Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality

Adrian Hoff*, Christoph Seidl*, Mircea Lungu*, Michele Lanza†
*IT University of Copenhagen, Denmark — †Software Institute @ USI Università della Svizzera italiana, Switzerland

***Abstract*—Re-architecting a software system requires significant preparation, e.g., to scope and design new modules with their boundaries and constituent classes. When planning an intended future state of a system as a re-engineering goal, engineers often fall recur to mechanisms such as freehand sketching (using a whiteboard). While this ensures flexibility and expressiveness, the sketches remain disconnected from the source code. The alternative, tool-supported diagramming on the other hand considerably restricts flexibility and impedes free-form communication.**

We present a method for preparing the architectural software re-engineering via freehand sketches in virtual reality (VR) that can be seamlessly integrated with the model structure of a software visualization and, thus, also the code of a system, for productive use: Engineers explore a subject system in the immersive visualization, while freehand sketching their insights and plans. Our concept automatically interprets sketched shapes and connects them to the system's source code, and superimposes code-level references into a sketch to support engineers in reflecting on their sketches.

We evaluated our method in an iterative interview-based case study with software developers from four different companies, where they planned a hypothetical re-engineering of an open-source software system.

***Index Terms*—Software Re-Engineering, Software Visualization, Whiteboard Sketching, Reflexion Models, Virtual Reality**

***Video Demonstration*—https://youtu.be/NKC5YpH3n4Y**

I. Introduction

Re-engineering an existing software system is an endeavour that requires significant preparation [1]. This preparation encompasses cycles of (1) reverse engineering (exploring and understanding relevant aspects of the system, such as its architectural structure), (2) identifying re-engineering opportunities (such as unintended dependencies between architectural components), and (3) planning an intended future state as re-engineering goal [2], [3]. Different methods exist that support engineers in preparing for software re-engineering. Time-proven means include software visualization [4], [5] and architecture conformance checking techniques such as reflexion modeling [6]–[9]. These support engineers in establishing a high-level overview of a system that they deepen and refine over time while exploring and planning. In doing so, it is crucial for engineers (and their peers) to persistently externalize their insights and intentions [10], [11]. Software engineers' preferred method for that is freehand sketching, e.g., on a whiteboard or piece of paper [11]–[13]. It allows them to capture complex problems and situations in intentionally incomplete sketches that they refine over time [12], [14], [15].

Existing techniques for software re-engineering preparation (such as software visualization or architecture conformance checking) do not provide sufficient flexibility and expressiveness. Engineers prefer other mediums that provide the necessary flexibility for capturing insights and plans such as, in most cases, physical whiteboards. The result is a mix of separate, disconnected artifacts that need to be maintained in parallel to the system's code itself [10], [16].

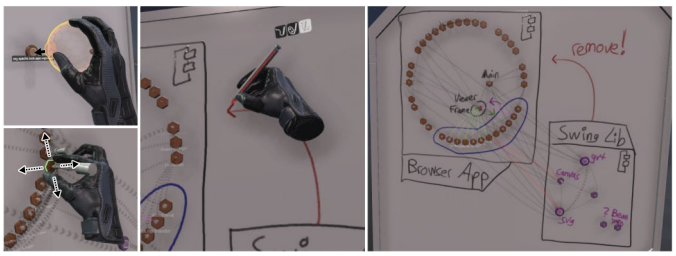

Fig. 1. Freehand Sketching in VR.

We present a method for extending existing VR software visualizations with virtual whiteboards for creating freehand sketches on a system's structure (see Figure 1) while continuously receiving automated conformance checks, similar to those proposed by the reflexion modeling approach. Engineers pin elements from the visualization (representing source code elements such as classes or packages) on a virtual whiteboard and draw freehand sketches on it using a virtual pen, as they would on a physical whiteboard.

Our method enables the re-engineer to sketch outlines around pinned elements and to connect these outlines with arrows. It automatically interprets sketches and maps them on the source code of the subject system such that it can subsequently provide engineers with visual feedback on the conformance of a sketch with the ground-truth structure of the subject system. This helps engineers with reflecting on their sketches ("is this what the system looks like?") and planned re-engineering goals ("should it really look like this?"). Engineers benefit from the overview of a subject system provided by existing software visualizations, while being able to capture insights and plans via flexible freehand sketches with instant conformance checks along the way. To support engineers with implementing their plans, our method mirrors sketches captured on a VR whiteboard to a traditional 2D-screen IDE. This closes the gap between otherwise disconnected artifacts and the source code of a system and, thus, facilitates the preparation of architectural software re-engineering.

II. Related Work: Preparing Re-Engineering

Various techniques exist for preparing software architecture re-engineering, to support engineers in analyzing the architectural structure and behavior of a system, identifying re-engineering opportunities, and planning an intended future state as re-engineering goal. We elaborate on three relevant areas, highlighting a gap in the corpus of existing techniques.

A. Software Visualization

Software visualization techniques represent the intangible structures, interrelations, and interactions of software via visual metaphors in 2D and 3D [17].

Most 2D metaphors are abstract (e.g., graphs and tree maps [2], [18]–[20]), whereas 3D metaphors range from being abstract (e.g., 3D graphs [21]–[23]) to real-world inspired with the information city [24]–[31] as one of the most commonly applied 3D software visualization metaphors. Visualizations in 3D can be distinguished by their medium as being displayed on a 2D standard screen or in immersive virtual reality (VR) or augmented reality (AR) via head-mounted devices.

Software visualization is helpful for gaining an overview of a system during re-engineering [2] [32] [33] and new techniques, such as VR, have the potential to advance the state of user interaction with visualizations for the purpose of documenting and planning a re-engineering process.

A common shortcoming of many software visualization techniques is that they remain disconnected from the source code, and often run as stand-alone tools or web applications, losing the crucial link to the IDE.

