Accepted at IEEE Working Conference on Software Visualization (VISSOFT 2017), Copyright IEEE

# Towards the Visualization of Usage and Decision Knowledge in Continuous Software Engineering

Jan Ole Johanssen*, Anja Kleebaum†, Bernd Bruegge*, and Barbara Paech†

*Technical University of Munich
Department of Informatics
Garching b. München, Germany
{jan.johanssen, bruegge}@in.tum.de

†Heidelberg University
Institute of Computer Science
Heidelberg, Germany
{anja.kleebaum, paech}@informatik.uni-heidelberg.de

***Abstract*—Continuous software engineering (CSE) includes activities to continuously evolve software artifacts. Along with these activities, developers employ knowledge such as usage and decision knowledge. Usage knowledge helps developers to understand how users apply software, while decision knowledge comprises all decisions taken during CSE and their rationale. However, due to the frequency, extent, and complexity of knowledge in CSE environments, accessing and processing knowledge is challenging for developers. We propose a dashboard for developers that visualizes knowledge from various sources. This enables developers to follow, reflect, interact, and react on knowledge in CSE environments. We introduce springboards that serve as knowledge selectors while the selected extract of knowledge is visualized in widgets. Widgets allow developers to gain insight into knowledge. We suggest three widget classes: spot, compare, and range. We discuss dashboard extensions such as interaction triggers to add, modify, or combine knowledge. We plan to implement the dashboard and evaluate it with teams during software development projects in an industrial setting.**

## I. Introduction

In recent years, continuous software engineering (CSE) evolved as a frequently applied approach to ”develop, release and learn from software in very short rapid cycles” [1]. CSE encompasses activities such as continuous integration and continuous delivery [2]. These activities are characterized by fast iterations on software increments. Each software increment carries knowledge, for example regarding its implementation process or user acceptance. Developers employ this knowledge to further improve their software. This becomes a difficult task considering the frequency, extent, and complexity of knowledge in CSE environments. As part of our previous efforts to integrate usage and decision knowledge in CSE, we indicated the need for a dashboard [3]: such a dashboard enables developers to access, visualize, and analyze knowledge.

We focus on usage knowledge and decision knowledge, since CSE opens new opportunities to integrate both knowledge. Usage knowledge, i. e. knowledge of how software is applied by users, profits from the rapid release of new increments in CSE. For instance, continuous delivery guarantees that users always give feedback on the latest version of the software and enables the prompt replacement of deprecated software. Decision knowledge, i. e. knowledge of the decisions taken during software development and their rationale, benefits from rapid cycles of practices such as merging branches in the version control system or status changes of issues in the issue tracking system. These practices involve the documentation of decision knowledge in commit messages or in issue comments, respectively. As discussed in our previous work [3], we pose that the joint consideration of usage and decision knowledge leads to an improved development process and increased software quality. However, the combined visualization of this knowledge is an open question.

In this paper, we introduce a dashboard to visualize both usage and decision knowledge in CSE environments. Its main goal is to allow developers to *follow*, *reflect*, *interact*, and *react* on knowledge available in CSE environments. Therefore, the dashboard visualizes data from various sources, such as issue tracking systems or data analytics platforms. One important idea is to use springboards. Springboards enable developers to select points or intervals of interest. Each springboard provides an individual perspective on a software project. For instance, a commit springboard visualizes branches and code commits, while further filters can be applied to highlight certain aspects. Three widgets—spot, compare, and range—visualize knowledge associated to the springboard selection. We suggest that the dashboard should offer possibilities for visual interaction, enabling basic commands that frequently occur during the developers’ daily work.

The paper is structured as follows. Section II discusses visualization approaches of usage and decision knowledge in literature. Section III describes the components for knowledge preparation. The dashboard visualization is presented in Section IV and discussed in Section V. Section VI sketches our current and future work and Section VII concludes the paper.

## II. Related Work

To the best of our knowledge, the visualization of knowledge in CSE—in particular the visualization of different knowledge types such as usage and decision knowledge at the same time—has not yet been explored in previous research.

However, there is work regarding the separate visualization of usage and decision knowledge. For example, the visualization of usage knowledge is addressed by Guzman *et al*. [4]: they propose visualizations to make user feedback accessible.

Definitive version available at https://doi.org/10.1109/VISSOFT.2017.18

Analytic platforms provide detailed insight into the application usage. In general, knowledge on users' interaction forms the basis to reiterate user interface designs [5].

