Collaborative, Code-Proximal Dynamic Software Visualization within Code Editors

2 authors:

Alexander Krause-Glau
Christian-Albrechts-Universität zu Kiel
18 PUBLICATIONS **246** CITATIONS

Wilhelm Hasselbring
Christian-Albrechts-Universität zu Kiel
402 PUBLICATIONS **5,630** CITATIONS

Collaborative, Code-Proximal Dynamic Software Visualization within Code Editors

Alexander Krause-Glau
Software Engineering Group
Kiel University
Kiel, Germany
akr@informatik.uni-kiel.de

Wilhelm Hasselbring
Software Engineering Group
Kiel University
Kiel, Germany
wha@informatik.uni-kiel.de

Abstract—Software visualizations are usually realized as standalone and isolated tools that use embedded code viewers within the visualization. In the context of program comprehension, only few approaches integrate visualizations into code editors, such as integrated development environments. This is surprising since professional developers consider reading source code as one of the most important ways to understand software, therefore spend a lot of time with code editors.

In this paper, we introduce the design and proof-of-concept implementation for a software visualization approach that can be embedded into code editors. Our contribution differs from related work in that we use dynamic analysis of a software system's runtime behavior. Additionally, we incorporate distributed tracing. This enables developers to understand how, for example, the currently handled source code behaves as a fully deployed, distributed software system. Our visualization approach enhances common remote pair programming tools and is collaboratively usable by employing shared code cities. As a result, user interactions are synchronized between code editor and visualization, as well as broadcasted to collaborators.

To the best of our knowledge, this is the first approach that combines code editors with collaboratively usable code cities. Therefore, we conducted a user study to collect first-time feedback regarding the perceived usefulness and perceived usability of our approach. We additionally collected logging information to provide more data regarding time spent in code cities that are embedded in code editors. Seven teams with two students each participated in that study. The results show that the majority of participants find our approach useful and would employ it for their own use. We provide each participant's video recording, raw results, and all steps to reproduce our experiment as supplementary package. Furthermore, a live demo of our tool is available online.[1] We invite other researchers to extend our open-source software.[2] Video URL: https://youtu.be/3qZVSehnEug

Index Terms—software visualization, dynamic analysis, program comprehension, pair programming, integrated development environment

I. INTRODUCTION

Source code comprehension is still the primary method to come to an understanding of a software system's behavior [1]. This is not unexpected, because developers are trained to recognize recurring patterns and resulting behavior in source code. They even might spend most of their development time in integrated development environments (IDE) [2]. However, navigation in IDEs leads to a redundant but unavoidable overhead [3] and in terms of software visualization (SV) developers are concerned about the context switch caused by standalone SV tools [4]. As a result, code proximity is a necessary property for SV [5], [6] to succeed in its intended area, i.e., professional software development. Code proximity means the ability of the visualization tool to provide easy and fast access to the original, underlying source code [7]. In this context, research approaches have been shown in the past that embed SV into code editors and IDEs (both from now on referred to as *code editor*) to link source code with its visualization [8]–[11].

In this paper, we introduce our collaboratively usable SV approach that can be embedded in code editors. In comparison to related approaches, we use dynamic analysis as source for rendering three-dimensional code cities [12], [13]. The SV is linked directly to the source code that is under development within the code editor and vice versa. Therefore, we directly connect runtime behavior with the related program elements, for example, Java methods. User interactions are synchronized between code editor and visualization, as well as broadcasted to collaborators. As proof of concept, we implemented a Visual Studio Code[3] (VS Code) extension that realizes our design. We conducted a first-time user study to collect feedback regarding the perceived usefulness and perceived usability of our approach. Furthermore, we collected logging information to provide more data regarding usage statistics of SV that are embedded into code editors. In this study, seven teams with two students each collaboratively used our approach in an onboarding-related scenario. Overall, the results show a highly rated usefulness.

The remainder of this paper is structured as follows. Section II presents the architectural overview and proof of concept implementation for our approach. We proceed by

---

[1] https://code.explorviz.dev
[2] https://github.com/ExplorViz
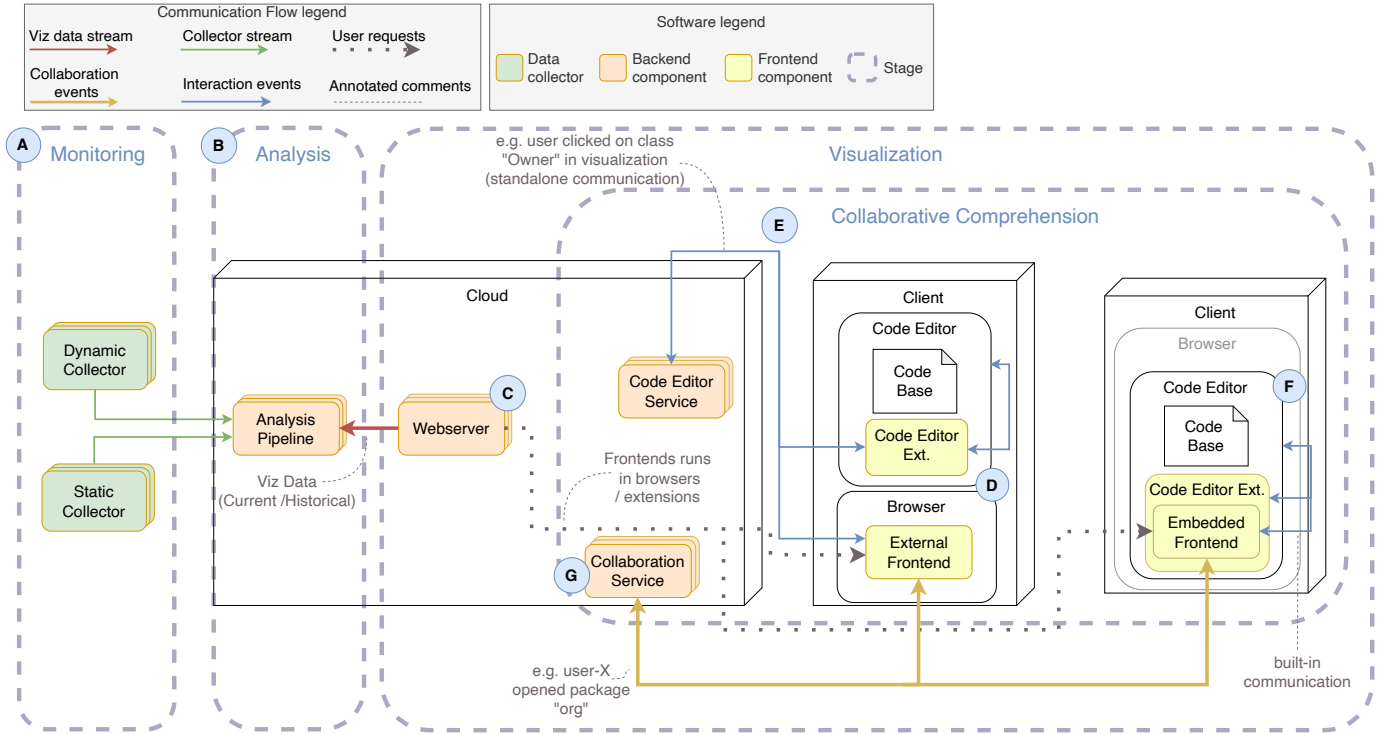
[3] https://code.visualstudio.com

1

Fig. 1: (Simplified) Architectural design of our approach.

introducing the envisioned usage scenarios for our approach in Section III. Afterwards, Section IV explains our experimental setup. Section V presents and discusses the results of our study. Then, Section VI introduces related work. Finally, we conclude this paper and present future work in Section VII.