B. Reflexion Models

Software architecture compliance checking approaches support engineers with building an understanding of a software system's architecture by providing insights into how it conforms to user-specified views or rules [7]. A prominent instance are reflexion models by Murphy et al. [8], [34] which let engineers specify a high-level view on a system's architecture via a graphical representation (boxes, arrows). Engineers then manually construct a mapping from architectural entities to software elements in a system. An automated analysis provides engineers with feedback on their specified high-level view on the system's architecture. That is, which arrows were placed in the high-level view where there actually are no relations in the system, which arrows are missing in the high-level view, and which arrows do conform with the code-level relations in the system? This supports engineers in reflecting on their high-level view and the structures these describe [6], [35]. They iteratively refine the high-level view until it reaches a satisfactory state. Along the way, this workflow fosters activities such as finding re-engineering opportunities, which in turn makes reflexion models a valuable tool for preparing architectural re-engineering.

Subsequent techniques extended Murhpy's approach with advanced support for hierarchical structures [4], applying it to a behavioral analysis of distributed systems [36], or easing the detection of architectural flaws [37].

The aspect of reflexion modeling that was most picked up by subsequent work is its mapping from architectural entities (boxes) to source code elements. In the original work [8], [34], engineers manually specify this mapping via regular expressions over the system's source code artifacts – which can be tedious and, at times, inaccurate. Subsequent approaches either improve the manual mapping process directly [38] or they replace it with automated techniques [39]–[44].

With regards to note making, reflexion models have the advantage of being able to capture incomplete views on a system's architecture, encompassing only architectural entities relevant for a given context. However, reflexion modeling (including its derivatives and extensions) requires engineers to follow a strict, deliberately limited notation when defining their architectural views. Deviations from that are not possible while additional comments and notes need to be externalized, resulting in different artifacts which need to be maintained separately. Thus, reflexion modeling alone is not suitable for documentation and planning purposes.

C. Freehand Sketching on Whiteboards and Paper

Engineers value flexibility when creating diagrams on their software systems [12], [13], [16], e.g., for planning purposes. More formal visual languages such as UML notations or ER are used less and are often mixed with informal sketches [10], [16], especially in early stages of planning, where engineers deliberately improvise rough sketches to ad-hoc capture thoughts [10], [12], [13], [45]. Generally, sketches are incomplete abstractions of complex situations and structures that incorporate only relevant aspects [46], [47], they serve as cognitive tools that externalize ideas to relieve the mind [14]. The workflow is to sketch situations, discover new relation, refine the sketch, and repeat [48], [49], which requires a high degree of flexibility in the sketching process.

A popular medium are freehand drawings on whiteboards and paper [10], [16], rated as the most effective [11]. Often, complex problems and situations are not clear to engineers who intentionally sketch incomplete diagrams and refine these over time [13], [47], [50]. The relevant feature for supporting engineers in expressing thoughts is being able to mix and improvise notations without restrictions [13]. The problem is how to persist drawings in a virtual format [10]: such diagrams have a transient nature, are disconnected from the code, and thus cannot provide feedback on conformance to reality.

Smart whiteboards let engineers freehand draw while automatically capturing their pen strokes in a digital format [51]–[53] [54]. Tools exist that interpret sketches to detect elements from certain visual notations (UML) [52], [55]–[57]. The general idea is to let engineers draw arbitrary forms and map these to a certain notations, e.g., UML class diagrams. While this is a useful step towards enabling engineers to more conveniently document architectures and plans digitally, a majority rigidly enforce conformance to certain notations (whereas we discussed previously that liberty in notation is important), and none maintain an explicit mapping to represented elements on source code level.

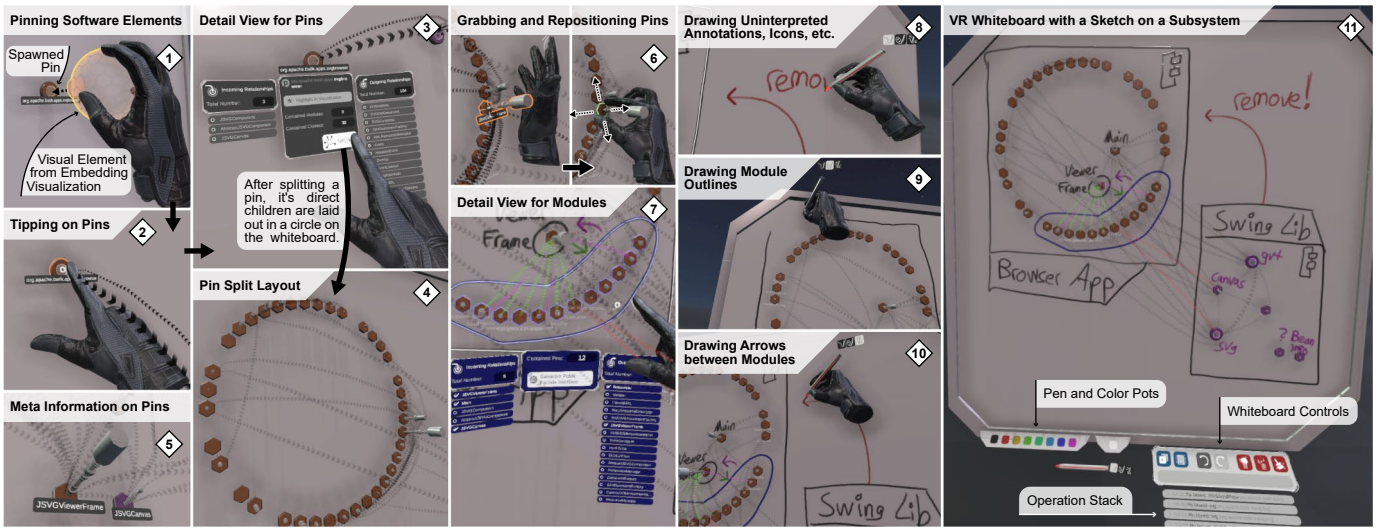

Fig. 2. Screenshots of an implementation of our method in an existing VR software visualization. The embedding visualization is not depicted.

III. Freehand Reflexion Models in VR

Our method extends an existing VR software visualization with a virtual whiteboard for the purpose of externalizing insights and plans on a system's structure via flexible freehand sketches that automatically integrate with the code of a system. We provide a conceptual overview over our method in Figure 3, which we discuss throughout this section.

