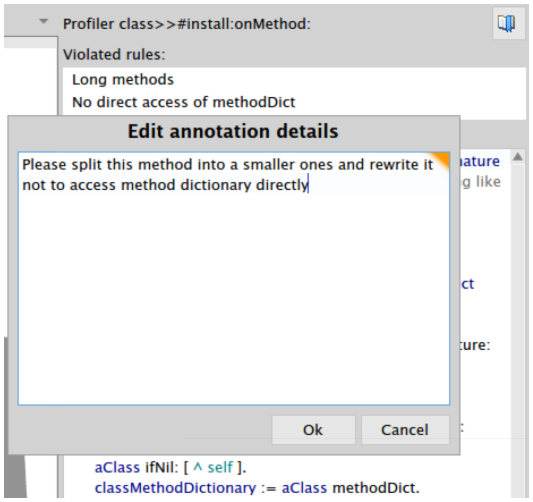


Fig. 8: Commenting on complex issue.

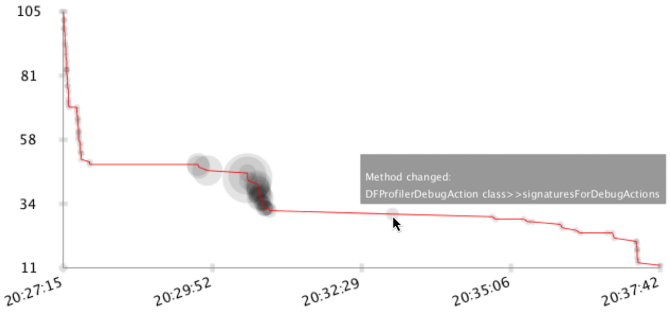


Fig. 9: Critics evolution on a DFlow review session

Suppose the reviewer is not the author of the method. The fact that critics cannot be automatically fixed leaves the reviewer in front of a choice: He could either manually fix the method or leave a note for future inspection. In the latter case, the reviewer can ask the author to split the method and remove direct access to internal class as shown on Figure 8. The note is left as a full-fledged critic in the system, that can be inspected when reviewing the system. Notes are stored in ViDI and can be manually exported and imported by a reviewer.

Figure 9 shows the critics evolution of the session, which was relatively prolific: the number of critics went from 105 to 11 in just 10 minutes. At the beginning, critics almost instantly dropped under the mark of 58 critics. This was caused by the automated resolution of categorization issues. Then, after *20:29* mark style and optimization issues were fixed which generated changes in the code, and so this part contains dark circles with larger diameters. These fixes also correspond to a steep drop on the number of critics, because resolution was automated. The next change appears after *20:32:29*.

By hovering over the circle, the reviewer can see a popup which informs that this change was a modification of a method called *signaturesForDebugActions* in a class *DFProfilerDebugAction*. A diff of this change can be seen by clicking on the circle. This was a non-trivial issue that could not be automatically fixed, as the reviewer was understanding how he should resolve the issue. 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 resolve the second issue and writing a note to the author. At the end there is a part where the critics line descends. These changes corresponded to the reviewer manually categorizing methods. Finally, close to the end, another steep drop can be seen. This happened because the reviewer categorized methods on the top of a class hierarchy and overriding methods at the bottom were categorized automatically.

V. Discussion

As we illustrated in Section IV, ViDI can be used not only to visually inspect the design of a software system, but also to effectively solve critics, ranging from simple style checks to more complex issues. The case studies we analyzed pointed out both benefits and important shortcomings of ViDI, that we now analyze to take a critical stance against our tool.

Critics Representation and Detection. We found relevant shortcomings in the way SmallLint critics are represented and detected. There is significant research to be done in detecting high-level design issues, for example by integrating mechanisms based on object-oriented metrics [20]. Another shortcoming we found involves false positives, like the ones related to long methods. While some rules require improvement in their representation, others may require a new representation of rules themselves. All SmallLint critics return a boolean result about the evaluated code entity, that is, they either violate the rule or not. This is too rough: Ideally, rules should have a severity grade [21], to identify the entities where rules are violated more seriously and to focus on them first.