II. APPROACH

In this section, we present the architectural design and proof of concept implementation for this research work. For that, we build upon our previously published approach named *Software Visualization as a Service* (SVaaS), i.e., providing an online-accessible and on-demand service for collaborative program comprehension using SV. Due to space constraints, we refer readers to [14] for a description of the basic concepts of our approach.

A. Architectural Design

Figure 1 shows (a simplified overview of) our approach's architectural design. It is technology independent with the exception of a browser-based SV component. As shown, it is divided into four stages (blue-striped areas). Figure 1-A and Figure 1-B depict the monitoring and analysis stages, respectively. These are the foundation of our SVaaS concept. The analysis pipeline for example can be horizontally scaled out to handle varying load of concurrent users, therefore positively influence the effectiveness of the overall tool [15]. Although data acquisition, analysis, and cloud technologies are important aspects of our concept, a detailed explanation is beyond the scope of this paper. Therefore, we refer readers to [14] for details and focus on the remaining two stages.

The Webserver (Figure 1-C) serves the static files that comprise the web-based SV, i.e., CSS, JavaScript, and HTML. Furthermore, it acts as reverse proxy for clients to connect to the backend services, e.g., to obtain data to be visualized. Users now have two options to link the SV with their code editor:

- For the first option, they can use the standalone SV that runs inside of their web browser and connects to an extension in their code editor (Figure 1-D). The latter acts as gateway between code editor and SV. This is similar to the 'classic' approach for code viewers embedded into SVs and relates to many other works (see Section VI). Interactions that should be linked between code editor and SV, e.g., 'open a class file in the code editor when the related visualization entity was clicked', are synchronized by the Code Editor Service (Figure 1-E).
- For the second option, users can install an extension in their code editor that already includes the Frontend (Figure 1-F). In this case, we do not need an external service to synchronize interaction events, but use a built-in communication mechanism between code editor and its extension. Therefore, we reduce the context switch overhead that occurs when switching from SV to code editor and vice versa [4]. Another advantage of the second option is that it can also be installed in cloud-based code editors that run in browsers. This can be beneficial in some use cases, e.g., onboarding of new developers, as shown in Section III.

Fig. 2: Proof of concept implementation – The editor of VS Code displays a Java class. The ExplorViz extension visualizes the associated runtime behavior and adds visual functions to the editor to directly link source code and visualization.

Regardless of the selected option, users can collaboratively use the SV. To achieve this, the Collaboration Service (Figure 1-G) broadcasts events, e.g., 'user X opened package Y', to all clients of the same session except the one that triggered an event [16]. The clients then apply the received events to their SV, therefore synchronize their states.

B. Proof of Concept Implementation

We have prototyped our approach within our SV tool ExplorViz.[4] Our tool's development commenced in 2012 [17] and focused on several aspects throughout time, such as development concerns [18], [19] and extended reality [16], [20]. More recently, we use ExplorViz to research collaborative software visualization in the context of program comprehension [16], [21]. ExplorViz currently uses dynamic analysis as source for the visualization. Our depicted SV is configured to visualize the aggregated runtime behavior of ten seconds [14].

Figure 2 shows a screenshot of our prototype implementation. We developed a VS Code extension that realizes the previously mentioned design. It can be used as gateway to link the external SV to the code editor or provide the embedded SV instead. Due to space constraints, we focus on the latter and refer readers to our supplementary video. The extension uses an HTML iFrame to embed the web-based Frontend, therefore SV, in VS Code (see Figure 1-F on the previous page). The embedded SV can be switched on or off via the ExplorViz logo button (Figure 2-A). It is automatically placed in a new editor group next to the source code (Figure 2-B). Users can select one of their (currently or previously) analyzed software systems (as shown in the supplementary video) and open the related SV. The latter is provided as three-dimensional code cities using Three.js[5] for rendering (Figure 2-C). The embedded Frontend uses cross-origin communication based on the JavaScript Window object to interact with VS Code. Therefore, we do not need an external service that synchronizes the interaction events as it is the case when using the external Frontend or as shown in related works

[4]https://explorviz.dev
[5]https://threejs.org

(see Section VI). Every tenth second the Frontend triggers a SV update. For that, it obtains the latest runtime data for the selected software system from the analysis pipeline and updates the visualization if required. Furthermore, the Frontend sends new data to VS Code, which then highlights Java classes and methods that have been used in the aggregated runtime behavior. This is shown by the gutter icons and code lenses in Figure 2-D. Users can click on a code lens to focus the related entity in the SV, e.g., a high-rise building visualizing a Java class. Vice versa, pressing for example on a communication line will cause the file to open and focus on the related method in VS Code. In terms of working together, users can join or host a collaborative session from within the embedded Frontend and use the collaborative features of the SV, e.g., pinging or shared popups (Figure 2-E), to interact with each other (please see [16] for more details). Furthermore, a collaborative session also enables remote pair programming. For VS Code in general, developers can for example use Microsoft's LiveShare extension for VS Code. LiveShare has great features and usability, but uses Microsoft servers that might be not available in the future or cannot be used due to compliance concerns. For the sake of our evaluation's reproducibility, we therefore decided against using an available product such as Microsoft's LiveShare, but developed our own solution (for the user study). This can be seen in Figure 2-F where the live text selection of another user is depicted (as yellow background of `OwnerRepository`). These text selection events are synchronized by an implementation of the external Code Editor Service (Figure 1-E) using WebSockets for almost real-time communication.

III. ENVISIONED USAGE SCENARIOS

Besides using advanced (web) technologies, our approach can be differentiated from related work by the use of dynamic analysis and collaborative SV features. Therefore, we now introduce envisioned usage scenarios that may follow from our approach and related future works.

Scenario 1 (SC1): Facilitate the Onboarding Process

In professional software development, companies utilize different techniques for the onboarding process of new developers. Peer support, product overview, and simple tasks are perceived as useful in that context [22], while finding documentation and technical issues, e.g., setting up a development environment, impede the onboarding process, especially for remote work [23]. We envision a scenario where cloud-based code editors with embedded SVs are prepared to guide new developers step-by-step through a software system's behavior. Users click on a use case of the analyzed (distributed) target system and understand its unfolding via SV. Furthermore, increasingly large portions of the source code (e.g., depending on experience) are directly linked to SV entities. This allows developers to understand which portion of the source code acts in which use cases. The approach can then be used for task-oriented onboarding, where developers also face small tasks to comprehend the software [22], [24]. At any time, users can invite other developers for collaborative comprehension or their mentor and ask for help. Next to voice communication, participants use collaborative features such as synchronized text selection and shared information popups to interact and exchange [16].