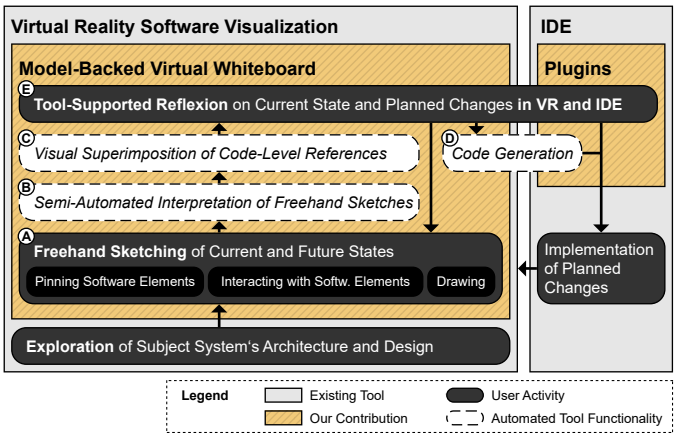

Fig. 3. Overview of our method for preparing software re-engineering via model-backed freehand sketches in a VR visualization.

Figure 2 shows an example implementation of our method in an existing VR software visualization[1]. The presented images do not include the embedding visualization, into which we implemented our whiteboard sketching method. However, the reader should imagine how engineers interact with the embedding visualization and bring source code elements (e.g., classes in a Java system) from the embedding visualization over to the whiteboard to pin them Ⓐ. They then freehand sketch on the whiteboard to visualize annotations, outline pinned elements, or establish relations between outlined elements as arrows between them. Our method automatically interprets these sketches Ⓑ to establish an explicit mapping between sketched forms and the source code they describe. It dynamically superimposes code-level references (such as method calls) between drawn elements on a whiteboard Ⓒ, which supports engineers in reflecting on their sketches as well as on the structures they describe Ⓔ, e.g., the architecture of a (sub-)system.

Engineers might then go back to the whiteboard to refine or re-plan, or they implement their intended changes in code using an IDE. To facilitate the latter, our method (i) mirrors sketches made in VR to an IDE for supporting fine-grained changes and (ii) offers automated code generation from within VR based on sketches Ⓓ. We elaborate on each of these steps.

Ⓐ *Freehand Sketching of Current and Future States*

From a user's perspective in VR, our method consists of a virtual whiteboard (Figure 2) on which engineers pin software elements and draw sketches on, similar to how they would on a physical whiteboard (Ⓐ in Figure 3). Our intention is to provide engineers with the means for flexible and rapid freehand sketching that integrates with the embedding software visualization (i.e., its visual elements, user actions, etc.) and the code of a represented system.

Pinning Software Elements: Engineers attach software elements by grabbing their representations in the VR visualization and pinning them on the whiteboard. Figure 2 ◇1 depicts an example implementation of this mechanism: It leaves behind a *pin* on the whiteboard which represents and explicitly maps to the represented software element. Each pin contains a small *avatar*, a miniaturized version of the pinned visual element it represents ◇5. For one, these avatars facilitate the engineers' mental mapping between pins and visual elements, which helps with distinguishing pins. For another, because the gestalt of visual elements in software visualizations usually encodes relevant metrics, avatars on pins carry this information, too.

[1]https://gitlab.com/immersive-software-archaeology

Including software elements in a sketch by pinning them on a whiteboard is quick and unambiguous and offers potential for cross-fertilizing effects with existing mechanisms in the embedding visualization (such as efficient means for navigating the subject system). It is up to the developers of a VR visualization that integrate our method to decide which software elements can be attached to a virtual whiteboard. We illustrate our method with high-level programming language constructs such as classes, interfaces, structs, etc. as well as architectural units that organize these, e.g., packages, namespaces, folders.

Interacting with Attached Software Elements: Engineers can grab pins on a whiteboard after they were placed and freely reposition or entirely remove them ⟨6⟩. This creates a rapid editing process with low costs for subsequent changes, especially when compared to sketching on a physical whiteboard. Engineers can also open a detail view for a pin (⟨2⟩ and ⟨3⟩) that displays information on the mapped software element, lists related elements, and offers element-specific operations.

Navigation along Containment: When planning changes to a system's architecture, these will concern its organization in architectural components (folders, packages, namespaces). To support engineers in working with such components, and given that many of these elements have nested elements, our method provides an automated split operation (⟨3⟩ and ⟨4⟩) that replaces one pin for an architectural component with pins for all of their constituent elements, e.g., a pin for a Java package can be replaced with pins for all of its direct children (i.e., sub-packages, classes, interfaces, etc.). The split operation enables the engineer to *zoom in* into the contents of an architectural component when re-planning its internal organization.

We position constituent pins in a circle with noticeable gaps in between clusters of strongly coherent pins (inspired by a technique by Hoff *et al.* [58]). The coherence between pins is computed based on a sibling linkage algorithm using references in the source code they represent. Figure 2 ⟨4⟩ depicts an instance of that in our example implementation. With this layout, we aim to support engineers in finding patterns in the freshly revealed sub-structure, based on which they might start re-organizing the pin layout manually. To implement the opposite direction, i.e., gaining an overview of which pins on a whiteboard represent software elements from the same or a co-located architectural component, our method uses a pin coloring scheme, which maps a unique color to each architectural collection ⟨5⟩. Sibling components receive similar colors to emphasize their local relationship.

Navigation along References: Regardless of the software element it represents, a pin's detail view maintains two lists of references to elements in the subject system as shown in Figure 2 ⟨3⟩ and ⟨7⟩: on the left-hand side the "incoming references" list contains one entry for each software element in the system that has a code-level reference to the pinned element; on the right-hand side, the "outgoing references" list contains one entry for each element that the pinned element has a reference to. Engineers can use the two lists to navigate the code-level relationships of a pinned element and also attach pins for related software elements.

Freehand Drawing and Writing: Engineers can pick up a virtual pen and freely sketch on the surface of a virtual whiteboard, e.g., to outline a selection of attached pins in a group or to make comments. This enables them to use arbitrary notations in the form of their own freehand drawings. Engineers can choose between different pen colors and remove previously drawn pen strokes with an eraser. An operation stack with undo and redo functionality, as well as features for duplicating a whiteboard, changing its size, and resetting it provide engineers with further means for flexible sketching and low change costs.