The visualization of decision knowledge is addressed by Lee and Kruchten [6] as well as by Shahin *et al*. [7]. Hesse *et al*. [8] use the DecDoc tool to visualize single design decisions as a tree of decision components such as arguments or alternatives. Notably, this work focuses on design decisions and not on decisions regarding generic software artifacts during CSE.

Sharing and understanding knowledge in agile software development is important, yet difficult to implement [9]. Alperowitz *et al*. work on a set of metrics to make knowledge in agile project courses measurable and provide a visualization in the form of a report sheet [10]. Elsen introduces a tool to visualize complex branch structures in git environments [11].

The presented work shares a focus on the visualization of specific aspects. However, it differs from the work presented in this paper: we introduce a dashboard that allows for a comprehensive and all-encompassing visualization of usage and decision knowledge related to software artifacts during CSE. As described in our prior work [3], this dashboard represents a major part of our research project CURES ("Continuous Usage- and Rationale-based Evolution Decision Support").

## III. Knowledge Preparation

The dashboard bundles various knowledge gathered from different sources. Figure 1 illustrates involved components.

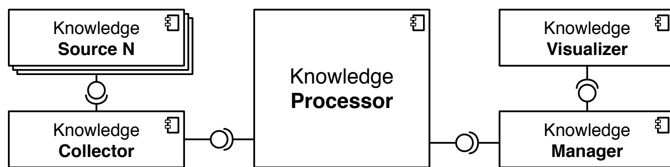


Fig. 1. Preparing knowledge from various sources for visualization.

A *Knowledge Collector* component is in charge of gathering knowledge from different *Knowledge Source*s, such as an issue tracker, a code repository, a wiki, or an analytics platform.

It provides knowledge to a *Knowledge Processor* component, which prepares knowledge to be visualized in accordance with other knowledge. This includes important steps of the knowledge preparation process, for example converting raw user click streams into a structured sequence of instructions or linking decision knowledge to an individual commit. The *Knowledge Manager* component guarantees access to the knowledge and stores only newly generated knowledge; references to the original knowledge sources are used to avoid data redundancy. The *Knowledge Visualizer* component encompasses the key functionality of the dashboard.

## IV. Knowledge Visualization

In this section, we present our ideas and sketches for a dashboard to visualize knowledge related to CSE environments. We first give an overview and then present the visual components such as springboards and widgets in more detail.

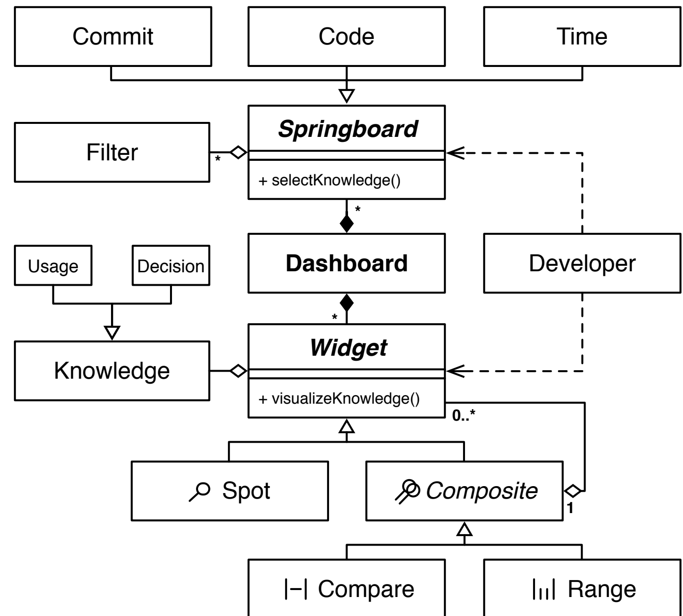


Fig. 2. The dashboard's class diagram depicting the main abstractions *Springboard* and *Widget*. *Filter* and *Knowledge* can have further sub-classes.

### A. Dashboard Overview

The dashboard contains two main abstractions: *Springboard*s and *Widget*s. The class diagram in Fig. 2 depicts the conceptual structure of both abstractions and their relationship to the participating role, namely the *Developer*.

A springboard allows developers to specify the scope of knowledge visualized in the widgets. Therefore, at least one springboard instance must be present in the dashboard at any time. Each springboard reflects the project from a different perspective. We propose three different springboard classes: *Commit*, *Code*, and *Time*. Each class has a specific relation to both usage and decision knowledge and therefore qualifies as a knowledge selector. However, the dashboard, as shown in Fig. 2, allows to add new springboard classes.