Scenario 2 (SC2): Highlight changes during code reviews

Feature requests and resulting change-based code reviews are commonly used in professional software development [25]. However, reviewers tend to give vacuous feedback and generally report on review tools' limitations when used in complex scenarios [26]. In this context, we see another potential usage scenario for our approach that we outline in the following. A team member is supposed to review source code changes of a colleague. To do this, he or she can click on a link inside of the pull request that opens a prepared, cloud-based code editor with an embedded SV of the new program behavior (due to the source code change). Source code changes are color-coded in the IDE. For understanding the program behavior, it is possible to switch between old and new program behavior in the SV by pressing a button. The colleague who issued the pull request can be invited to the session such that the changes can also be discussed together.

Scenario 3 (SC3): Integrate Runtime Information into Development Activities

Staging environments are used to test software systems in a production-like environment. We envision code editors informing selected developers about performance problems of a software system installed (e.g., in the staging area). A developer can click on this notification to open the embedded SV. The visualization depicts the runtime behavior which includes the performance problem. It also highlights the entity that introduces the problem, e.g., a method call that took too long to finish. Based on this, developers get runtime information displayed in their code editor and can analyze affected code lines.

IV. Experiment Design and Demographics

Effectiveness is one of the most common properties used to evaluate SV approaches. In that context, Merino et al. [27] present a systematic literature review of SV evaluation. Their work analyzes the literature body of full papers that were published in the SOFTVIS/VISSOFT conferences, resulting in the examination of 181 papers. The authors focus on evaluations that validate the effectiveness of their presented approach. It is mentioned that multiple evaluations omit other variables that can contribute to or generally influence the effectiveness [28], such as recollection and emotions. We share this opinion and argue that we must first evaluate properties such as perceived usefulness, perceived usability, or feature requests to potentially refine a new, exploratory approach. Only afterwards, we should evaluate effectiveness and efficiency with a sufficiently large number of participants in controlled experiments [29]. As a result, we decided to conduct an exploratory user-study first. We designed an experiment in which participants use and evaluate our approach in a task-oriented onboarding process, i.e., in a scenario similar to SC1 (see Section III). In the future, we will also evaluate our approach in other scenarios by using a similar experiment. In this paper however, we developed the experiment with a focus on SC1 due to the approach's prototype implementation, the exploratory nature of the study, and the duration of a single experiment run. As a result, our research questions (RQ) are not concerned about effectiveness or efficiency. Instead, we focus on several aspects to gather qualitative feedback and quantitative results, such as time spent in the embedded SV, to gain first insights into the use of our approach:

- **RQ1**: How do subjects use the embedded SV and code editor during task solving?
- **RQ2**: Is the code editor perceived as more useful than the embedded SV?
- **RQ3**: Do subjects recognize the usefulness of collaborative SV features for specific tasks?
- **RQ4**: What is the general perception of the usefulness and usability of the approach?
- **RQ5**: Is the approach perceived as useful in the envisioned usage scenarios?

We again emphasize that the findings of this contribution should be seen as first insights and indicators for refinements rather than statistically grounded results. However, by answering the research question, we can derive the following main **contributions** of our evaluation:

- Further insights regarding the perceived usefulness of software cities to comprehend runtime behavior.
- First quantitative and qualitative results regarding the perceived usefulness, perceived usability, and usage time for collaborative, code-proximal software cities.
- A supplementary package containing the evaluation's raw results, screen recordings of all participants, and detailed instructions as well as software packages for reproduction [30].
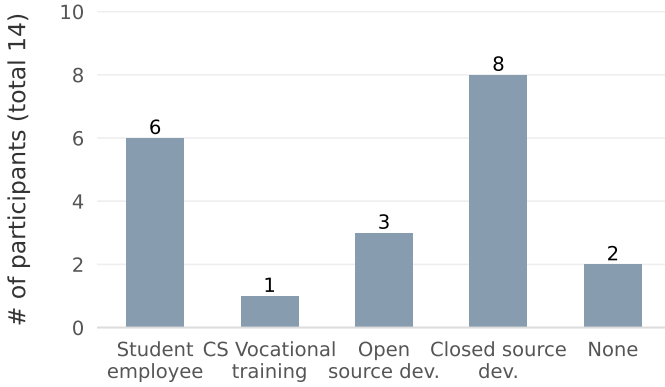
Fig. 3: Participants' reported experiences with software development based on work experiences (multi-choice).

In the following, we now present the participants' demography and our experiment's procedure.

A. Participants

We invited students of Kiel University that attend the Bachelor's or Master's program in computer science to participate in our user study [31]. The participation was voluntary. All participants could sign up for random group assignment or participate with a fellow student. Each group had the chance to win two out of ten 100 € gift cards for an e-commerce shop [32].

Distribution: The conducted user study included seven groups with two students each. The number of participants is therefore slightly larger than the median participant count (thirteen) in the related literature body [27], but too small to

Fig. 4: Participants' reported experiences for different aspects.

be effectively used in a controlled experiment [29], [33]. With the exception of one group, all other participants within their group knew each other. Five students attend the Master's program, the remaining students are undergraduates in computer science. All participants reported that they intend to become professional software developers.

Experiences: Figure 3 shows participants' reported experiences with software development based on work experiences. The two students who indicated they had no experience are in the undergraduate program, and one of them also indicated (as only person) that the decision to become a software engineer is not final. The remaining twelve participants have either gained experiences while working as student employee or in private software development. Three participants are additionally involved in open source development. Figure 4 shows the results of various experiment-related aspects that were asked. All participants stated that they have knowledgeable or even better experiences in VS Code. Three persons rate their web development and software architecture experiences at beginner level. One of the participants with no software engineering work experience reported to have no experience in software architecture. Overall, the distribution of experiences match the courses of study, since SVs are often treated as seminar papers in the master's program, for example. However, we probably also see overestimation such as the persons that stated to be at expert level for VS Code and web development, as well as half of the participants stating to have at least knowledgeable experiences in SV. In this context, half of the participants have used ExplorViz at least once in the past. The participants of three groups each have different experiences with ExplorViz.

B. Target System and Task

ExplorViz' SV visualizes a software system's runtime behavior. However, it is not limited to application tracing of monolithic software systems, but also supports distributed tracing,[6] e.g., network requests between applications that use distributed architectures. Since distributed software systems are pretty common nowadays, we incorporated this fact in our experiment. To achieve that, we used the distributed version of the Spring PetClinic[7] as target system for the experiment. As done in the past [16] we recorded traces during the execution of use cases within the PetClinic. For the experiment, these were then provided as so called snapshots, i.e., aggregated runtime behavior, to the Frontend, resulting in a structural, 'static' SV of dynamic runtime behavior. We decided against using multiple snapshots so as not to overwhelm new users with the amount of features. However, this can be seen in the supplementary video of this work. The participants explored the target system by means of its source code as well as embedded SV and were asked to solve two tasks.