Ⓑ *Semi-Automated Interpretation of Freehand Sketches*

Figure 4 shows a meta model of the sketched diagram structure of our virtual whiteboards.

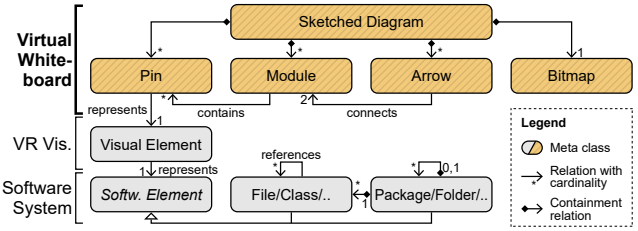

Fig. 4. Meta model for the sketched diagram structure of our method.

Each diagram includes a simple bitmap that stores engineers' sketches in terms of colorized pixels. It also maintains a list of pins that engineers attached to a whiteboard, including their position and a mapping to both the pinned visual element and the represented software element. Our method interprets drawn sketches (Ⓑ in Figure 3) in terms of two fundamental kinds of shapes to lay the foundation for subsequent automated analysis and operations. Our goal is to achieve a solution that is as automated and reliable as possible while maintaining engineers' freedom in their choice of visual notations.

To maximize reliability while ensuring a high degree of automation and flexibility, our method expects user input on the kind of visual element they currently sketch on a whiteboard. That is achieved by letting engineers switch between different drawing modes. We limit this input to the two most fundamental types of elements used when visually representing elements and relations between them [10], [14], i.e., (i) outlines around elements (pins on a whiteboard) that group these into what we refer to as *modules* and (ii) arrows between modules to express directed *relations*. In addition, we incorporate a third mode for uninterpreted drawing.

Uninterpreted Drawing ⟨8⟩***:*** Per default, engineers draw on a whiteboard without having their pen strokes interpreted as visual elements, e.g., to write textual comments or to draw symbols and icons. We persist each pen stroke in the colored bitmap and register operations in the undo stack.

Module Outlining ⟨9⟩***:*** When editing a sketch in module drawing mode, our method automatically interprets freehand drawn shapes as outlines around pins on the whiteboard and assigns included pins as the containment of the module (see Figure 4). These outlines can be arbitrarily shaped, so that engineers can freely decide on the visual notation they use.

We do not specify semantics to modules but, instead, leave this decision to the engineer and/or visualization that embeds our method. In our example visualization in Figure 2, modules have no semantics beyond grouping together classes and packages. When engineers interact with the pins on a whiteboard, e.g., by grabbing and repositioning them on the whiteboard, our method automatically updates its internal model to re-evaluate the module contents.

In cases where two (or more) module outlines are nested or intersecting one another, pins are assigned to all modules that contain them (cf. Figure 4). While this means it is not possible to construct module hierarchies – at least not in the underlying model structure, visually it is of course possible (cf. Figure 2 ⟨11⟩) – this behavior is easy-to-grasp for users and flexible because it allows for arbitrary module shapes.

Relation Arrow Drawing ⟨10⟩***:*** Previous work shows that developers draw relations between elements as directed arrows [10]. Our method includes a relation drawing mode in which it automatically interprets arrows between the borders of previously drawn modules as relations between these and updates the sketched diagram model accordingly (cf. Figure 4). Our method determines which modules a sketched arrow connects by finding the closest module to the arrow's start point and the closest module to the arrow's end point respectively. This approach allows engineers to draw self references. To make the direction of sketched arrows explicit, our method automatically completes them with arrow tips at the end (cf. Figure 2 ⟨10⟩). By explicitly modeling the relations between modules in a sketch, we lay the foundation for subsequent automated steps, especially the visual superimposition of code-level relationships.

Ⓒ *Visual Superimposition of Code-Level References*

To support engineers with establishing and maintaining an overview of the relations between software elements on a virtual whiteboard, our method automatically superimposes code-level references via arced, semi-transparent lines between the respective pins on a whiteboard (Ⓒ in Figure 3). Figure 2 ⟨11⟩ shows examples where pins represent classes in a Java system and superimposed reference lines between them are based on method calls, field accesses, and type references. The thickness of reference lines indicates their weight (number of references to another), while a texture on the lines indicates their direction, which is further emphasized via a subtle animation.

Layout: To avoid occlusion of superimposed reference lines, they bend perpendicularly to the normal direction of the whiteboard depending on how far apart the connected pins are located (see Figure 2). This achieves a layout where a reference line between a pair of pins that is far apart bends further out than a line connecting two nearby pins. In case of mutual references between two pins, i.e., two pins are connected by two superimposed lines (one in each direction), these are slightly bent in a counter-clockwise rotation.

Color: Per default, our method renders reference lines as semi-transparent black lines. To provide engineers with visual feedback on their freehand drawn arrows (see Section III), our method displays superimposed reference lines in the same color as an arrow if they match the arrow's path through the structures depicted in a sketch. Examples of that are depicted in Figure 2 ⟨11⟩, where reference lines between two module in one direction are colored in red due to a red freehand sketched arrows between them. Thereby, our method provides engineers with continuous automated feedback on their freehand sketches in the form of conformance checks with the ground-truth relations between software elements (Ⓔ in Figure 3), similar to the reflexion modeling approach by Murphy *et al.* [8], [34]. The key difference is that Murphy *et al.* employed a deliberately limited modeling notation and required a manual triggering of the conformance checks at discrete time points, whereas our approach captures models in the form of flexible freehand sketches while continuously providing instant conformance feedback. In combination with the workflow of pinning and repositioning software elements on a whiteboard, our method thereby achieves quick cycles of visualizing and reflecting which are tied in closely with ground-truth information on a system's architectural structure.

Ⓓ *Integration with IDE and Automated Code Generation*

When it comes to implementing planned changes, i.e., performing statement level edits to a system's code, we argue that the most suitable tool are IDEs with their well-established features and user interfaces. To make insights and plans sketched in VR available in an IDE, our method includes an automated synchronization that mirrors diagrams from VR to the IDE, where engineers are then able to zoom and pan in the sketch as well as to click on pins to jump to the respectively mapped source code artifacts such as a class.

