DSDGen: Extracting Documentation to Comprehend Fork Merges

Jacob Krüger
Eindhoven University of Technology
Eindhoven, The Netherlands
j.kruger@tue.nl

Alex Mikulinski
Otto-von-Guericke University
Magdeburg, Germany

Sandro Schulze
Anhalt University of Applied Sciences
Köthen, Germany
sandro.schulze@hs-anhalt.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

Gunter Saake
Otto-von-Guericke University
Magdeburg, Germany
saake@ovgu.de

ABSTRACT

Developers use the forking mechanisms of modern social-coding platforms to evolve and maintain their systems. Using such mechanisms often leads to a larger number of independent variants with individual features or bug fixes that the developers may want to merge after a longer period of co-evolution. At this point, they may have forgotten (or never had) knowledge about differences between the variants. Tackling this problem, we built on the idea of on-demand documentation to develop a technique that automatically extracts and presents information for merging a class from two forks. We implemented our technique as a prototype called DSDGen and evaluated it through an experimental simulation with 10 students who should comprehend two real-world merge requests. Using DSDGen instead of code diffs only, more of the students could correctly comprehend the merges (6 / 10 versus 2 / 10) within a similar time. The students actively inspected the additional information provided by DSDGen and used it to comprehend the differences between the forked classes. So, DSDGen can help developers recover information for comprehending the differences caused by fork co-evolution during merges, with our results indicating opportunities for future research and improvements.

CCS CONCEPTS

• **Software and its engineering → Software version control**; **Software evolution**; **Maintaining software**.

KEYWORDS

program comprehension, software documentation, merging, fork ecosystems, software evolution, variant-rich systems

ACM Reference Format:
Jacob Krüger, Alex Mikulinski, Sandro Schulze, Thomas Leich, and Gunter Saake. 2023. DSDGen: Extracting Documentation to Comprehend Fork Merges. In *27th ACM International Systems and Software Product Line Conference - Volume B (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3579028.3609015

SPLC '23, August 28-September 1, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0092-7/23/08.
https://doi.org/10.1145/3579028.3609015

1 INTRODUCTION

Most of today's software systems are developed using version-control systems, such as Git, and social-coding platforms, such as GitHub [11, 44, 62]. Social-coding platforms extend the concepts of distributed collaboration and revisions in version-control systems by providing enhanced tooling and traceability, for instance, via issues or forks with pull requests. Developers use particularly the forking mechanisms (i.e., creating a copy of the system that an independent group of developers can evolve) to structure their development and manage its complexity. For instance, forks (or branches) are used for developing individual features, testing a system, or managing variant-rich systems [20, 24, 32, 35, 63, 76, 77].

After implementing their changes in a fork, the respective fork developers can create a pull request to ask the main developers to integrate these changes into the original system. While smaller changes are often continuously integrated and deployed, there are many use cases in which the integration becomes more challenging, for instance, longer co-evolution causing divergence between the main and forked system version [20, 22, 32, 39, 63, 65, 77]. Such cases occur when an organization is unsure whether an innovative product or feature fits its integrated platform and develops that product independently, or when open-source developers customize a fork to their needs. In such cases, the integration of a fork may occur weeks or even years after it was forked from the main system. Due to the long co-evolution of the main system and its fork, their differences may be significant, particularly if changes have not been properly communicated or cause side effects with other changes in the main system, for instance, because of feature interactions.

For merging, it is key to comprehend the differences between the main and forked system, which is an expensive and cognitively challenging activity when only source code is available and knowledge is missing [3, 12, 31, 33, 38–40, 45, 68, 72]. Unfortunately, other sources of information may be unavailable (e.g., developer who left [58]), hard to find (e.g., natural-language discussions in an unknown issue [5]), or mistrusted (e.g., outdated code comments [19, 50])—among many other issues [1]. This can easily cause problems when merging forks. For instance, Brindescu et al. [7] report that almost 20 % of the merges (≈6.5 % of 556,911 commits) in 143 open-source projects cause a conflict, with the respective code being twice as likely to involve a bug. In more than 75 % of these cases, a developer needs to inspect the code and it becomes 26 times more likely that a bug is involved. So, merging diverging versions of a system (forks) can easily become expensive and cognitively

challenging [3, 12, 29, 31, 39]. Specifically, McKee et al. [45] have found that program comprehension is the primary challenge for developers during merges; with getting and displaying the right information being two of the other top four challenges. This highlights the need for techniques that help developers collect, explore, and compare information of the different forks involved in a merge.

Inspired by the idea of on-demand software documentation [54], we have developed a technique to facilitate developers' comprehension during fork merges by recovering and displaying such information. We rely on different information sources (e.g., project data, pull requests, commits) that developers use to discuss, report, or document their changes. Using our prototype *Dynamic Software Documentation Generator* (DSDGen), we performed an evaluation based on an experimental simulation [64] with 10 student developers. The results indicate that DSDGen helped the students recover important information to support their program comprehension of the implemented changes, resulting in more students correctly answering our questions within a comparable time.

In detail, we contribute the following in this paper:

- We introduce DSDGen, a prototype for generating documentation to compare two forked variants of a class.
- We report an experimental simulation with which we evaluated DSDGen's impact on comprehending fork merges.
- We publish our prototype in an open-access repository.[1]

Based on our results, we argue that DSDGen can help developers (e.g., novices or those lacking knowledge about the forks), and provides a foundation for designing new techniques for creating on-demand documentation and supporting program comprehension when merging forks.

2 DSDGEN

In this section, we present our technique for extracting and displaying documentation about two forked files. We refer to *base* system as the one a *fork* is merged into—but base may also be another fork instead of the actual base system (e.g., to synchronize between forks).

2.1 Overview

Based on our motivation and existing research (cf. Section 1), we defined five requirements (Req) for our technique:

Req$_1$ *Documentation on class/file level:* We argue that a class has the appropriate size, scope, and complexity to provide information on-demand without overwhelming a developer compared to packages, modules, or subsystems.

Req$_2$ *Highlighting differences:* Our technique extracts and displays commonalities as well as differences to allow developers to easily identify and understand these.

Req$_3$ *Multiple data sources:* While source code is perceived the most reliable specification of software [50, 56, 68], further valuable sources of information exist in social-coding or other platforms (e.g., Stack Overflow if the project is prominent enough) [35, 37], which we integrate into our technique.

Req$_4$ *Dynamically extracted information:* To reflect the most recent state, we extract all pieces of information in an on-demand fashion to minimize the risk of missing important details or making wrong suggestions due to outdated information.