Based on the selection performed in a springboard, widgets update their content to visualize selection-specific knowledge. *Developer*s can add multiple widgets to the dashboard to fit their knowledge needs. There are three classes of widgets:

- **Spot** This class allows access to knowledge that is either related to a point in time, a version of an artifact, or an event such as a commit. This includes knowledge that needs to be aggregated over a period of time to reflect an information value, as long as it can be related to a reference point.
- **Compare** This class enables developers to contrast knowledge from two different dates or events. This supports the exploration of two alternative implementations or the discovery of an optimal solution when choosing between existing implementation alternatives.
- **Range** This class supports inspecting the evolution of knowledge either over a period of time or by aggregating the knowledge over a collection of events.

Definitive version available at https://doi.org/10.1109/VISSOFT.2017.18

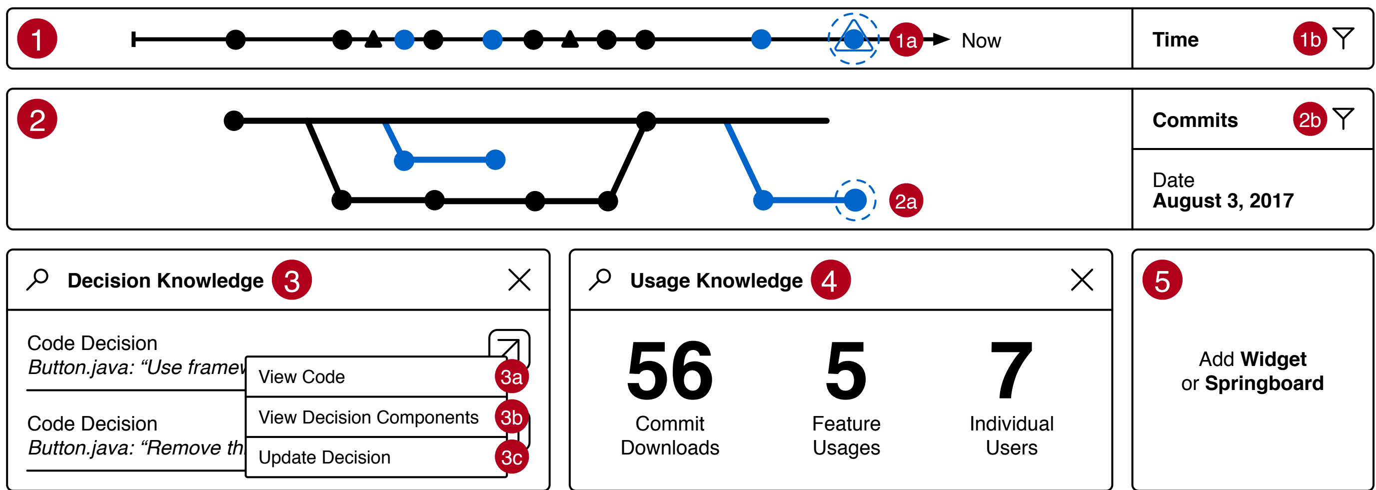


Fig. 3. Sketch of the dashboard showing a time springboard ① and a commit springboard ②. Both indicate that a commit is selected, depicted by a dashed circle ((1a) and (2a)). Additional filters can be applied to the springboards ((1b) and (2b)). Spot widgets for decision knowledge ③ and usage knowledge ④ show knowledge related to the selection. Further widgets and springboards can be added using a dedicated button ⑤.

As indicated in Fig. 2, both compare and range widgets are a composite of the spot widget, which represents a leaf widget. This enables the creation of complex widgets, such as comparing more than two alternatives at the same time. Moreover, for usage knowledge, we envision widgets to compare current usage knowledge with potential results after a change. This allows to assess the effect of changes without applying them.

Widgets are implemented independent of the knowledge source as described in Fig. 1. Every instance of a widget conforms to a defined interface visualizing specific knowledge, such as usage knowledge or decision knowledge. Additional sub-classes enable widgets to amplify this knowledge.

We suggest a grid-based layout for the dashboard as indicated in Fig. 3. The time springboard (Fig. 3-①) and commit springboard (Fig. 3-②) occupy fixed positions and cannot be removed. Additional springboards and widgets are hand-picked by developers from a list of available widgets and added to the dashboard (Fig. 3-⑤). Widgets can be moved and resized within the dashboard grid. This provides developers with the flexibility to adjust the dashboard to their needs.