To support engineers in implementing a plan captured in a freehand sketch, our method provides automated code generation operations (Ⓓ in Figure 3). These operations are based on the model structure of a plan (see Figure 4), allowing for coarse-grained operations, such as the generation of an interface for a freehand drawn module.

IV. Evaluation

The overarching research objective we aim to address with our method is supporting engineers in representing and reflecting on views and plans on architecture-level software structures. We evaluate to what extent our method achieves this objective by answering two research questions.

RQ_1: How does VR freehand sketching support engineers in representing architecture-level software structures?

RQ_2: How does VR freehand sketching support engineers in reflecting on architecture-level software structures?

We collected qualitative data to answer these research questions via an iterative evaluation with software engineering practitioners from companies located in 3 different countries (anonymized, anonymized, anonymized).

TABLE I
SHORTENED VERSION OF THE OVERARCHING TASKS OF THE CASE STUDY (COMPLETE INTERVIEW GUIDE IS AVAILABLE IN OUR ONLINE APPENDIX)

Task	Task Description (shortened)	Simulated Activity	RQs
Task$_1$	Sketch the package "svgbrowser" with its constituents classes: identify a sub-composition into clusters of classes with strong cohesion and weak coupling.	Representing the inner structure of an architecture-level software element	RQ$_1$, RQ$_2$
Task$_2$	Analyze how the package "swing" is related with the package "svgbrowser": which classes are responsible for the relation from "swing" to the "svgbrowser"? Annotate that you want to change this relationship between the packages.	Analyzing an interrelation between architecture-level software elements and planning to change it	RQ$_1$, RQ$_2$
Task$_3$	Analyze which classes of the two previously investigated packages are potentially affected by the changes planned in Task$_2$.	Reflecting on planned changes (via a change impact analysis)	RQ$_2$

We divided our evaluation into three iterations, which each consist of (i) a study phase where we let participants solve tasks using an implementation of our method while collecting qualitative data in a semi-structured interview and (ii) a development phase in which we improved our concepts and tool based on the results of the preceding evaluation phase. Figure 5 depicts this process.

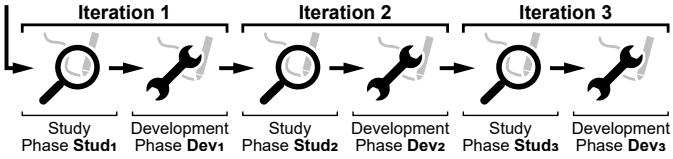

Fig. 5. Structure of our iterative evaluation. We alternated between study phases where we tested our method with participants in a case study and development phases where we improved and extended it.

The groups of participants were mutually exclusive across the iterations with a distribution as follows:

- *Iteration 1*: Four developers from one company
- *Iteration 2*: Three developers from two companies
- *Iteration 3*: One developer from one company

A. Study Phases

The study consisted of tasks which guided the participants through a step-wise re-engineering preparation for the open-source software system "Apache Batik" (~2.600 Java classes). To solve the tasks, participants used an implementation of our VR freehand sketching method. After each task, we queried participants' verdict on the support they receive from our method. The average duration of sessions was ~1 hour.

Tasks: The evaluation phases of our case study are organized into three tasks. Table I provides shortened versions of these along with the activity they required from participants and the research question they cover respectively. The tasks were consistent across all iterations. A complete version of our interview guide with more elaborate task descriptions and questions is available in our online appendix[2].

In Task$_1$, we asked participants to analyze an example application package (38 directly contained classes and interfaces, no sub-structure). The idea was to identify cohesive groups and represent the resulting structure on a whiteboard.

[2]https://doi.org/10.6084/m9.figshare.22710490

In Task$_2$, we asked participants, first, to investigate the relationship of the example application investigated in Task$_1$ to a package that it builds upon and, second, to re-plan it. To achieve a plausible scenario for that task, we deliberately introduced questionable design decisions into the subject system as preparation for the case study. That is, we added method calls that resulted in a mutual dependency between the example application from Task$_1$ and the package it builds upon.

In Task$_3$, we asked participants to reflect on the change they had planned in Task$_2$ via a change impact analysis.

Questions: After each task, we collected qualitative feedback from participants via open questions. These were consistent across all iterations.

- How would you usually [*solve this task*]?
- How do you assess the support you receive from the virtual whiteboard for [*solving this task*]?

→ Do you see benefits over your usual approach?
→ Do you see drawbacks compared to your usual approach?
→ Do you miss functionality that would be helpful?

We substituted task-specific terms in the questions above. Full descriptions of each task and question are available in our online appendix[2].

Analysis: We recorded videos of each participant's point of view in VR along with audio of their responses to our questions. After each iteration we analyzed the recordings by transcribing them and applying an open coding procedure. First, we highlighted verbatim statements in the transcript that we identified as relevant for answering our research questions. Second, we grouped these verbatim statements based on their core statement. Tables with details on these two steps are available in our online appendix[2]. Third, we sorted participants' core statements and established categories among them. We also identified recurring topics on the verbatim statements, orthogonal to the established categories. Figure 6 depicts a graphical representation of our results after Iteration 3.

B. Development Phases

Subsequent to each study phase, we conducted a development phase in which we addressed identified problems and suggestions. We discuss relevant instances of these in Section IV-C. More detailed descriptions of changes with a mapping to verbatim statements of participants can be found in our online appendix[2].