[1] https://doi.org/10.5281/zenodo.8132214

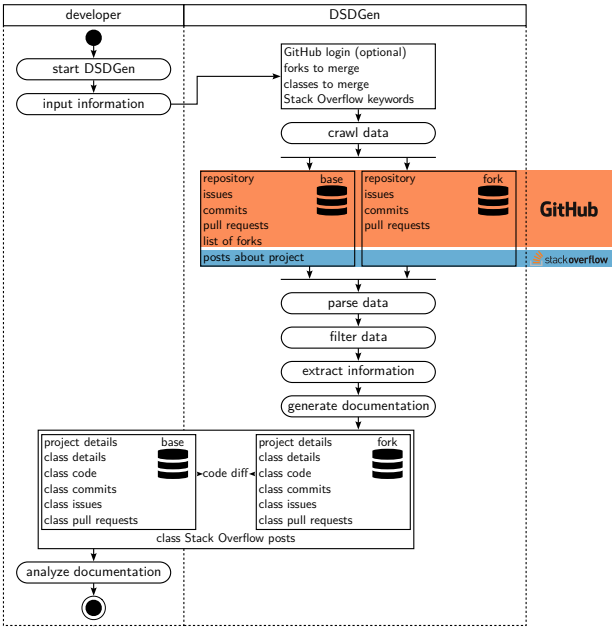

Figure 1: The overall workflow of DSDGen.

Req$_5$ *Graphical user interface:* To provide the information in an intuitive way, while also allowing a developer to explore it easily (e.g., scrolling, searching), we use a simple graphical user interface instead of a plain command line interface. For our prototype, we create the user interface as a separate document, but envision its integration into development tools (e.g., into IDEs) with further improvements in the future.

Guided by these requirements, we designed DSDGen.

In Figure 1, we display the general workflow of DSDGen. A developer starts DSDGen to access its graphical user interface, and inputs basic information, namely the base system, fork, and class for which the documentation shall be generated. Optionally, information for the filtering step can be provided (e.g., what information is needed). After crawling the corresponding data (cf. Section 2.2), our technique parses and filters the input to preprocess the raw data (cf. Section 2.3). Based on the preprocessed data, our technique extracts all information relevant for the class (cf. Section 2.4) and subsequently presents it to the developer (cf. Section 2.5).

2.2 Data Sources

For DSDGen, we decided to limit our scope to GitHub (social-coding platform) and Stack Overflow (community-question answering platform), since these two are the most popular for software development. In detail, we collect the following pieces of data (Req$_3$):

Commits are the most important data source besides the source code itself. They reflect how code has changed over time, allowing a developer to track, analyze, and compare the evolution in detail or to identify whom to contact for more information. Commit messages can contain valuable information about changes, for instance, whether a feature has been modified, added, or bug fixed [4, 35, 76].

Issues provide insights into the communication between developers [5, 35]. For instance, issues can involve bug reports or feature requests, and keep a history of the involved developers' discussion.

Pull Requests are used to request that changes are merged from a fork into the base system, and they may involve references to external documentation [75]. Understanding what pull requests have been submitted for and integrated into each fork allows developers to identify differences in the forks' co-evolution (e.g., bug fixes existing only in one of the two).

Forks are used to create a new variant of a software system, for example, to implement a new feature, fix a bug, or manage independent variants [15, 32, 63]. In parallel, forks can easily result in duplicate or redundant development efforts [52], which is valuable information for a developer.

Posts on community-question answering systems like Stack Overflow are used to discuss all types of questions on a topic [6, 28]. Some posts are related to prominent systems, and thus can be helpful for developers to better understand a change, for example, if it fixes a bug discussed or even solved on Stack Overflow [42, 70].

2.3 Crawling, Parsing, and Filtering

Once a developer has specified which of the previous sources they consider relevant, DSDGen downloads the respective data. Afterwards, DSDGen parses and filters the data, since we aim to present the documentation on class and not on system level (Req_1). For **forks**, DSDGen asks the developer to specify the fork they want to merge into their base system, if they did not provide this information in the beginning. Note that we support this scenario to allow any developer to inspect and pick any forks of their system even when there are no explicit merge requests. In contrast, this information is already provided when merging dedicated forks during a pull request. To support the former, we display a list of all existing forks for the developer to select relevant ones. We ease this inspection by allowing to filter out inactive forks or to filter based on the forks' descriptions (e.g., new feature, bug fix).

Then, DSDGen collects all **commits** of the base and fork system. So, we obtain all information on the common and diverging evolution of the two systems, together with what classes have been affected in each commit. Next, we filter out commits from either system that do not concern the specified class. DSDGen uses the class name to further filter for relevant **issues** and **pull requests**. Specifically, we collect all issues and pull requests that mention the specified class in their description, title, or comments. Using this filtering, we can enrich our documentation with additional information provided by developers, if they (correctly) mention the class in these fields. For instance, a bug fix reported in an issue may be linked to a different class, but the developers identified and discussed that the consequent changes would also impact the specified class.

For filtering **Stack Overflow posts**, we rely on the fact that popular systems have own tags on this platform to ensure the performance of DSDGen (i.e., filtering all data would require too many resources). Of course, this data source is limited to certain systems. From all relevant (i.e., tagged) posts, we identify any that mention the relevant class in their title or the discussion (i.e., questions, comments). If the class name is contained, we store the respective post together with a link to its discussion. However, this may lead to a vast amount of posts (e.g., if the class relates to an API). To manage this situation, DSDGen allows developers to further filter based on keywords, such as "bug" to find discussions about a certain bug or "example" to find code examples related to the class. Also, the developer can filter based on whether a post is closed or open, is answered or unanswered, and has a certain score.

2.4 Information Extraction

DSDGen extracts and displays (cf. Figure 2) the following information as concise as possible to avoid overwhelming developers [71]:

Project Information. We extract meta information about the base and fork system, namely links to their repositories, their descriptions, and the number of "watchers" as an indicator of relevance. Also, we collect the date each of the two systems has been created, last pushed, and last updated to indicate activity.

Class Information. For the class itself, we extract the fully qualified name and a description for the class, which is important to understand the context in which changes have been employed [14]. We assume that the class description is provided in the first comment at the top of the class (e.g., recommended by Oracle[2] and Google[3]); if such a comment exists.

Class Information. For each commit, we extract the date it has been committed (for sorting) as well as the message to help a developer understand the changes, and thus decide on the relevance of a commit. We also collect the link to the original GitHub commit as well as the name and email of the committer as specified.

Class Information. We extract the status of each issue and pull request, which allows a developer to focus on current problems and changes (i.e., the status is "open"). Additionally, we extract the title, assuming that it is shorter than an issue's or pull request's description, while still providing a valuable overview about what is being addressed. We collect a link to every issue and pull request on GitHub, enabling the developer to read the full description and comments if needed. Finally, we extract the labels (e.g., bug report, feature request) belonging to an issue, which typically provide more information about the nature of an issue.

Stack Overflow Posts. For Stack Overflow posts, we extract the date they have been posted to identify recency, their title as an indicator for the topic discussed, and their score to highlight received attention. To enable developers to inspect a post, we collect the link to each post on Stack Overflow.