We pose that the consideration of usage and decision knowledge benefits the development process and software quality. For instance, relating an increase in positive user feedback to an implementation proposal might accelerate the decision-making process. Likewise, a drop in a feature usage indicates users' disapproval, an important basis for decision-making towards the users' needs. The joint visualization facilitates the discovery and utilization of such relations.

### *B. Springboards*

Springboards are the main interaction components in the dashboard. They allow to specify the knowledge visualized in widgets, similar to the `grep` command in Linux. Springboards are distinguished by their perspective on the software project. We propose three springboards, namely a commit, code, and time springboard as depicted in Fig. 2, while additional springboards are possible. In addition, springboard filters provide the possibility to further detail the selection. For example, the springboard filter in Fig. 3-(2b) could be used to highlight commits that concern a particular decision.

We sketch the commit springboard in Fig. 3-②. Black lines and circles indicate existing branches and commits, respectively. Open branches and related commits are colored.

Clicking on a commit selects it as a reference point, indicated by a dashed circle around it (Fig. 4-①). This notifies all spot widgets to update their content accordingly. Clicking on a branch, in particular on a line between two commits, selects the branch as a period of time or a collection of events (Fig. 4-②). This notifies all range widgets to update their content accordingly. Multiple commits can be selected by pressing the option key (Fig. 4-③). This notifies all compare widgets to update their content accordingly.

The time springboard in Fig. 3-① works similar to the commit springboard. Commits are represented as circles and distributed horizontally along a timeline. However, in contrast to the commit springboard, the time springboard uses timestamps as the main viewpoint. Hence, it is able to select knowledge that is not introduced by or bound to a specific commit: black triangles represent decisions that were made at a specific point in time. In case decisions are documented in the commit message or as code comments, a triangle is added around the related commit circle as shown in Fig. 3-(1a).

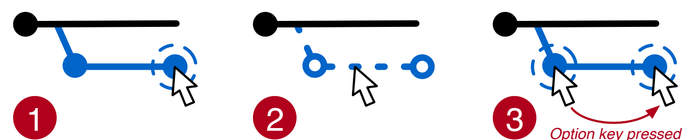


Fig. 4. Commit springboard showing ① one commit selected, ② range of commits selected, and ③ two commits selected.

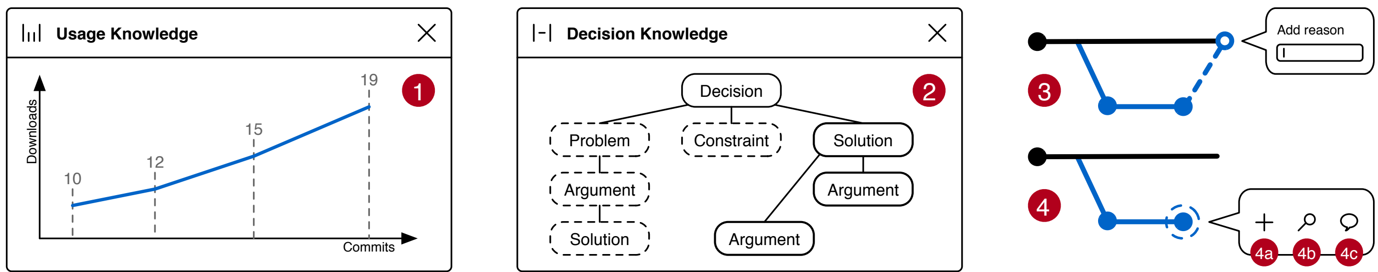


Fig. 5. ① Range usage knowledge widget showing commit downloads. ② Compare decision knowledge widget showing a decision at two different points in time. Dashed lines indicate earlier decision components. ③ Initiate a pull request by dragging a commit. ④ Trigger additional actions based on a commit.

### C. Usage Knowledge Widgets

Usage knowledge widgets visualize knowledge that relates to the usage of a software artifact. The dashboard allows to inspect both explicit and implicit feedback given by users.

Existing approaches to visualize explicit feedback can be integrated into the widgets: as introduced by Guzman *et al.*, a spot usage knowledge widget could provide a rating distribution or a sentiment distribution [4]. The dashboard extends their approach by allowing to select the reference point of the visualized knowledge using springboards.