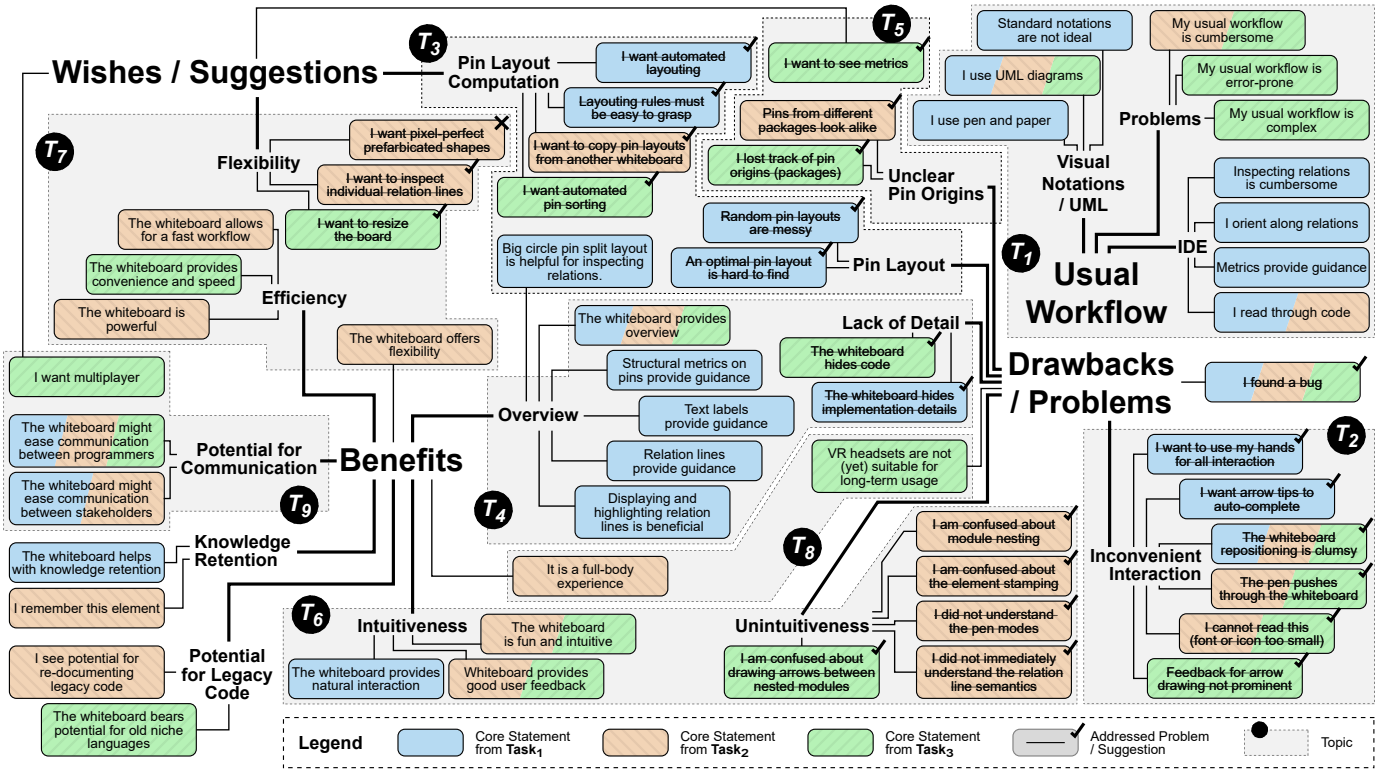

Fig. 6. Overview of participants' core statements during the three study phases organized into a hierarchy of categories and recurring topics. Each core statement summarizes one or more verbatim statements. A complete table with the mapping from statements to topics is available in our online appendix[2].

In the following, we elaborate on our technical implementation[1], which we provided to participants.

Implementation: We integrated the concepts presented in Section III into the open-source VR software visualization tool "Immersive Software Archaeology" (ISA)[1]. Figure 2 depicts screenshots of that implementation. ISA already provides users with immersive functionality for grabbing and moving its visual elements. This integrates well with our method. Because our research objective is concerned with architecture-level software structures, we built upon ISA's existing grabbing functionality to allow participants to pin packages and classifiers (classes, interfaces, and enums) of a Java subject system.

C. Participants' Answers

During the analysis of our interview transcripts (see Section IV-A), we extracted recurring topics in participants replies to our questions. In the following, we elaborate on these (T_1-T_9). We highlight a number of potential obstacles in technical implementation of our method and how we addressed them as well as conceptual challenges that participants identified. The graphical representation in Figure 6 provides an overview of the core statements made by the participants, grouped into categories and summarized into topics (T_1-T_9). Furthermore, we annotate whether any of these statements referred to issues that were resolved during the prototype's development (indicated by strikethrough in the figure).

T_1) Usual Tools Are Not Ideal: A majority of participants stated that their usual approach for solving $Task_1$ (sorting package contents) would entail graphical notations, most notably in UML. One participant emphasized that relations between classes are hard to track in class diagrams. In contrast, participants answered to usually solve problems like $Task_2$ (analyzing and re-planning a relation between packages) purely by reading through code with metrics and references as guidance, e.g., "*looking at each class* [in an IDE] *and seeing if it is being used in the browser app or not.*" When describing their usual workflow for tasks similar to $Task_3$ (impact analysis for the changes planned in $Task_2$), participants reported on problems with their current practice. One participant stated to use an IDE for similar tasks and continues "*It's time consuming. It's possible, but it's time consuming*" while another concludes after solving $Task_3$ "*It would be very complex. Much more complex than what just happened now.*" Another participant comments "*Oftentimes, I do not have documentation or visualizations. Oftentimes, I only have code* [..] *which I have to consider as black box.* [..] *If I now have a possibility to say 'okay, I draw a rectangle' and then I say 'okay, it includes this and that method and so on', then I can myself document code that was previously undocumented in an easy way.*"

T_2) Whiteboard Interaction Must be Realistic: In Iterations 1 and 2, participants made remarks on VR interactions in our concrete technical implementation which they perceived as cumbersome to use.

We addressed all encountered obstacles in the development phases of our evaluation. Because these points are specific

to the concrete technical implementation of our method, we refer to our online appendix[2] for further points and details. We discuss two aspects in the following which we deem relevant for implementations of our (or similar) concepts.

Whiteboard Repositioning: In our initial implementation, whiteboards snapped to the user's hand when grabbed, adopting its position and rotation. This led to confusion. We addressed this problem in Dev_2 by reworking the grabbing mechanism to attach to the user's hand relative to its current position and rotation so that when grabbed, the whiteboard always remains in place until the user moves the grabbing hand.

Pen Clipping: In the real world, physical whiteboards provide feedback on when a pen touches them simply because the objects collide and the pen cannot be pushed further. In VR, this is not the case. Users can move their physical hand further although, in the virtual space, a whiteboard should block their movement. Major challenges for implementing our method are (i) providing users with feedback on when their pen touches the whiteboard and (ii) preventing, to some degree, their virtual hand and pen to clip through a whiteboard although the physical hand might move further. We solved these challenges (i) via haptic feedback on the VR controllers (vibrating upon touching the whiteboard with the pen tip) and (ii) by temporarily disconnecting the physical and virtual hand's synchronization and projecting the virtual pen on the surface of a whiteboard when users push their hands too far into a it.