2.5 Presentation

In Figure 2, we show the layout of the documentation (Req_5). Note that we use a simple, clearly structured layout to avoid confusion and clutter; and that we use a separate HTML document for our prototype, which is why there is no integration into development environments, yet. We decided to use a browser-based solution using HTML and JavaScript, as this does not require additional tooling and allows developers to reuse certain browser capabilities, such as searching. Moreover, we present the documentation about a class on one page, so that developers can intuitively jump between different pieces of information—similar to reading websites or books [53, 71].

The documentation involves three columns: On the left (①), we display a typical menu that allows developers to observe the general structure of the document and jump to dedicated sections. In the

[2] www.oracle.com/technical-resources/articles/java/javadoc-tool.html#styleguide
[3] https://google.github.io/styleguide/javaguide.html#s7-javadoc

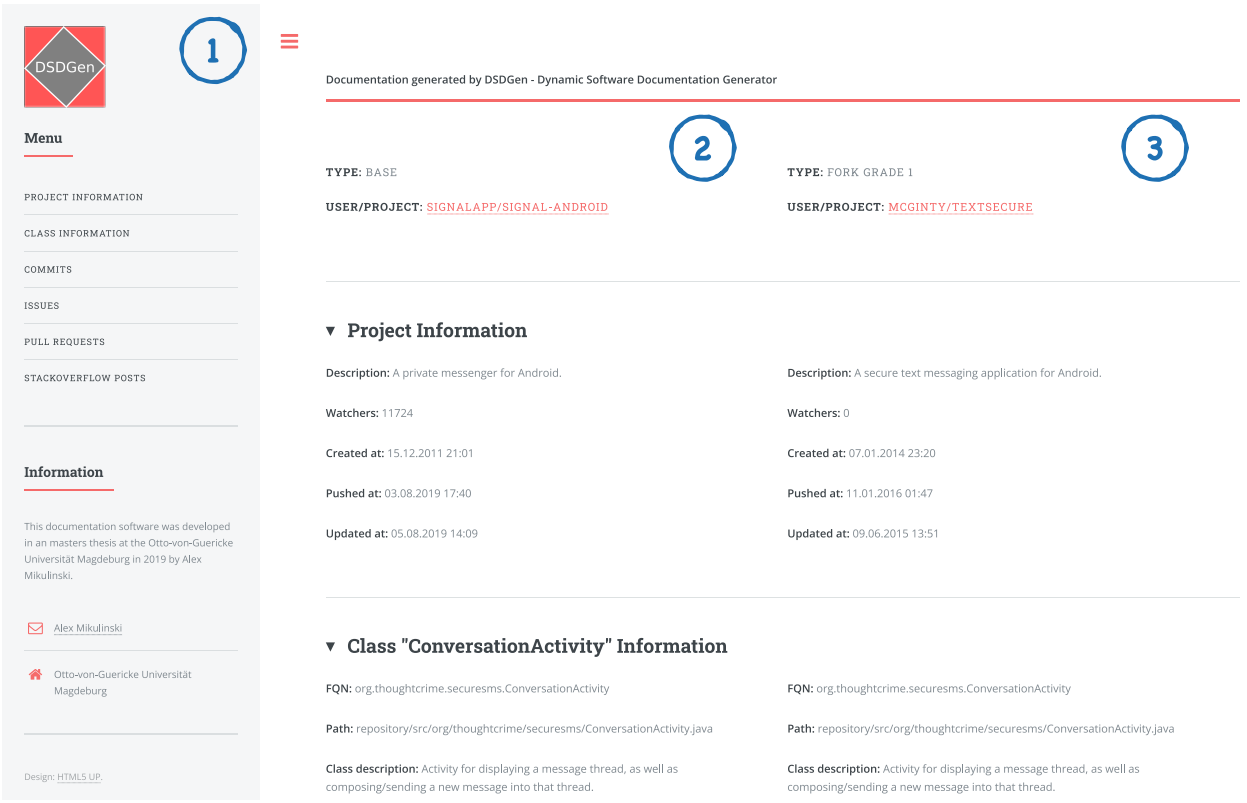

Figure 2: Snapshot of the documentation generated by DSDGen for the CA example of our simulation (cf. Section 3).

center (②), we display the information on the base system. On the right (③), we show the information on the fork. As we can see, we display the same types of information in a side-by-side two-column format (i.e., a developer compares ② with ③) to enable developers to easily grasp the commonalities and differences between the two system variants (Req_2). This view is inspired by the typical side-by-side diff view supported in development tools and on social-coding platforms. However, we argue that it is less distracting to provide all pieces of information together in one document. In each section, we summarize the extracted information, structured in the same fashion and order to facilitate their interpretation. For the section class information, we also display the source code of the two system variants individually and as a unified diff. Across all sections, we provide hyperlinks to the original sources (e.g., commits, issues, pull requests, Stack Overflow posts) to check details.

3 EXPERIMENTAL SIMULATION

We now report the design of our evaluation of DSDGen.

3.1 Research Questions

We evaluated whether the documentation by DSDGen helps comprehend forks and merges based on three research questions (RQs):

RQ_1 *Does DSDGen facilitate understanding a pull request?*
We analyzed whether our participants could understand the purpose of a pull request more easily using DSDGen compared to a plain diff view. So, we evaluate to what extent developers can become more effective in terms of *correctly solved tasks*.

RQ_2 *Does DSDGen facilitate analyzing a pull request?*
We analyzed how much time our participants needed to understand the purpose of a pull request. So, we evaluate their performance in terms of the *time required to solve tasks*. Note that, because our participants needed to familiarize with DSDGen and to inspect the documentation, we would not necessarily expect them to be faster compared to a diff view.

RQ_3 *Does DSDGen provide helpful information?*
We elicited our participants' perceptions regarding DSDGen and inspected our session recordings to see whether they used its additional information. So, we aimed to understand the perceived usefulness and limitations of our technique.

To answer these questions, we conducted an experimental simulation [64], combining quantitative (i.e., correctness for RQ_1, time for RQ_2) and qualitative data (i.e., perceptions and actual usage for RQ_3). We refrain from hypothesis testing, since the goal of our experimental simulation is to shed light into the usefulness of our technique and identify directions for improvements—not to verify explicit hypotheses. For this reason, we decided not to conduct a full-fledged controlled experiment and have a smaller number of participants; which does not allow us to perform statistical tests reliably.

3.2 Study Design

We used a *between-subject* design by half of our participants using DSDGen and the other half using a plain (unified) diff view for one of two examples. Between the examples, we switched the groups to avoid learning biases, while also gaining *within-subject* insights.