Table I depicts the program comprehension tasks that all participants had to solve during the experiment. We did not use metric analysis tasks such as 'find the class with the
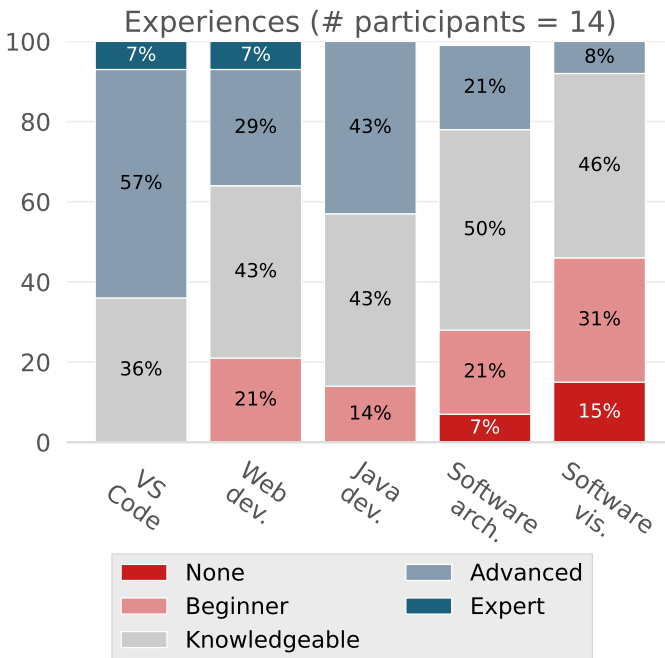
---

[6]https://opentelemetry.io
[7]https://github.com/spring-petclinic/spring-petclinic-microservices

5

TABLE I: Program comprehension tasks that participants had to solve..

ID	Category	Question
T1	Structural Understanding	What do you think is the reason that the 'Owner' class is instantiated multiple times, but the other classes in the relevant program flow are instantiated only once?
T2	Software Insight	Name all Java classes that are involved in a program flow to show the visit screen with the new 'select veterinarian' feature.

highest instance count'. Instead, the chosen tasks instructed the participants to structurally comprehend the software and find analogies based on the depicted runtime behavior. Therefore, the tasks of the experiment refer to a scenario as presented in SC1, i.e., a guided, task-oriented introduction for onboarding. With the focus on SC1, we intend to investigate both the non-collaborative and collaborative onboarding process. Therefore, T1 had to be solved alone and served as an introduction to both the target system and the use of our approach. T2 introduced the collaborative features, e.g., shared SV and synchronized text selection events, and asked the participants to work together.

C. Procedure

In the following, we present the experiment's procedure. For additional information, we refer readers to the second prepared video[8] that demonstrates an exemplary experiment run. Overall, the experiment is divided into pre-questionnaire, mid-questionnaires, i.e., questions that had to be solved after each task completion, and post-questionnaire.

The user study took place at Kiel University and included one instructor who also co-authored this paper. The instructor designed and implemented the approach as well as conducted the user study. Although our approach can be used remotely, we decided to have the study take place in one locality, so that the instructor could intervene if necessary. In each experimental run, the participants were first informed about the data that would be recorded and used for publication. After signing a consent form, the instructor gave a brief introduction to VS Code and the embedded SV. It was mentioned that all introduced features were additionally described on a cheat sheet, which was placed on the table in front of the subjects. Afterwards, the participants were told to openly ask questions if they had a problem. Furthermore, they were told that they could pause or abort the experiment at any time. They draw their login token for the survey tool LimeSurvey[9] and started with the pre-questionnaire. Then T1 was introduced and all participants were redirected to browser-based VS Code instances by clicking a button inside of the LimeSurvey form. Each VS Code instance was specifically prepared for a given

task and ready to use. It did not require any setup, so that the participants could completely focus on the task itself. They began by reading a markdown file that introduced the target system, controls, and the task itself. After answering T1 in LimeSurvey, all participants gave their feedback to the just used approach. T2 was introduced in the same way as T1. However, here participants were instructed to test the collaborative features first and then work together on solving T2. Again, the subjects gave their feedback and concluded with the post-questionnaire. During each experiment run, the instructor made a note of noticeable mentions stated by the participants.

V. RESULTS & DISCUSSION

Our mid-questionnaires and post-questionnaire contained statements for which participants had to indicate their level of (dis)agreement on a 5-point Likert scale. The questionnaires also included free reply fields to leave a comment on any experiment-related matter. Additionally, the instructor made a note of observations such as rational usages of specific features as well as noticeable emotions [27] and mentions of the participants. In the following, we present and use the results of our conducted user study to revisit our posed research questions. Furthermore, we discuss the threats to validity of our evaluation. Although we use the term SV in this paper, we do not want it to be understood as a generalization of our results. We again emphasize that the results and their interpretation are restricted to our particular prototype using collaborative code cities and our experiment. Therefore, the findings should be seen as first insights and indicators for refinements rather than statistically grounded results.
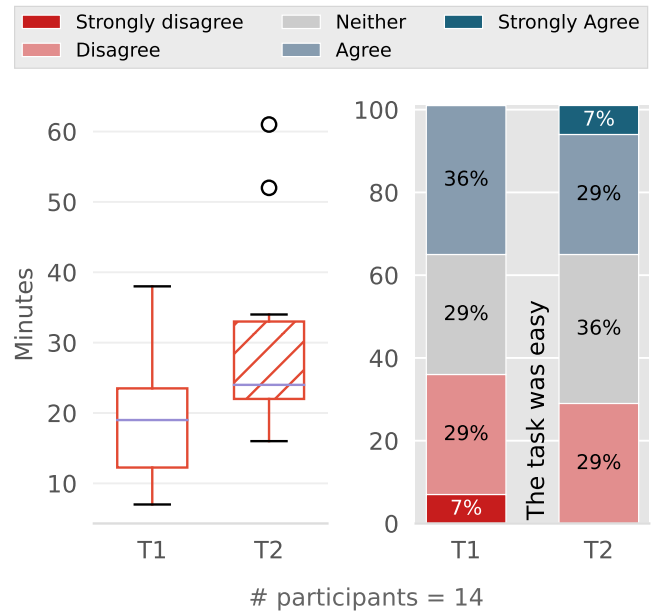

Fig. 5: Total time spent & perceived difficulty per task.

6

Task evaluation

We measured an overall task correctness of 90 %. The related time spent solving the tasks is depicted in Figure 5. The average time spent on T1 is for both the mean and median 19 minutes. The fasted participant correctly solved T1 in seven minutes. This person was already familiar with ExplorViz. For T2, we see 29 minutes for the mean and 24 minutes for the median. Both tasks were without time limit, hence the outlier group for T2. Figure 5 also depicts the participants' perceived task difficulty. T1 and T2 were found to be difficult by four participants, with T1 also found to be very difficult by one person. Due to the overall distribution, we conclude that the tasks were neither too easy nor too difficult.