T_3) Layout Algorithm for Splitting Pins Should be Intuitive: In iteration 1, we used a simple algorithm to determine the layout of split pins by randomly searching for unoccupied space on the whiteboard.

Participants complained about that: "*Spaghetti. Yeah, it's very messy!*" While manually sorting the randomly positioned pins, they were wondering "*Why is it me who's doing that?*" For the subsequent iteration, we implemented layout algorithms that built on a clustering technique [58] to group together pins for interrelated software elements, i.e., (a) spawning one circle of pins for each identified cluster and (b) arranging all pins in a big circle with pins clustered together as neighbors and noticeable gaps to other clusters (see Section III). Strategy (a) was not perceived as intuitive and, thus, helpful because splitting one pin resulted in multiple different circles. One participant remarked "*This layout seems to follow some concept. And if I do not understand that concept* [..]*, it does not help me.*" before going over to manually arranging the pins according to strategy (b). Therefore, we decided for strategy (b) in Dev_2 which, in comparison with strategy (a), was again assessed as beneficial in Iteration 3 ($Task_1$): "*I like as basic layout this outer ring* [of pins]*, because it keeps the center tidy, and also it provides a maximum transparency for the* [reference] *lines.* [..] *And then I can say 'okay, this* [pin] *seems relevant, I put it in the center'.*"

T_4) VR Whiteboards Provide Overview But No Code: Participants across all iterations commented on the overview of the software structures they represented with our method.

One participant (Iteration 1) reported on a lack of detail due to the high level of abstraction, pointing out that a more thorough answer to $Task_2$ would require to read through source code: "*This would be just a starting point for understanding where to investigate*" while another remarks "*I think it is a great tool for the overview and a lesser great tool for the detailed look.*" after finishing $Task_3$ (Iteration 2).

The high level of abstraction in our method was perceived as positive. Participants across all iterations and tasks mentioned that using our method provided them with a good overview over what they have sketched compared to their usual approach, e.g., "*In this whiteboard, I have a clearer overview of everything.*" (Iteration 1, $Task_1$) One participant emphasized particularly the reference lines between pins: "*This isolated class would maybe be hard to find [in code] because it has no relations. But here it popped into my view.*" (Iteration 2, $Task_1$)

T_5) Pins Should Visualize Meta Information: In our initial implementation (used in Iteration 1), all pins for classes had an identical appearance, i.e., white cylinders without further geometry. We received multiple comments on this:

Visualizing Metrics: One participant remarked "*Some of the metrics* [from the original visualization] *are missing.* [..] *They would help me pinpoint faster which are the classes that have problems.*" We implemented this suggestion in Dev_1 by displaying a small avatar of the respective represented visual element on each pin (see Section III). Because visual elements in software visualizations are usually generated based on relevant metrics for the software elements they represent, the avatars on pins communicate these metrics as well. In subsequent iterations, we observed that the avatars on pins helped participants with identifying relevant source code entities, e.g., only a few seconds after seeing all 38 pins of the package in $Task_1$, one participant grabs a pin for a large class and states "*I want to put this aside to demonstrate it is a central element, from the relations and its size alone.*"

Visualizing Pin Origins: A problem participants reported on in Iteration 1 was keeping track of pin origins, e.g., "*Did that* [pin] *come from here or there? So that will kind of confuse my box thinking.*" We addressed that problem in Dev_1 by coloring pins according to the subsystem they stem from (see Section III). It was not brought up in subsequent iterations.

T_6) Module and Relation Sketching is (Mostly) Intuitive: Across all iterations and tasks, intuitiveness was often mentioned. We received critique and suggestions regarding unintuitive controls for the earlier stages of our implementation. Because we consider them relevant for technical implementations of our method, we report on the two most notable instances, both caused by insufficiently explained tool functionality. Both instances could be resolved with a short explanation during the respective sessions. We addressed them via explanations in the UI.

Drawing Nested/Overlapping Modules: One point of confusion brought up by two participants in Iteration 2 ($Task_2$ and $Task_3$) were the semantics of multiple modules outlining one or more common pins (i.e., the modules are overlapping or nested into one another).

For instance, one participant drew a large blue module around ~30 pins of which 4 were already outlined by a smaller yellow

module. The participant stopped for a moment, pointed at the pins and wondered "*Are these both blue and yellow in principle?*" Although the behavior of our method in such cases was easy to grasp for the participants once explained (in the instance above, both modules indeed contained the 4 pins in question, cf. Section III), it was not obvious to them initially.

Reference Line Directionality: A point of confusion for one participant in Iteration 2 were the semantics of the reference lines between pins. Instead of the "calls" relationship that they display, one participant interpreted the direction of the lines as "is called", leading to wrong assumptions in $Task_3$. The majority of comments on our method's intuitiveness was positive across all iterations and tasks. One participant reported "[..] *it feels quite intuitive. It's fun to keep grabbing* [pins] *and moving them.*" (Iteration 2, Task 2). Another participant comments "*It seems quite visually intuitive.*" (Iteration 2, Task 1).

T_7) Good Efficiency and Flexibility: Participants in all iterations highlighted a high degree of flexibility and efficiency in our method. While drawing modules and continuously moving pins around to solve $Task_2$, one participant in Iteration 1 comments "*This is very powerful. I can, one, define modules and, two, I can use the tool to provide me visual hints on the kind of relationships and so on.* [..] *This is way better* [than my usual approach]. *I mean, here I can see what I have to do. Yeah, it's very cool.*" Another participant comments after solving $Task_3$: "*The benefit in this approach here with this whiteboard and especially these relations is being able to see very quickly and in a very dynamic way – because I was able to move classes around – where the dependencies lie and in which direction they are. It's just way faster than anything I would do with an IDE, for example, because IDEs usually just let you do one thing at a time. In this case, instead, it's like doing these kinds of analysis in parallel because I'm doing it for* [multiple] *classes at the same time.*"

T_8) A Full-body VR Experience: While solving the tasks, participants were using their bodies extensively – especially their arms. They reached out to pins all over their whiteboards, scribbled, outlined and wrote annotations, stepped back to reflect over their drawing, walked to other visual elements in the embedding visualizing to pin them on the board, and so on. While this workflow contributes to the aforementioned aspects of intuitiveness and flexibility, one participant pointed out: "*The drawback of VR is always that you need put on the headset and change the environment you are in. VR is a great supplementary tool, but you cannot use it for 8 hours a day, that would not be pleasant at the state that VR is in right now.*" Another participant comments: "*You are requiring your body to be more used.* [..] *It's a full body experience. I like it personally, but it's something to keep in mind. Why should people be standing and using their whole body? What does it add? It needs to add something. I think it does in this case.*"

T_9) Potential for Communication Purposes: Across all tasks, participants hypothesized a usefulness of our method for communication purposes.