Fixing Critics. At the current status, some of critics can be solved automatically, while others require manual fixing by the developer. Requiring a manual fix does not mean that we should not provide at least semi-automatic support for resolution, especially for critics that would require specific refactoring methods. For example, proposed changes can be presented to the reviewer before being applied, and can be personalized to adapt them to meet the reviewer intention.

Notes and Review support. ViDI gives a basic support to leave notes on code entities, which are treated as full-fledged critics. This idea can be expanded in many directions, for example to support more complex comments [22], [23] that are common on other code review tools, or to provide dedicated mechanisms to handle exceptions and personalizations.

Diff support. We provide basic support for code diff. We plan to improve ViDI by considering related approaches outside the area of code reviewing. For example, the approach of supporting integration of source code changes provided by Torch [24] could inspire solutions for ViDI on code contributions, and not on the analysis of entire systems.

VI. Conclusion

We presented ViDI, an approach that envisions quality inspection as the core concern of code review. We focused on this particular concern after a detailed analysis of the state of the art of code review approaches, which is another contribution of this paper. ViDI supports design inspection by providing a dedicated user interface that enables an immediate understanding of the overall quality of a system. It leverages automatic static analysis to identify so-called critics in the code, it enables their inspection and fixing, either manually or automatically. Moreover, we designed ViDI to record reviewing sessions that can be inspected (and reviewed) at any time, highlighting how the system has been improved during the session, and enabling a quick evaluation of the impact of changes performed by the reviewer. We performed a preliminary assessment of ViDI by providing two case studies, involving the review of ViDI on ViDI itself, and on DFlow, an advanced IDE profiler. Given a number of conceptual limitations and shortcomings, ViDI is just the first step for the more ambitious goal of providing a full-fledged design inspector to support all code review desiderata. In that sense, the *vici* is our future work.

Acknowledgments

We thank the Swiss National Science foundation for the support through SNF Project "ESSENTIALS", No. 153129.

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] N. Ayewah, W. Pugh, D. Hovemeyer, D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, Sept 2008.

[6] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, 2013, pp. 931–940.

[7] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of ICSE 2014 (36th ACM/IEEE International Conference on Software Engineering)*, 2014, pp. 345–355.

[8] D. E. Perry, H. P. Siy, and L. G. Votta, "Parallel changes in large-scale software development: An observational case study," *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 3, pp. 308–337, Jul. 2001.

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

[10] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *Proceedings of CHASE 2014*, 2014, pp. 119–122.

[11] D. Kramer, "Api documentation from source code comments: A case study of javadoc," in *Proceedings of SIGDOC 1999 (17th Annual International Conference on Computer Documentation)*, 1999, pp. 147–153.

[12] 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.

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

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

[15] 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.

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

[17] W. Harrison, "Eating your own dog food," *IEEE Software*, vol. 23, no. 3, pp. 5–7, May 2006.

[18] J. Palsberg and M. I. Schwartzbach, "Object-oriented type inference," *SIGPLAN Not.*, vol. 26, no. 11, pp. 146–161, Nov. 1991.

[19] 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.

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

[21] M. Lungu, "Reverse engineering software ecosystems," Ph.D. dissertation, University of Lugano, Switzerland, Oct. 2009.

[22] A. Brhlmann, T. Grba, O. Greevy, and O. Nierstrasz, "Enriching reverse engineering with annotations," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, vol. 5301, pp. 660–674.

[23] Y. Hao, G. Li, L. Mou, L. Zhang, and Z. Jin, "Mct: A tool for commenting programs by multimedia comments," in *Proceedings of ICSE 2013 (35rd International Conference on Software Engineeering)*, 2013, pp. 1339–1342.

[24] V. U. Gomez, S. Ducasse, and T. D'Hondt, "Visually supporting source code changes integration: The torch dashboard," in *Proceedings of WCRE 2010 (17th Working Conference on Reverse Engineering)*, 2010, pp. 55–64.