CSE is characterized by its high frequency of iterations. Therefore, visualizing implicit user feedback is of great interest, since this kind of knowledge can be inferred directly from usage, without the need to wait for explicit feedback such as app reviews. As shown in Fig. 3-④, a spot usage knowledge widget provides insight into a commit, such as knowledge of the overall number of downloads, the number of feature usage, or individual users. This knowledge might reveal if a feature is adopted by users or not. Based on this insight and in accordance with decision knowledge presented in another widget, developers can reconsider implementation decisions.

Range usage knowledge widgets provide insight over time as shown in Fig. 5-①. If a branch is selected, its containing commits are rendered on the x-axis, while for example the number of commit downloads are shown on the y-axis.

### D. Decision Knowledge Widgets

Decision knowledge widgets visualize decisions from the software lifecycle, for instance regarding requirements or code. This knowledge is retrieved from various sources, such as issue trackers, code repositories, or wiki platforms. If required, developers are provided with tools for the respective knowledge source to record associations between individual artifacts.

Developers continuously reflect on decisions during CSE. For example, they might wonder which decisions preceded or influenced a commit. Decision knowledge widgets enable developers to spot such decisions (Fig. 3-③) and to navigate to the associated artifact (Fig. 3-(3a)). We base the visualization of decisions on the decision documentation model [8]: a decision contains any number of decision components such as rationale, the problem to be solved, alternatives, goals, or constraints. For further insight, they can be viewed in a spot widget, which is added to the dashboard by developers (Fig. 3-(3b)).

Developers might want to update decisions based on information retrieved from usage knowledge widgets. As shown in Fig. 3-(3c), they can replace a decision by an alternative or incrementally refine it by adding more decision components.

In Fig. 5-②, we sketch a compare widget to contrast decision components at two points in time. Dashed border lines indicate earlier documented decision components, whereas the solid border indicates those documented recently.

To better understand changes in decision components over time, developers utilize range widgets. For example, a range widget could visualize the evolution of decisions using spiral curves in which new decision components appear along this spiral as suggested by Zhi and Ruhe [12].

### E. Interaction

In addition to the visualization of existing knowledge, the dashboard provides the possibility to actively create, trigger, maintain, and relate knowledge visually.

We suggest that springboards are the main point for interaction. In Fig. 5, we provide examples for this interaction within the commit springboard. For instance, a *pull request*, i. e. the process of merging back code from a feature branch into the master branch, could be achieved by dragging a commit onto a branch as sketched in Fig. 5-③. A popup might ask for background information on merging the code in question. We do not intend to replace existing tools for these interactions. Instead, by providing an additional way to trigger these interactions, the dashboard allows developers to add knowledge in a convenient way. In particular, this facilitates the collection of knowledge and enhances its likelihood.

In Fig. 5-④, we illustrate actions that might be triggered from a commit: attach additional knowledge (Fig. 5-(4a)) or set up a new widget for a detailed analysis (Fig. 5-(4b)). We envision that knowledge attachment is supported by knowledge inference techniques. For example, the documentation of decisions concerning code changes can be supported by their automatic summarization as done by Linares-Vásquez *et al.* [13]. Furthermore, springboards can serve as an interaction point to end users: for instance, feedback can be proactively requested (Fig. 5-(4c)). Thereby, users can be invited to use a new build from a commit. At the same time, a new usage knowledge widget could be added to the dashboard that keeps track of the feedback results or visualizes the users' behavior.

## V. Discussion

We discuss the dashboard and its visualization approach on the basis of our four goals, namely to enable developers to *follow*, *reflect*, *interact*, and *react* on knowledge.

CSE is accompanied by activities that lead to constant change, which is reflected in branching strategies. The dashboard allows developers to visually follow code changes, especially the ones they have not been involved in.

An important goal of the dashboard is to allow the reflection on knowledge. By using springboards and widgets the developers are able to reflect on knowledge and knowledge relations. This enables them to understand the reasons for the current state of the software artifact. However, a widget's expressiveness depends on the quality of the widget's visualization. Therefore, we will investigate on metrics that capture the knowledge content of a widget. For instance, the number of visual elements required to express a widget's content could be considered for this purpose. Still, it is up to the developers' capability of discovering correlations between the knowledge visualized in the widgets. To support inexperienced developers, it might be helpful to allow experienced developers to link widgets that have proven to be valuable in joint usage.