Table 1: Overview of the systems from which we selected our examples.

id	project	fork	link	forks	commits	issues	pull requests
CA	Signal	base	https://github.com/signalapp/Signal-Android	3,020	4,405	6,867	2,071
		fork	https://github.com/mcginty/TextSecure/tree/convo-ab-stateless	0	1,240	0	0
SLL	Algorithms	base	https://github.com/TheAlgorithms/Java	5,971	802	108	697
		fork	https://github.com/ojasiiitd/Java/tree/patch-2	0	741	0	0

Requirements for Examples. We aimed to use real-world pull requests from different systems and of varying complexity. This way, we intended to avoid biases, for instance, of participants becoming familiar with a system from which two pull requests stemmed, or too simplistic examples. To identify candidates, we analyzed GitHub projects in Java (due to our participants' background and our prototypical implementation analyzing Java only) and identified accepted pull requests. An accepted pull request ensures that the changes were deemed meaningful by the original developers, and we could use the code as well as additional comments of the merge for evaluating our participants' correctness. Consequently, we considered only pull requests that occurred in the past. During our simulation, we ensured that DSDGen provided only information that is older than the respective pull request to simulate the real-world scenario. For this purpose, we reset the main repository to the state of the pull request and considered only older information, also checking dates on issues, the pull-request comments, as well as potential Stack Overflow posts.

We defined six selection criteria for a pull request:

- The changes should not be too large in terms of lines of code to limit the time our participants need to analyze them.
- The changes should be part of a single class, since our prototype focuses on comparing classes and to avoid that participants need to switch between documents.
- The changes should modify (e.g., fix a bug) an existing class (i.e., not add a new one, remove one) so that it is necessary to understand the changes and their context.
- The system is actively maintained and information is available through additional sources (e.g., tags on Stack Overflow, issues) to exploit different sources.
- The system should have a larger number of forks (i.e., hundreds), commits, issues, and pull requests related to the changed class to ensure a larger data basis; simulating its use case of analyzing and merging co-evolving forks.
- The changes must be comprehensible without knowing details of the systems for our participants to work on them.

One problem that limited the number of candidates for our simulation was that the fork of the pull request had to still be available. In practice, developers may delete forks after they have been successfully merged, which does not allow us to use these forks. To identify candidates, we manually searched through Java projects on GitHub ordered by stars and the number of forks.

Selected Examples. We selected two examples based on our criteria. In Table 1, we display an overview of the base systems and forks from which these examples stem (at the point in time when we selected them). As we can see, they have substantial sizes in terms of commits, forks, issues, and pull requests, indicating active communities. Moreover, these systems match our initial use case of supporting developers in merging long-living forks. Unfortunately, while both systems are discussed on Stack Overflow, the classes involved in our particular examples are not. Thus, DSDGen could not retrieve information from this data source for our simulation. Nonetheless, the two examples represent the fork-analysis and merging scenario we envisioned, and are well-suited for our simulation.

First, we picked a bug-fixing pull request[4] from the Android version of the Signal App. The merged bug fix involves 39 changed lines in the method `initializeTitleBar()` that is part of the class `ConversationActivity`, and is responsible for displaying the correct conversation names (e.g., group chats). According to the pull request, the title of the conversation would sometimes not adapt according to the change of the conversation type. Besides fixing this bug, the changes simplify and refactor the code to improve readability. Consequently, the actual purpose of the pull request (fixing the bug) is not apparent when looking only at the code, and thus is complex to understand. DSDGen identified 22 relevant issues, eight pull requests, and 161 commits for the class in the base system (162 in the fork, due to the pull request). *We assessed a participant to correctly comprehend this example if they were able to identify the fixed bug.* In the following, we refer to this example via its id "CA," and consider it to be the more complicated one.

Second, we picked a code-improvement pull request[5] from the Java algorithms collection. The collection provides various data structures and algorithms. We picked a pull request that improves the `SinglyLinkedList` implementation by removing a global variable and introducing an alternative method to return the list's size. This change should reduce the error proneness of the code. Moreover, the pull request improves the performance of inserting or removing an element at the beginning, removed an unneeded temporary code element, and introduced a method for removing an element at a certain position. While there are a few changes in this pull request, they are rather simple and easy to identify, which is why we consider this example to be the easier one. For the base system, DSDGen identified two issues, six pull requests, and 13 commits (14 in the fork) that are connected to the class. *We assessed a participant to correctly comprehend this example if they were able to identify that the size computation was changed for what reason, the newly introduced method, and the performance improvement.* Consequently, a correct solution depends on three issues, which compensates for the more simplistic code example. In the following, we refer to this example via its id "SLL."

Questions. During our simulation, we asked each participant the 18 questions we display in Table 2. To address RQ_1, we defined three questions aimed at measuring their program comprehension of each example. With these questions, we checked whether a participant understood the functionality of the changed class (C_1), why the pull request was created (C_2), and the implemented change (C_3).

[4] https://github.com/signalapp/Signal-Android/pull/2297
[5] https://github.com/TheAlgorithms/Java/pull/713

Table 2: Overview of the questions we used in our simulation.

id	question	answer
Regarding correctness (RQ_1) and performance (RQ_2) for each example		
C_1	What functionality has the changed class/method?	free text
C_2	What was the purpose of the pull request?	free text
C_3	What does the change implement?	free text
Regarding the participants' perceptions (RQ_3)		
P_1	Did you have any comprehension problems during the experiment?	1 (no problems) – 10 (significant problems)
P_2	What do you think about the concept of DSDGen?	1 (very good) – 10 (very bad)
P_3	What do you think about the usability of DSDGen?	1 (very good) – 10 (very bad)
P_4	How useful was the additional information?	1 (very useful) – 10 (not useful)
P_5	What was your strategy to comprehend the code?	free text
P_6	What was (not) helpful about DSDGen?	free text
P_7	What information was missing?	free text
P_8	For what purposes would DSDGen be useful?	free text
P_9	Any other comments?	free text
Regarding the participants' background		
B_1	How do you rate your programming experience?	1 (experienced) – 10 (inexperienced)
B_2	How do you rate your experience with Java?	1 (experienced) – 10 (inexperienced)
B_3	How do you rate your experience of using a version-control system?	1 (experienced) – 10 (inexperienced)
B_4	For how many years have you been programming?	number
B_5	For how many years have you been programming professionally?	number
B_6	What is your highest degree of education?	free text

Answering these questions requires that a participant comprehends the code itself as well as the diff, and can explain them in their own words. For RQ_2, we measured the time until a participant considered their answers to these three questions complete.

After they worked on the two examples using DSDGen and a standard diff view, we asked our participants about their perceptions (RQ_3). First, we checked whether they had any problems understanding our simulation (P_1) to check for threats to the validity of our results. Then, we asked them to rate the overall idea (P_2), usability (P_3), and information provisioning (P_4) of DSDGen. Moreover, we asked each participant to elaborate on their strategy for comprehending the examples (P_5), the usefulness of DSDGen (P_6), missing information (P_7), and relevant use cases (P_8). Via these questions, we aimed to identify whether our technique was helpful and what potential improvements exist. In the end, we asked for any other comments the participants may have had (P_9).