RQ1: How do subjects use the embedded SV and code editor during task solving?

To the best of our knowledge, this work presents a novel approach that combines code editors with remote pair programming techniques and embedded, collaborative code cities. Therefore, we first intend to understand how the participants in our study use the approach with free choice of the tool, i.e., embedded SV and code editor, as well as with tasks referenced to SC1. In that context, Figure 6 depicts the time spent using each tool per task. For measurement, a VS code event was used to capture the time at which participants clicked on the code editor or the ExplorViz extension, therefore switched their focused context. We would like to mention that it was technically (due to VS Code's limitations for extensions) only possible to measure the time spent between context switches. Thus, if a participant did not change the context but, for example, only used the SV, then our measurements indicate a time spent of one minute for the SV. This is the case for the fastest participant for T1 mentioned above, who actively interacted only with the SV during this task (as confirmed by the video recording). The average time spent using the SV for T1 is seven minutes and nine minutes for VS Code (both mean and median). During this task, participants comprehended the source code for the first time and probably spent more time reading it. It is therefore surprising that the time difference for the first task is already quite small. The reason for this is that code cities can facilitate the understanding of structures and are therefore suitable for the task of obtaining an overview [34], [35]. This was also explicitly mentioned by three participants in the free text fields. For T2, the average time spent using the SV is fifteen minutes and eight minutes for VS Code. The (almost) double amount of time spent using the SV results from the two outliers. For this task, however, the median for time spent using the SV is thirteen minutes and eight minutes for VS code. We suppose that this comes from the shared software cities and the ability to highlight objects in question. The instructor's notes mention the frequent use of shared popups within two groups. The video recordings confirm that these groups often use the popups as a basis for discussion. Also, participants often use the ping feature of our tool to highlight certain details for their collaborator. Therefore, they spent more time
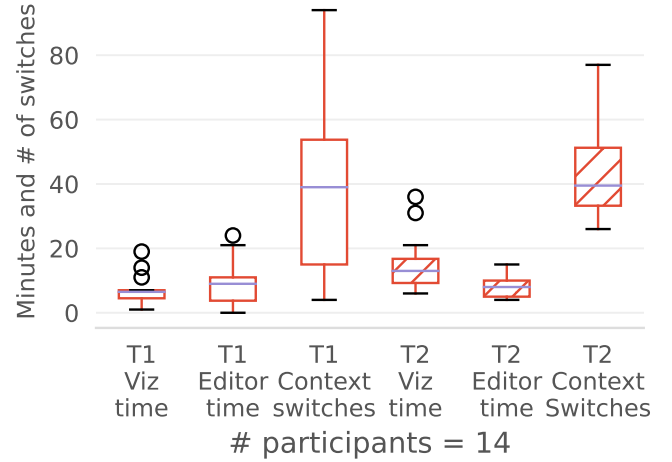

Fig. 6: Time spent per tool & number of context switches performed per task.

using the SV. However, collaboration is not the only reason for that. T2 explicitly requires to understand and extend a program flow. The SV provides a visual overview of the software system's structure and in our case also of a runtime behavior snapshot (see Section IV-B). As a result, it is far easier and obvious to use this available visualization and for example trace imaginary method calls with the mouse cursor (especially, when combined with collaborative features).

Figure 6 also presents the number of context switches for each task. We observe that for T1 the number of switches between SV and code editor is much more distributed among the participants than for T2. Again, the reason for that is presumably the collaboration in T2. Most of the time, the participants work together and therefore change their tool when initiated by the other collaborator. For both T1 and T2, the median of context switches is around forty, indicating that the amount of context switches is independent on our tasks and collaboration.

Since our approach incorporates the runtime behavior of the target system, we also intended to know how participants perceived the usefulness of the two tools to comprehend the posed program flow of T1. In this context, Figure 7 shows that the SV was perceived as more useful than the code editor. One participant mentioned that the communication lines are one of the most beneficial properties of the SV. In ExplorViz, the communication lines incorporate runtime information such as the method call's frequency in the visualized snapshot. These information are important to comprehend runtime behavior. Additionally, the SV already maps the runtime information that the users would otherwise have to find and understand on their own.

> We conclude that the participants used the SV as supplement to the code editor for specific comprehension tasks.
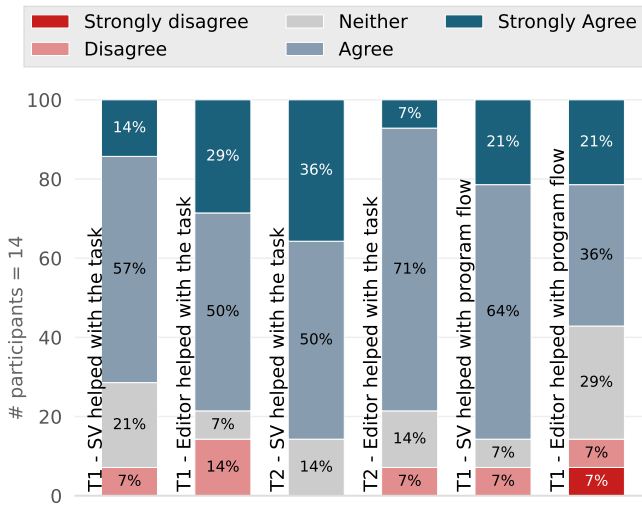
Fig. 7: Mid-questionnaires - Perceived usefulness for tasks

Fig. 8: Mid-questionnaire - T2 - Collaboration

RQ2: Is the code editor perceived as more useful than the embedded SV?

Traditionally, understanding a software system's behavior is primarily achieved by comprehending the source code [1]. For this experiment, the results related to RQ1 show that our approach was, for example, used by the participants to gain an overview of the target system. This is a common and suitable use case for SV, as shown in the past [34]. However, professional developers question the need for SV [15], [36]. In our opinion, one of the reasons for that is the lack of properties such as code proximity [5], [6] and the SV tool's setup [4]. In that context, we now examine how participants rate the usefulness of our approach.

Figure 7 depicts the results of the mid-questionnaires regarding the perceived usefulness of the tools for a task. For T1, overall 71 % agree with the posed statement 'SV helped with the task'. The usefulness of the code editor was slightly (one person difference) more agreed to. However, for the SV the number of participants who neither agree nor disagree is higher and those who disagree is lower. Regarding T2, we see that overall 86 % agree with the posed statement 'SV helped with the task'. In comparison, the code editor's usefulness was slightly (one person difference) less agreed to.

> We conclude that the participants perceive code editor and SV as approximately equally useful (in the context of the task solving).

RQ3: Do subjects recognize the usefulness of collaborative SV features for specific tasks?