Developers are encouraged to refine existing knowledge. We proposed first ideas to allow an intuitive and convenient way of interaction to connect and add new knowledge using the springboards. Still, we are investigating other concepts to enrich existing knowledge within the dashboard.

This leads to the goal of enabling developers to react to reports on knowledge gaps and missing links that were detected automatically. Our visualization proposals rely on structured knowledge sets. In real CSE environments, the data might not be well-structured: for instance, branch relations might become complex and difficult to understand [11]. We imagine utilizing automatic approaches that support developers to understand knowledge, detect missing knowledge elements, and apply a hybrid approach to resolve problems.

## VI. Current and Future Work

As part of a case study with practitioners from industry, we are currently conducting interviews to further investigate their needs on usage and decision knowledge. Based on these results, we will refine the concepts presented in this paper.

We plan to realize the dashboard on a prototypical basis and add support for various CSE tooling platforms. The prototype will be evaluated within a capstone course at the Technical University of Munich in which up to 12 development teams consisting of seven to ten students work on a software project that is given by an industrial customer over the period of one semester [14]. The evaluation will focus on the dashboard's feasibility given real development situations and the acceptance by developers. We will search for metrics that indicate the dashboard's support during the development process, for instance how often developers relied on a specific widget. We will gather developers' impressions—for instance regarding the ease of use or typical working routines—using a questionnaire at the end of the course for further improvement.

## VII. Conclusion

We introduced a dashboard that supports developers to better understand knowledge and knowledge relations in CSE environments. Therefore, we showed the main components. We proposed a structure for the dashboard and ideas for the visualization of the selection and presentation of knowledge. In particular, we introduced springboards as knowledge selectors and spot, compare, and range widgets for knowledge presentation. We showed visual instances of these widgets for usage and decision knowledge. The dashboard is extensible; additional classes of springboards and widgets can be added and interactions such as adding, modifying, or combining knowledge can be triggered. As part of our current and future work, we study further knowledge needs, investigate additional ideas, and evaluate a prototypical implementation of the dashboard in a case study.

## Acknowledgement

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution (CURES project).

## References

[1] J. Bosch, *Continuous Software Engineering: An Introduction*. Springer, 2014.

[2] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, 2015.

[3] J. O. Johanssen, A. Kleebaum, B. Bruegge, and B. Paech, "Towards a systematic approach to integrate usage and decision knowledge in continuous software engineering," in *Proceedings of the 2nd Workshop on Continuous Software Engineering*, 2017, pp. 7–11.

[4] E. Guzman, P. Bhuvanagiri, and B. Bruegge, "Fave: Visualizing user feedback for software evolution," in *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)*, 2014, pp. 167–171.

[5] L. Leiva, "Automatic web design refinements based on collective user behavior," in *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, 2012, pp. 1607–1612.

[6] L. Lee and P. Kruchten, "A tool to visualize architectural design decisions," in *Quality of Software Architectures. Models and Architectures: 4th International Conference on the Quality of Software-Architectures*, S. Becker, F. Plasil, and R. Reussner, Eds. Springer, 2008, pp. 43–54.

[7] M. Shahin, P. Liang, and M. R. Khayyambashi, "Improving understandability of architecture design through visualization of architectural design decision," in *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge - SHARK '10*, 2010, pp. 88–95.

[8] T.-M. Hesse, A. Kuehlwein, and T. Roehm, "Decdoc: A tool for documenting design decisions collaboratively and incrementally," in *Proceedings of the 1st International Workshop on Decision Making in Software ARCHitecture*, 2016, pp. 30–37.

[9] J., C. Anslow, and F. Maurer, "Information visualization for agile software development," in *2014 Second IEEE Working Conference on Software Visualization (VISSOFT)*, 2014, pp. 157–166.

[10] L. Alperowitz, D. Dzvonyar, and B. Bruegge, "Metrics in agile project courses," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 323–326.

[11] S. Elsen, "Visgi: Visualizing git branches," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[12] J. Zhi and G. Ruhe, "Devis: A tool for visualizing software document evolution," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–4.

[13] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *37th IEEE International Conference on Software Engineering*, 2015, pp. 709–712.

[14] B. Bruegge, S. Krusche, and L. Alperowitz, "Software engineering project courses with industrial clients," *ACM Transactions on Computing Education*, vol. 15, no. 4, pp. 17:1–17:31, 2015.

Definitive version available at https://doi.org/10.1109/VISSOFT.2017.18