In the end, we elicited background information for each participant, following established guidelines [60]. Specifically, we first asked our participants to self-evaluate their programming experience (B_1), knowledge about Java (B_2), and familiarity with version-control systems (B_3) to understand our participants' composition in terms of experience. Then, we asked for how many years they have been programming in general (B_4) and professionally, for instance, in a company or on established open-source projects (B_5). Lastly, we asked for their highest degree of education (B_6).

Audio Recording. To understand how the participants used DSDGen (RQ_3), we audio recorded each session. Note that we did not have the equipment for screen or video recordings in a proper quality. We informed the participants when inviting them to a session that these would be recorded (they could decline without any consequences) and asked each one to think-aloud [59] while working on the examples. The first author of this paper listened to each recording and coded labels with time stamps. Via open coding and open-card sorting to synthesize codes, we defined eight activities:

- Inspection of DSDGen itself (e.g., scrolling through it for familiarizing, exploring general system data).
- Inspection of pull requests via DSDGen as well as the linked information (e.g., description, comments, date).
- Inspection of issues via DSDGen (e.g., status, discussion).
- Inspection of commits via DSDGen (e.g., iterating through the list, reading messages, dates).
- Inspection of code via DSDGen or the diff view (i.e., reading the plain class code or the diff).
- Inspection of unclear elements via DSDGen (two occasions in which we could not reliable assign a category, due to participants not speaking for a longer time while scrolling).
- Discussions with the experimenter (e.g., answering a question about DSDGen or the experiment, feedback).
- Documenting solutions (e.g., typing answers to the questions or comments, reading the questions).

Our activities abstract from highly specific instances, and were quite good to identify. The first author relied on the participants' think alouds (examples in parentheses) as well as other recorded sounds to identify activities and switches (e.g., scrolling, longer typing). We argue that, while not fully detailed, the abstract activities are useful and provide reliable insights into the use of DSDGen.

Assignment. We randomly assigned each of our ten participants into one of two groups: The first group worked on the first example (CA) using the plain diff and on the second example (SLL) using DSDGen. The second group had the flipped setup, namely DSDGen for the first and the diff for the second example. This design allowed each participant to work with DSDGen and the diff view without introducing bias due to learning or differences in the examples.

3.3 Conduct

We conducted individual sessions with each participant, supervised by the second author to clarify any questions a participant may have had. Following guidelines on experiments with human participants,

we performed a short discussion on each participant's performance after a session [26]. In detail, we

(1) welcomed each participant (i.e., purpose of the simulation; setup; DSDGen; how to navigate on a small toy example; terminology; task; consent),
(2) conducted the session with both examples (i.e., we provided the general context for each example; the participant inspected each example until they answered C_{1-3} in Table 2),
(3) asked our remaining survey questions (cf. Table 2), and
(4) performed the follow-up discussion.

Depending on the group and example, we displayed either the respective pull request on GitHub or the HTML document generated by DSDGen in a web browser. We allowed the participants to use the browser's search functionality and to inspect all links provided by DSDGen or GitHub. On a second screen, they could see the questions they had to answer and could enter their answers. Using this design, we aimed to mitigate comprehension problems and avoid that a participant may misunderstand something. We stopped the required time for each example, but did not tell the participants about this to not cause stress.

3.4 Participants

We recruited ten volunteering computer-science students with practice experience via personal contacts of the second author (convenience sample). Instead of professional developers, we relied on students due to availability and because they are feasible participants in software engineering [18, 23, 55, 66]. Also, students mimic our target audience: Developers that are new to a system (or fork) or even novices in software development, and thus may benefit more from additional documentation.

Our participants (cf. Table 3) had an average of 7.9 years of programming experience, with 2 years of professional programming knowledge. Two students were undergraduates, six had a Bachelor degree, and two a Master degree. The students rated their experience with Java at around 4.2 and with version-control at around 4.1. They indicated only little comprehension problems. When discussing these problems, we found that they were mainly related to the subject source code, not the study design. Reflecting upon their experience, we argue that our participants represent a reliable sample, seeing their extensive programming experience.

4 RESULTS AND DISCUSSION

Next, we report the results of our simulation and discuss their implications. We display visualizations of the recordings of each session in Figure 3 (first example) and Figure 4 (second example). In these, we highlight the different activities and the periods they cover in the recordings (each pixel represents one second). At the end of each bar, we show whether the respective participant could answer the comprehension questions correctly (C_{1-3} in Table 2), and how much time they needed. We can already see that our assumption of the first example (CA) being more complicated than the second one (SLL) is represented in the results: Fewer participants have been able to correctly understand the pull request (2 versus 6), and they used more time (2:15 mins on average). Moreover, we can see that the individual participant's performance changed between the examples, indicating that any changes we can see are likely caused by the treatment (i.e., DSDGen or diff) rather than the participant's experience. So, we argue that we designed a feasible setup to study the impact of DSDGen on our participants' program comprehension.

Table 3: Our participants' self-assessments (B_{1-3}) and experiences (B_{4-5}) based on the questions in Table 2.

	question				
part.	B_1 (prog. L)	B_2 (Java L)	B_3 (VCSs L)	B_4 (prog. Y)	B_5 (prof. Y)
1	6	4	7	5	0
2	2	2	4	10	5
3	6	4	4	5	0
4	3	4	4	6	3
5	4	7	2	5	0
6	1	1	1	10	4
7	3	2	3	12	3
8	5	7	5	15	0
9	5	3	3	6	3
10	7	8	8	5	2
mean	4.2	4.2	4.1	7.9	2

participant, **prog**ramming experience, **prof**essional experience
Likert scale, **Y**ears

***RQ1*: Correctness.** We can see for both examples that more participants have been able to comprehend the differences between the two forks using DSDGen. In the first, more complex, example (cf. Figure 3), none of the participants could correctly comprehend the differences using the diff view only. However, using DSDGen, two of the participants were able to provide correct answers. In the second example (cf. Figure 4), the number of correct solutions increased from two (diff) to four (DSDGen). This supports that our technique can help developers when comprehending the differences between forks, and thus can facilitate fork merges.

> *RQ1*: Correctness
> Using DSDGen, more of our participants could comprehend the two examples correctly (i.e., 6 / 10 versus 2 / 10).

***RQ2*: Performance.** We can see that the participants using DSDGen took on average ≈2:21 mins more to solve their tasks. During the sessions, we found that this additional time is a result of getting used to DSDGen (i.e., yellow coloring in Figure 3 and Figure 4), inspecting the additional information and tool views (blue, green, and black), and a more careful analysis. So, some of the time differences may be caused by additional attention and effort of the participants rather than DSDGen itself. As noted, these were the particular reasons for which we did not necessarily expect faster solutions from our participants. Arguably, by getting more used to DSDGen, the additional time required will reduce. Since the time differences are not large, and DSDGen leads to correct solutions more often than the diff view, we do not perceive the worse performance as a negative result. Still, we clearly see potential to improve DSDGen, particularly to facilitate its use and decrease the time needed to familiarize.