With RQ3, we expand the results of our previous work [16] regarding the perceived usefulness of collaborative code cities. In this context, we asked all participants to state their level of agreement with two statements posed.
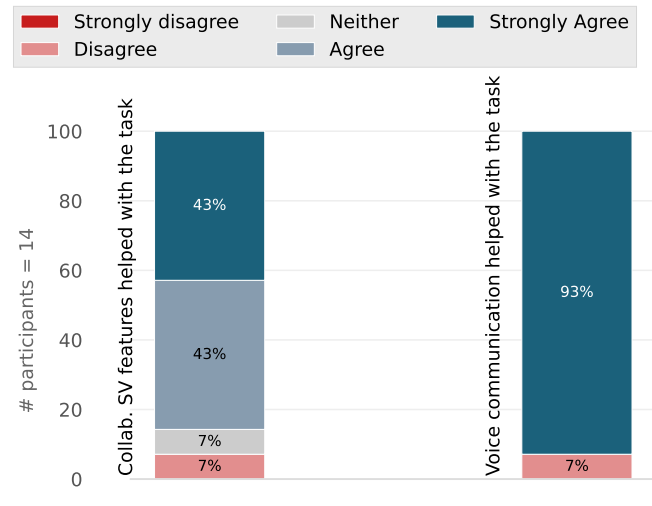
Figure 8 presents the related results. We see that 43 % of the participants agree or strongly agree with the statement 'Collaborative SV features helped with the task', respectively. The one person that disagrees with the statement mentioned that the collaborative SV features did not help in his case, since there was barely any input from the other participant. However, he agrees that the communication would be a big help in pair programming supported development in the real world. Presumably due to the low contribution of his collaborator, the same person also disagrees with the second statement that is concerned about voice communication. Due to low input from his collaborator, the same person also disagrees with the second statement, which refers to the perceived usefulness of voice communication. Nevertheless, all of the remaining thirteen participants strongly agree that voice communication was helpful in the task. This is consistent with our previous findings indicating that voice communication is one of the most useful collaborative tools in SV [16] .

> We conclude that the majority of participants perceive the collaborative SV features as useful in the given task.

RQ4: What is the general perception of the usefulness and usability of the approach?

The post-questionnaire was designed to capture participants' overall perceptions of the approach's usefulness and usability. By answering RQ1, we have seen that the participants indeed use the SV as supplement during the comprehension task. For RQ2, we concluded that participants perceived code editor and SV to be about equally useful in the context of a real-world task. Finally, Figure 9 shows:

> All participants agree or strongly agree that the SV's code proximity is generally useful.
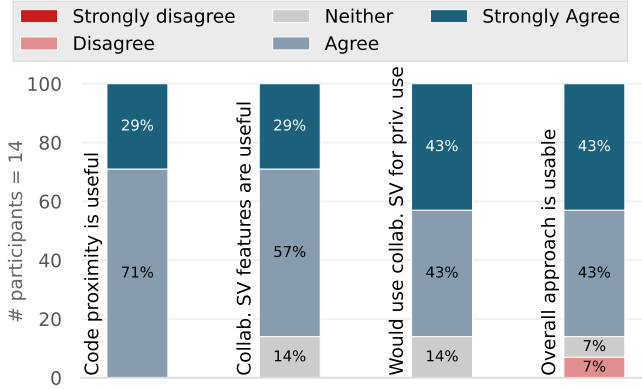
Fig. 9: Post-questionnaire - Perceived usefulness and usability

Fig. 10: Post-questionnaire – Perceived usefulness of the approach when applied in a described scenario. (see Section III)

Collaboration is obviously dependent on many factors, e.g., mutual perception of collaborators or motivation. In our context, we have seen this for RQ3 or in previously published results [16]. The participants rate the collaborative SV features slightly different when to be evaluated independently of a task. Figure 9 shows a shift in the distribution of approval ratings. The one person who previously disagreed with the usefulness of the collaborative features now neither agrees nor disagrees. That fits his previous mentions. Compared to perceived usefulness for T2, overall perceived usefulness of collaborative SV features shows less strong agreement. As a matter of fact, we could not find a reason why two participants downgraded their level of agreement to 'agree'. However, the overall approval rate remains the same.

> We conclude that the majority of subjects perceive the collaborative SV features as useful.

Although this evaluation is overall more concerned about the perceived usefulness of embedded SV, identified usability problems can help to identify desirable refinements. In this context, Figure 9 also presents the participant's perceived usability of our approach. The results show that 86 % of the participants find the used combination of embedded SV and code editor usable. There are some desirable improvements that are mentioned via text response, e.g., better performance. However, the biggest usability problem was the unintended minimization of the embedded SV. The reason for that is that VS code opens files that have been clicked in the package explorer in the currently focused editor group. This behavior can be disabled by locking an editor group. However, at the current time of writing, the lock mechanism cannot be triggered from within a VS Code extension. Figure 9 also shows that another 86 % would use this approach for private purposes such as collaborative program comprehension with fellow students.

> We conclude that the majority of participants find our approach usable.
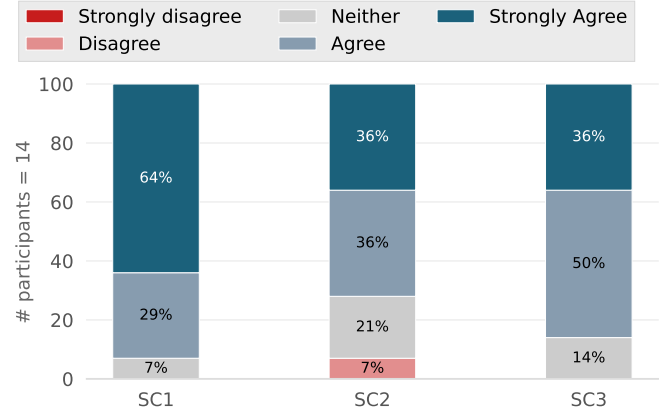
RQ5: Is the approach perceived as useful in the envisioned usage scenarios?

Our pilot study found that a single experiment run would take about an hour to complete. In order not to discourage potential participants due to the time to be spent, we decided to ignore the other usage scenarios and only use tasks in the experiment based on SC1. Nevertheless, the post-questionnaire was also used to capture participants' perceived usefulness in applying the approach in the remaining, envisioned scenarios. In this case, they were described in text and subjects were asked to state their agreement on a 5-point Likert scale. Figure 10 depicts the related results. The complete scenario descriptions are available in the supplementary package of this paper [30], but essentially summarize the envisioned usage scenarios in Section III. The participants rated SC1 with the highest overall agreement and strong agreement, respectively. The experiment's tasks and their introduction originate from SC1. SC2 has the highest amount of neutrality and disagreement. One person that answered with neither agreement nor disagreement mentioned that code changes are usually reviewed before deploying them. Since our approach only shows runtime behavior, he is not sure how changes will be visualized for the code review. This detail was in fact omitted in the textual description of SC2. We believe that this uncertainty is the reason for the highest amount of neutrality and disagreement for SC2. However, the majority consensus was positive for all scenarios.

> We conclude that the majority of subjects find the application of our approach useful in the posed scenarios.