> *RQ2*: Performance
> Our participants needed more time for the examples using DSDGen (2:21 mins), due to getting familiar with the documentation.

***RQ3*: Perception.** When asking for qualitative feedback (P_{1-9} in Table 2), most of our participants liked the concept of DSDGen, with

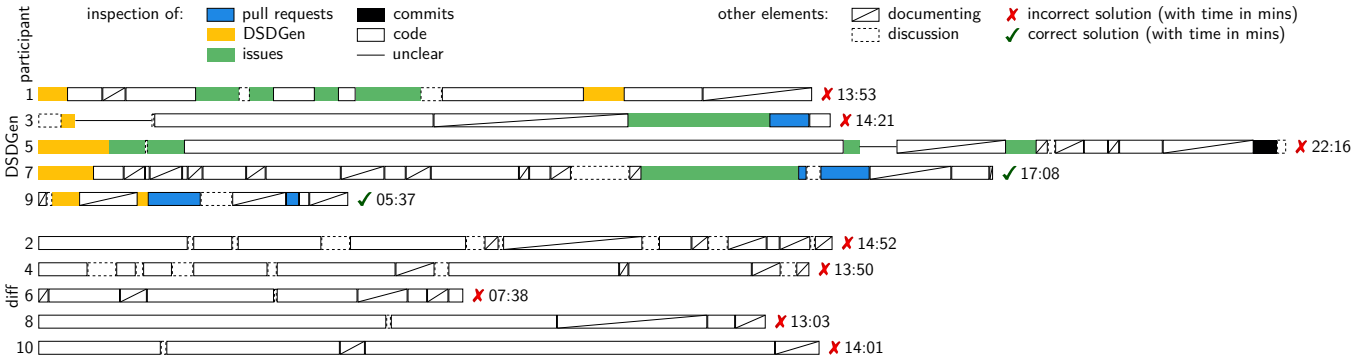

Figure 3: Results and coded recordings for the example CA in our simulation.

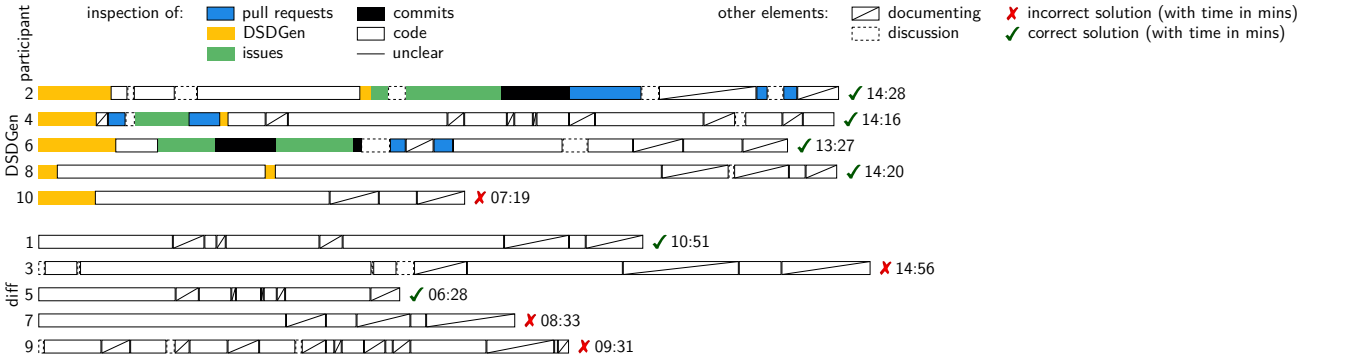

Figure 4: Results and coded recordings for the example SLL in our simulation.

an average scoring of 1.3 on a scale of 1 (very good) to 10 (very bad). Since we implemented only a prototype, DSDGen currently is not a polished or user-friendly tool. This is reflected by the more varying scores regarding the usability of DSDGen, which is on average 2.9 on the same scale. For the usefulness of the information DSDGen provides in comparison to the plain diff, our participants provided an average rating of 2.2. Note that the few participants who mostly ignored the additional information (i.e., participants 8 and 10, see Figure 4), graded this question based on their retrospective of the problems they faced, and potential benefits the information would have had to resolve these problems.

As **additional information** (for DSDGen, but also in general) that would have been useful, our participants mentioned: (1) line numbers for every code excerpt, (2) displaying omitted lines of code in all diff views, (3) the time when a pull request was created, (4) links between issues and corresponding pull requests, (5) displaying the Git history, and (6) visualizing when the forks started to differ (e.g., highlighting the information). Most of these requests are simple extensions and rather an engineering than a scientific problem, but we plan to implement them in the future. As particularly **useful information** provided in DSDGen, our participants mentioned: (1) links to the GitHub repositories, (2) additional meta information, (3) visualization on a single page, (4) an interactive user interface, (5) displaying multiple forks next to each other, and (6) the visuals of the prototype itself. This feedback improves our confidence that DSDGen is a helpful technique for developers and that we tackled our requirements (cf. Section 2.1). As **use cases** in which DSDGen would be useful, our participants mentioned various scenarios in the context of fork merging: (1) analyzing systems with many pull requests, (2) support for handling poor project management or missing documentation, (3) getting a better overview over many commits and additional pieces of information, and (4) analyzing a class and its commit history.

RQ3: Perception

DSDGen received mainly positive feedback and was perceived helpful for fork merging, with most issues relating to extensions that can be added easily.

Discussion. In Figure 3 and Figure 4, we can see that most participants inspected the additional information provided by DSDGen. By analyzing the recordings, we found that they often inspected a certain piece of information, matched it to the code, and thereby improved their comprehension. Of course, any difference could be understood through the code if needed, but our participants often found relevant information much easier in DSDGen. As an extreme example, participant 9 basically used DSDGen only when inspecting our example CA (cf. Figure 3). They immediately identified that the documentation provided and linked to the information needed to comprehend the differences, and only skimmed the code for confirmation—leading to the shortest analysis time overall and a correct solution.

Our participants inspected the systems' issues in the greatest detail, followed by pull requests, and occasionally commits. Two participants did not use any of this information when analyzing the second example with DSDGen (participants 8 and 10 in Figure 4)—one

of them being the only participant using DSDGen and not answering our questions correctly for this example. Seeing the results and reflecting upon our participants' perceptions, it seems that particularly the high-level change summaries in issues and pull requests are helpful to facilitate fork comprehension. Nonetheless, we can also see that our participants generally tended to focus on comprehending the source code; potentially because they mistrust other pieces of (potentially not maintained [19, 50]) information. Overall, we argue that DSDGen is helpful for developers despite such concerns.