A. Threats to Validity

Remote pair programming solution: As mentioned in Section II-B, we decided to implement our own remote pair programming approach, so that the reproducibility of our evaluation is not dependent on the availability of external services. However, this custom implementation lacks useful

features compared to full-fledged solutions for remote pair programming. For example, one participant mentioned that he was unable to draw the attention of the collaborator to a specific code part. Although our study did not aim to evaluate whether one tool is better than the other, this custom implementation may have influenced the perceived usefulness or usability of the SV or code editor. In contrast, Figure 7 shows that the participants find the SV to be more suitable to understand dynamic program flows. With that being said, we conclude that more empirical research is required in this context.

Experiment duration: The average time spent on the user study was about one hour, both median and mean. It follows that the attention span of the participants and thus the results might have been influenced. To mitigate this, we told participants during the introduction that breaks could be taken at any time and participation could be aborted. Moreover, T2 was solved collaboratively and therefore presumably relieved the experimental situation.

Target system: The prepared target system contains 26 application logic-related Java files that are distributed among four Maven subprojects. As a result, the small project size may have influenced the perceived usability of the SV, as also mentioned by one participant. We agree, but also emphasize that we did not intend to evaluate usability based on the scalability of the visualization, but on the overall concept. Overall, this evaluation is more concerned about the perceived usefulness of SV incorporating distributed tracing for the onboarding process. In addition, we argue that a real-world application of the onboarding scenario with SV should guide new developers through the software system's behavior with increasingly large portions of the code base.

Participants: The use of students in experiments is a valid simplification that is often said to possibly compromise external validity [31]. In our case, the participants' experiences might have influenced their perception regarding the usefulness of the SV as well as their time spent using the SV. In this context, professional developers can benefit from their experience, e.g. with the Spring framework, and can understand the source code faster. As a result, we will repeat the experiment with professional developers.

VI. Related Work

Code proximity is often not clearly emphasized in SV publications, but follows from the mentions of code viewers in the text itself. Therefore, there are numerous research approaches that use embedded code viewers within SV such as code cities [34]. This also applies to more recent and often collaboratively usable virtual reality approaches [37]–[41]. Other publications present different use cases for embedded SV in code editors, such as dependency browsing [42] or sketching [43], [44]. Few approaches enable developers to modify source code via embedded code editors [45]. Due to space limitations, we cannot address and discuss all related approaches [8], [10], [11], but focus below on what we consider to be the most comparable work.

In 2015, Balogh et. al presented a refined version of their tool CodeMetropolis [9]. Their approach uses static analysis of a software systems' source code and visualizes the result as 3D code city. The related rendering is achieved using a modded version of the video game Minecraft. Thanks to the multiplayer mode of Minecraft, the code city can also be explored collaboratively. Overall, CodeMetropolis and ExplorViz share the aspects of collaboration and code editor integration. However, these features are implemented differently in each case. For example, in CodeMetropolis users navigate through the same instance of a given code city using the first-person perspective. They can see the avatars of collaborators and interact based on Minecraft's limitations. In ExplorViz, the collaboration is achieved using collaborative SV features, e.g., shared popups. Regarding the code editor integration, both CodeMetropolis and ExplorViz provide an extension than can be installed in Eclipse and VS Code, respectively. In this context, both extensions provide a comparable set of features, e.g., open Java class in the SV. However, our extension is also able to embed the SV in the actual code editor, whereas the Metropolis approach can only be used as external SV that links to the code editor (see Section II-A).

VII. Conclusions & Future Work

In this paper, we presented the architectural design of our approach for collaborative, code-proximal dynamic software cities within code editors. The main idea is to link collaborative SVs directly to the source code that is under development within a code editor and vice versa. We have prototyped this approach within our SV tool ExplorViz. The result is a VS Code extension that either embeds three-dimensional software cities in the code editor or acts as gateway between code editor and external SV. Therefore, we directly link runtime behavior with the related program elements, for example, Java methods. Users can collaboratively explore the SV from within their code editor using synchronized software cities and collaborative SV features, e.g., shared popups. In addition to the implementation, we sketched three envisioned usage scenarios.

We conducted an initial user study to collect first-time feedback regarding the perceived usefulness and perceived usability of our approach. The results show that the majority of participants generally perceive the approach as useful and usable. In this context, participants rated code editor and SV as equally useful in solving the given program comprehension tasks. The measured time spent in each tool, i.e., SV and code editor, indicates that the participants indeed use the SV as supplementary tool.

In the future, we will implement useful features and refinements. Additionally, we plan to repeat the experiment with professional developers.

REFERENCES

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018. DOI: 10.1109/TSE.2017.2734091

[2] R. Tiarks, "What maintenance programmers really do: An observational study," *Softwaretechnik-Trends*, vol. 31, no. 2, 2011. [Online]. Available: https://dl.gi.de/handle/20.500.12116/40982

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006. DOI: 10.1109/TSE.2006.116

[4] M. Sensalire and P. Ogao, "Visualizing object oriented software: Towards a point of reference for developing tools for industry," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007. DOI: 10.1109/VIS-SOF.2007.4290696 pp. 26–29.

[5] S. Bassil and R. Keller, "Software visualization tools: survey and analysis," in *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, 2001. DOI: 10.1109/WPC.2001.921708 pp. 7–17.

[6] H. M. Kienle and H. A. Muller, "Requirements of software visualization tools: A literature survey," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007. DOI: 10.1109/VISSOF.2007.4290693 pp. 2–9.

[7] M. Lanza and S. Ducasse, "Polymetric views – A lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, Sep. 2003. DOI: 10.1109/tse.2003.1232284

[8] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu, "Plugging-in visualization: Experiences integrating a visualization tool with eclipse," in *Proceedings of the 1st ACM Symposium on Software Visualization (SoftVis 2003)*, ser. SoftVis '03. Association for Computing Machinery, 2003. DOI: 10.1145/774833.774840 p. 47–ff.

[9] G. Balogh, A. Szabolics, and A. Beszedes, "CodeMetropolis: Eclipse over the city of source code," in *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*, 2015, pp. 271–276.

[10] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: A systematic mapping study," *Journal of Visual Languages & Computing*, vol. 49, pp. 46–59, 2018. DOI: 10.1016/j.jvlc.2018.10.001

[11] Z. Kurbatova, Y. Golubev, V. Kovalenko, and T. Bryksin, "The intellij platform: A framework for building plugins and mining software data," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW 2021)*, 2021. DOI: 10.1109/ASEW52652.2021.00016 pp. 14–17.

[12] C. Knight and M. Munro, "Comprehension with[in] virtual environment visualisations," in *Proceedings of the 7th International Workshop on Program Comprehension*, 1999. DOI: 10.1109/WPC.1999.777733 pp. 4–11.

[13] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2007. DOI: 10.1109/VIS-SOF.2007.4290706 pp. 92–99.

[14] A. Krause-Glau and W. Hasselbring, "Scalable collaborative software visualization as a service: Short industry and experience paper," in *Proceedings of the 10th IEEE International Conference on Cloud Engineering (IC2E 2022)*, 2022. DOI: 10.1109/IC2E55432.2022.00026 pp. 182–187.

[15] M. Sensalire, P. Ogao, and A. Telea, "Classifying desirable features of software visualization tools for corrective maintenance," in *Proceedings of the 4th ACM Symposium on Software Visualization (SoftVis 2008)*, ser. SoftVis '08. New York, NY, USA: Association for Computing Machinery, 2008. DOI: 10.1145/1409720.1409734 p. 87–90.

[16] A. Krause-Glau, M. Bader, and W. Hasselbring, "Collaborative software visualization for program comprehension," in *Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 2022)*, 2022. DOI: 10.1109/VISSOFT55257.2022.00016 pp. 75–86.

[17] W. Hasselbring, A. Krause, and C. Zirkelbach, "ExplorViz: Research on software visualization, comprehension and collaboration," *Software Impacts*, vol. 6, Nov. 2020. DOI: 10.1016/j.simpa.2020.100034

[18] C. Zirkelbach, A. Krause, and W. Hasselbring, "Modularization of research software for collaborative open source development," in *Proceedings of the 9th International Conference on Advanced Collaborative Networks, Systems and Applications (COLLA 2019)*, Juni 2019, pp. 1–7. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=colla_2019_1_10_50005

[19] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice decomposition via static and dynamic analysis of the monolith," in *Companion Proceedings of the 17th IEEE International Conference on Software Architecture (ICSA-C 2020)*, 2020. DOI: 10.1109/ICSA-C50368.2020.00011 pp. 9–16.

[20] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," in *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT 2015)*, 2015. DOI: 10.1109/VIS-SOFT.2015.7332423 pp. 130–134.

[21] A. Krause-Glau, M. Hansen, and W. Hasselbring, "Collaborative program comprehension via software visualization in extended reality," *Information and Software Technology*, vol. 151, p. 107007, 2022. DOI: 10.1016/j.infsof.2022.107007

[22] J. Buchan, S. G. MacDonell, and J. Yang, "Effective team onboarding in agile software development: techniques and goals," in *Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2019)*, 2019. DOI: 10.1109/ESEM.2019.8870189 pp. 1–11.

[23] P. Rodeghero, T. Zimmermann, B. Houck, and D. Ford, "Please turn your cameras on: Remote onboarding of software developers during a pandemic," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*, 2021. DOI: 10.1109/ICSE-SEIP52600.2021.00013 pp. 41–50.

[24] A. Ju, H. Sajnani, S. Kelly, and K. Herzig, "A case study of onboarding in software teams: Tasks and strategies," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*, 2021. DOI: 10.1109/ICSE43902.2021.00063 pp. 613–623.

[25] T. Baum, H. Leßmann, and K. Schneider, "The Choice of Code Review Process: A Survey on the State of the Practice," in *Product-Focused Software Process Improvement*, vol. 10611. Springer, 2017. DOI: 10.1007/978-3-319-69926-4_9 pp. 111–127.

[26] L. Dong, H. Zhang, L. Yang, Z. Weng, X. Yang, X. Zhou, and Z. Pan, "Survey on pains and best practices of code review," in *Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC 2021)*, 2021. DOI: 10.1109/APSEC53868.2021.00055 pp. 482–491.

[27] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *Journal of Systems and Software*, vol. 144, no. C, p. 165–180, oct 2018. DOI: 10.1016/j.jss.2018.06.027

[28] D. Feitelson, "Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension," *Empirical Software Engineering*, vol. 27, 11 2022. DOI: 10.1007/s10664-022-10160-3

[29] M. D. Gall, W. R. Borg, and J. P. Gall, *Education research, an introduction*. Longman, 1996.

[30] A. Krause-Glau and W. Hasselbring, "Supplementary data for: Collaborative, code-proximal dynamic software visualization within code editors," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8111133

[31] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "Empirical software engineering experts on the use of students and professionals in experiments," *Empirical Softw. Engg.*, vol. 23, no. 1, p. 452–489, feb 2018. DOI: 10.1007/s10664-017-9523-3

[32] C. F. Camerer and R. M. Hogarth, "The effects of financial incentives in experiments: A review and capital-labor-production framework," *Journal of Risk and Uncertainty*, vol. 19, no. 1, pp. 7–42, 1999. DOI: 10.1023/A:1007850605129

[33] L. Cohen, L. Manion, and K. Morrison, *Research Methods in Education*. Routledge, 2017.

[34] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*, 2011. DOI: 10.1145/1985793.1985868 pp. 551–560.

[35] F. Galperin, R. Koschke, and M. Steinbeck, "Visualizing code smells: Tables or code cities? a controlled experiment," in *Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 2022)*, 2022. DOI: 10.1109/VISSOFT55257.2022.00014 pp. 51–62.

[36] J. Siegmund, "Program comprehension: Past, present, and future," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, vol. 5, 2016. DOI: 10.1109/SANER.2016.35 pp. 13–20.

[37] R. Koschke and M. Steinbeck, "See your clones with your teammates," in *Proceedings of the 15th IEEE International Workshop on Software Clones (IWSC 2021)*, 2021. DOI: 10.1109/IWSC53727.2021.00009 pp. 15–21.

[38] R. Oberhauser and C. Lecon, "Gamified virtual reality for program code structure comprehension," *International Journal of Virtual Reality*, vol. 17, no. 2, p. 79–88, Jan. 2017. DOI: 10.20870/IJVR.2017.17.2.2894

[39] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola, "Code Park: A new 3D code visualization tool," in *Proceedings of the 5th IEEE Working Conference on Software Visualization (VISSOFT 2017)*, 2017. DOI: 10.1109/VISSOFT.2017.10 pp. 43–53.

[40] A. Hori, M. Kawakami, and M. Ichii, "CodeHouse: VR code visualization tool," in *Proceedings of the 7th Working Conference on Software Visualization (VISSOFT 2019)*, 2019. DOI: 10.1109/VISSOFT.2019.00018 pp. 83–87.

[41] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "CityVR: Gameful software visualization," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, 2017. DOI: 10.1109/ICSME.2017.70 pp. 633–637.

[42] K. Borowski, B. Balis, and T. Orzechowski, "Graph Buddy — an interactive code dependency browsing and visualization tool," in *Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 2022)*, 2022. DOI: 10.1109/VISSOFT55257.2022.00023 pp. 152–156.

[43] L. Lichtschlag, L. Spychalski, and J. Bochers, "CodeGraffiti: Using hand-drawn sketches connected to code bases in navigation tasks," in *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2014. DOI: 10.1109/VL-HCC.2014.6883024 pp. 65–68.

[44] C. Kurtz, "Code gestalt: A software visualization tool for human beings," in *Extended Abstracts on Human Factors in Computing Systems (CHI 2011)*. New York, NY, USA: Association for Computing Machinery, 2011. DOI: 10.1145/1979742.1979518 p. 929–934.

[45] J. Dominic, B. Tubre, C. Ritter, J. Houser, C. Smith, and P. Rodeghero, "Remote pair programming in virtual reality," in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*, 2020. DOI: 10.1109/ICSME46990.2020.00046 pp. 406–417.