Threats to Validity. Since we are concerned with program comprehension, we cannot control all variables related to the individual characteristics of our participants [48, 61]. For instance, developers in the real world rely on various tools that may infer with DSDGen and some cognitive biases may not manifest in a laboratory setting. Instead, we aimed to avoid control and confounding variables by defining a controlled setup, and thus to improve our confidence that any observed differences are caused by our intervention (i.e., DSDGen). However, this also means that our simulation is not fully representative of the real world, for instance, because our examples did not involve added/deleted or changes to multiple classes.

We relied on student participants, who are typically considered to perform similar to developers in practice [18, 23, 55, 66]. Still, practitioners with more experience about the programming language, version-control systems (particularly diffs), and the actual subject system may have yielded different results. Moreover, the students have been personal contacts of the second author, which may mean that their perceptions of DSDGen (e.g., P_2, P_3) could be more positive. We mitigated this threat by involving more qualitative questions and measuring the participants' actual performance, which hint in the same direction as their responses. While both threats regarding our participants remain, we could clearly show that DSDGen helped the students understand the subject source code, and similar improvements are likely for less experienced or novice developers (e.g., during on-boarding).

5 RELATED WORK

There is extensive research showing how forks (co-)evolve and that it can take a lot of time until they are merged [32, 63, 65, 76, 77]. Moreover, different researchers have focused on the challenges developers face when merging forks [21, 22], particularly merge conflicts [7, 8, 45]. These studies highlight the comprehension problems fork merges can cause, and motivate the need for our technique.

The idea of on-demand documentation has been proposed by Robillard et al. [54], and there have been many proposals for it. Most of these focus on a single class and analyzing its source code or evolution [2, 13, 41, 49, 67]. Closely related, researchers have worked on integrated software documentation [10, 25, 47]. Finally, visualizations for the differences between complete software variants have been proposed in product-line engineering [9, 16, 36, 43, 46]. None of such techniques is concerned with our use case, and thus we contribute a complementary technique.

Similarly, reverse-engineering techniques have been proposed, but they typically go beyond recovering documentation and aim to create new artifacts instead, for instance, in the form of class diagrams or API usage summaries [17, 27, 30, 51, 57, 69, 73, 74]. Most of these techniques employ static or dynamic program analysis to create novel documentation artifacts. We complement the existing works by providing a means for recovering and displaying documentation that can be further expanded with such techniques.

6 CONCLUSION

Analyzing and merging forks is a challenging process. We built on the idea of on-demand documentation to design DSDGen, which extracts information from various sources and provides a comparative overview to support developers when understanding fork differences. Our results show that more participants could comprehend two different fork merges (6 / 10 versus 2 / 10) using DSDGen, but they required more time (2:21 mins on average) due to familiarizing with the additional documentation. The overall feedback on DSDGen was positive and indicates that it is a helpful technique, which we plan to extend based on that feedback in future work.

ACKNOWLEDGMENTS

The research reported in this paper has been partially supported by the German Research Foundation (DFG) project EXPLANT (LE 3382/2-3, SA 465/49-3) [34].

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. 2020. Software Documentation: The Practitioners' Perspective. In *ICSE*. ACM.
[2] Alireza Aghamohammadi, Maliheh Izadi, and Abbas Heydarnoori. 2020. Generating Summaries for Methods of Event-Driven Programs: An Android Case Study. *Journal of Systems and Software* 170 (2020).
[3] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *SPLC*. ACM.
[4] Khadijah Al Safwan and Francisco Servant. 2019. Decomposing the Rationale of Code Commits: The Software Developer's Perspective. In *ESEC/FSE*. ACM.
[5] Deeksha Arya, Wenting Wang, Jin L. C. Guo, and Jinghui Cheng. 2019. Analysis and Detection of Information Types of Open Source Software Issue Discussions. In *ICSE*. IEEE.
[6] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. 2014. What are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow. *Empirical Software Engineering* 19, 3 (2014).
[7] Caius Brindescu, Iftekhar Ahmed, Carlos Jensen, and Anita Sarma. 2020. An Empirical Investigation into Merge Conflicts and Their Effect on Software Quality. *Empirical Software Engineering* 25, 1 (2020).
[8] Caius Brindescu, Yenifer Ramirez, Anita Sarma, and Carlos Jensen. 2020. Lifting the Curtain on Merge Conflict Resolution: A Sensemaking Perspective. In *ICSME*. IEEE.
[9] Siyue Chen, Loek Cleophas, and Jacob Krüger. 2023. A Comparison of Visualization Concepts and Tools for Variant-Rich System Engineering. In *SPLC*. ACM.
[10] Sridhar Chimalakonda and Akhila S. M. Venigalla. 2020. Software Documentation and Augmented Reality: Love or Arranged Marriage?. In *ESEC/FSE*. ACM.
[11] Valerio Cosentino, Javier Luis, and Jordi Cabot. 2016. Findings from GitHub: Methods, Datasets and Limitations. In *MSR*. ACM.
[12] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *SPLC*. ACM.
[13] Michael J. Decker, Christian D. Newman, Michael L. Collard, Drew T. Guarnera, and Jonathan I. Maletic. 2018. A Timeline Summarization of Code Changes. In *DySDoc*. IEEE.
[14] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *ICSE*. IEEE.
[15] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*. IEEE.
[16] Slawomir Duszynski. 2010. Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices. In *SPLC*. Springer.
[17] Sascha El-Sharkawy, Saura J. Dhar, Adam Krafczyk, Slawomir Duszynski, Tobias Beichter, and Klaus Schmid. 2018. Reverse Engineering Variability in an Industrial Product Line: Observations and Lessons Learned. In *SPLC*. ACM.

[18] Davide Falessi, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo. 2017. Empirical Software Engineering Experts on the Use of Students and Professionals in Experiments. *Empirical Software Engineering* 23, 1 (2017).

[19] Beat Fluri, Michael Würsch, and Harald C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *WCRE*. IEEE.

[20] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-Based Software Development Model. In *ICSE*. ACM.

[21] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *ICSE*. ACM.

[22] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *ICSE*. IEEE.

[23] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering* 5, 3 (2000).

[24] Jing Jiang, David Lo, Jiahuan He, Xin Xia, Pavneet S. Kochhar, and Li Zhang. 2017. Why and How Developers Fork What from Whom in GitHub. *Empirical Software Engineering* 22, 1 (2017).

[25] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel R. V. Chaudron. 2020. Software Engineering Whispers: The Effect of Textual vs. Graphical Software Design Descriptions on Software Design Communication. *Empirical Software Engineering* 25, 6 (2020).

[26] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering* 20, 1 (2015).

[27] Sebastian Krieter, Jacob Krüger, Thomas Leich, and Gunter Saake. 2023. VariantInc: Automatically Pruning and Integrating Versioned Software Variants. In *SPLC*. ACM.

[28] Jacob Krüger. 2019. Are You Talking about Software Product Lines? An Analysis of Developer Communities. In *VaMoS*. ACM.

[29] Jacob Krüger. 2021. *Understanding the Re-Engineering of Variant-Rich Systems: An Empirical Work on Economics, Knowledge, Traceability, and Practices*. Ph. D. Dissertation. Otto-von-Guericke University Magdeburg.

[30] Jacob Krüger, Mustafa Al-Hajjaji, Sandro Schulze, Gunter Saake, and Thomas Leich. 2018. Towards Automated Test Refactoring for Software Product Lines. In *SPLC*. ACM.

[31] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *VaMoS*. ACM.

[32] Jacob Krüger and Thorsten Berger. 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *ESEC/FSE*. ACM.

[33] Jacob Krüger and Regina Hebig. 2020. What Developers (Care to) Recall: An Interview Survey on Smaller Systems. In *ICSME*. IEEE.

[34] Jacob Krüger, Sebastian Krieter, Gunter Saake, and Thomas Leich. 2020. EXtracting Product Lines from vAriaNTs (EXPLANT). In *VaMoS*. ACM.

[35] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is My Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019).

[36] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *SPLC*. ACM.

[37] Jacob Krüger, Sebastian Nielebock, and Robert Heumüller. 2020. How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In *EASE*. ACM.

[38] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *ICSE*. ACM.

[39] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-And-Own: Moving to a Software Product Line for Temperature Monitoring. In *SPLC*. ACM.

[40] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *ICSE*. ACM.

[41] Mingwei Liu, Xin Peng, Xiujie Meng, Huanjun Xu, Shuangshuang Xing, Xin Wang, Yang Liu, and Gang Lv. 2020. Source Code based On-Demand Class Documentation Generation. In *ICSME*. IEEE.

[42] Xuliang Liu and Hao Zhong. 2018. Mining StackOverflow for Program Repair. In *SANER*. IEEE.

[43] Roberto E. Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2016. Visualization for Software Product Lines: A Systematic Mapping Study. In *VISSOFT*. IEEE.

[44] Panagiotis Louridas. 2006. Version Control. *IEEE Software* 23, 1 (2006).

[45] Shane McKee, Nicholas Nelson, Anita Sarma, and Danny Dig. 2017. Software Practitioner Perspectives on Merge Conflicts and Resolutions. In *ICSME*. IEEE.

[46] Raul Medeiros, Jabier Martinez, Oscar Díaz, and Jean-Rémy Falleri. 2022. Visualizations for the Evolution of Variant-Rich Systems: A Systematic Mapping Study. *Information and Software Technology* (2022).

[47] Sahar Mehrpour, Thomas D. LaToza, and Rahul K. Kindi. 2019. Active Documentation: Helping Developers Follow Design Decisions. In *VL/HCC*. IEEE.

[48] Rahul Mohanani, Iflaah Salman, Burak Turhan, Pilar Rodriguez, and D. Paul Ralph. 2020. Cognitive Biases in Software Engineering: A Systematic Mapping Study. *IEEE Transactions on Software Engineering* (2020).

[49] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. 2018. Detecting and Summarizing GUI Changes in Evolving Mobile Apps. In *ASE*. ACM.

[50] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting Source Code: Is It Worth It for Small Programming Tasks? *Empirical Software Engineering* 24, 3 (2019).

[51] Kristian Nybom, Adnan Ashraf, and Ivan Porres. 2018. A Systematic Mapping Study on API Documentation Generation Approaches. In *SEAA*. IEEE.

[52] Luyao Ren, Shurui Zhou, Christian Kästner, and Andrzej Wąsowski. 2019. Identifying Redundancies in Fork-Based Development. In *SANER*. IEEE.

[53] Martin P. Robillard and Robert DeLine. 2011. A Field Study of API Learning Obstacles. *Empirical Software Engineering* 16, 6 (2011).

[54] Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil A. Ernst, Marco A. Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. 2017. On-Demand Developer Documentation. In *ICSME*. IEEE.

[55] Per Runeson. 2003. Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data. In *EASE*.

[56] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. 2017. Comprehending Studies on Program Comprehension. In *ICPC*. IEEE.

[57] Sandro Schulze, Jacob Krüger, and Johannes Wünsche. 2022. Towards Developer Support for Merging Forked Test Cases. In *SPLC*. ACM.

[58] Todd Sedano, D. Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *ICSE*. IEEE.

[59] Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.). 2008. *Guide to Advanced Empirical Software Engineering*. Springer.

[60] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and Modeling Programming Experience. *Empirical Software Engineering* 19, 5 (2014).

[61] Janet Siegmund and Jana Schumann. 2015. Confounding Parameters on Program Comprehension: A Literature Survey. *Empirical Software Engineering* 20, 4 (2015).

[62] Diomidis Spinellis. 2005. Version Control Systems. *IEEE Software* 22, 5 (2005).

[63] Ștefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*. IEEE.

[64] Klaas-Jan Stol and Brian Fitzgerald. 2020. Guidelines for Conducting Software Engineering Research. In *Contemporary Empirical Methods in Software Engineering*. Springer.

[65] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*. ACM.

[66] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. 2008. Using Students as Subjects – An Empirical Evaluation. In *ESEM*. ACM.

[67] Ahmed Tamrawi, Sharwan Ram, Payas Awadhutkar, Benjamin Holland, Ganesh R. Santhanam, and Suresh Kothari. 2018. DynaDoc: Automated On-Demand Context-Specific Documentation. In *DySDoc*. IEEE.

[68] Rebecca Tiarks. 2011. What Maintenance Programmers Really Do: An Observational Study. In *WSR*. University of Siegen.

[69] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. 2007. Empirical Studies in Reverse Engineering: State of the Art and Future Trends. *Empirical Software Engineering* 12, 5 (2007).

[70] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *ICSE*. ACM.

[71] Gias Uddin and Martin P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (2015).

[72] Anneliese von Mayrhauser and A. Marie Vans. 1995. Program Comprehension During Software Maintenance and Evolution. *Computer* 28, 8 (1995).

[73] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. 2004. A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction. In *WCRE*. IEEE.

[74] Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, Alexei Lapouchnian, and Julio C. S. do Prado Leite. 2005. Reverse Engineering Goal Models from Legacy Code. In *RE*. IEEE.

[75] Fiorella Zampetti, Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, and Michele Lanza. 2017. How Developers Document Pull Requests with External References. In *ICPC*. IEEE.

[76] Shurui Zhou, Ștefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *ICSE*. ACM.

[77] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How Has Forking Changed in the Last 20 Years? A Study of Hard Forks on GitHub. In *ICSE*. ACM.