Two participants each described scenarios where they imagined using our method to demonstrate software structures to other stakeholders on-screen, e.g., "*I think it would be beneficial, especially* [for] *a project manager or even a client trying to understand the complexity of something. You could have a shared dialog in a more visual way.*" Another participant imagined using the method to communicate with peers: "*If I were to communicate with someone else about this, it would be much easier for me to introduce them to my thoughts here than clicking through a thousand* [IDE] *windows and references. This is much easier.*" To facilitate the latter scenario, the participant suggested functionality that allows multiple users to enter the same virtual world simultaneously to collaboratively edit and view VR whiteboards.

D. Answers to Research Questions

We use the results presented in T_1-T_9 above to answer our research questions in the following, highlighting both advantages and disadvantages.

RQ_1: How does VR freehand sketching support engineers in representing architecture-level software structures?

To answer this research question, we identify feedback and comments related to pinning elements and sketching in different pen modes to persist views and plans on software structures.

Among that feedback were problems with the VR controls (T_2), requests for improved pin layout algorithms (T_3), suggestions for more meta information on pinned elements (T_5), and more unintuitive functionality (T_6). In Dev_1 and Dev_2 (Figure 5), we addressed all points that emerged directly from the technical implementation (e.g., problems with the VR controls, T_2) and extended our concepts and implementation for all those that required further work on our method (e.g., adding semantic structure to pins via color and avatars, T_5).

Positive feedback and perceived strengths of our method related to RQ_1 were its intuitiveness (T_6) and its high degree of flexibility (T_7).

The workflow of pinning software elements on a diagram and simply drawing on it was perceived as powerful and efficient (T_7): "*I was able to just do it all at once: Just selecting all the classes and* [it was] *telling me* [..] *which of these classes are being used in the browser* [package]. *In that sense, it was way faster.*" This has shown to be particularly useful for planning changes to the depicted software structures in $Task_2$, because participants were able to freely position, outline, highlight, and annotate software elements and their interrelations. Other key features were those that allowed participants to navigate along the software hierarchies in a sketch, i.e., splitting pins for architectural units (T_3) while maintaining an overview of the pins' origins (T_5).

RQ_2: How does VR freehand sketching support engineers in reflecting on architecture-level software structures?

To answer this research question, we consider feedback and comments related to participants having high-level reflections over (a) the software structures depicted in their sketches ("is this what the system looks like?") and (b) the sketches per se ("should it really look like this?").

Potential problems we identified were unintuitive semantics of reference lines (i.e., directionality representing "calls" versus

"is called", T_6). This was addressed in the subsequent development phase via UI explanations. For another, it was a perceived lack of detail in the drawn diagrams due to the unavailability of source code (T_4) reported by multiple participants. This is the most relevant critique of our method. On one hand, our method deliberately abstracts from target languages and leaves it to the embedding software visualization to display source code (should it allow this at all). This has advantages in terms of overview (T_4), programming language independence, and communication (T_9). On the other hand, using our method for in-depth reflections on software structures "[..] *would be just a starting point for understanding where to investigate*" (T_4).

The majority of comments relevant for RQ_2 are positive: The level of abstraction employed by our method provided participants with a good overview of the depicted structures (T_4), especially when compared to participants' usual approaches (T_1): "*it's faster to get an overview* [..] *for a task of identifying which are the problematic classes.*" Reference lines between pins and the continuous checks on their conformance to drawn arrows were emphasized as particularly helpful, e.g., "*It's a great tool for having an overview, especially when you're getting into a very complicated repository and packages are cross-referencing as much as this is.*" Lastly, participants saw potential in our method for jointly communicating and reflecting about view, ideas, and problems with peers and other (non technical) stakeholders (T_9).

Summary: How does our method support engineers in preparing for architectural re-engineering?

Our method has positive effects on participants' ability to represent and reflect on software structures. Although our study showed that the high level of abstraction in our concepts comes at the cost of an unavailability of details, it was overall perceived as flexible, powerful, and visually intuitive with potential value for dissemination purposes. A more extensive answer to how our method supports engineers in preparing architectural re-engineering requires further investigation into the full re-engineering circle, i.e., letting engineers enact plans made on our virtual whiteboards by performing code changes. The data gathered in this study alone demonstrates that our method supports engineers in externalizing views on software structures and plans to change these and, thus, that it facilitates preparing for architectural re-engineering.

E. Reflections on the Evaluation

We collected qualitative data to answer our research questions. In the following, we critically reflect on that process.

Number of participants: One aspect to consider is the number of participants in our study, i.e., 4 in study phase $Stud_1$, 3 in $Stud_2$, and 1 in $Stud_3$. Having only one participant in $Stud_3$ entails a risk to the evaluation of changes made in Dev_2. However, because we did not conceptually change our method in Dev_2, this risk pertains only to implementation aspects, mostly regarding interaction problems noted in $Stud_2$ for concepts developed and implemented in Dev_1.

Biases in Feedback and Analysis: The most critical risk to the results of our evaluation are potential biases in our participants' feedback and our analysis. To mitigate these, we took several countermeasures. For one, to mitigate biases towards answers that favor certain theories over others, we formulated tasks and questions neutrally and in an open-ended style. We recorded these in an interview guide[2], which additionally makes participants' statements and assessments comparable. As part of that, we explicitly invited both positive and negative feedback. For another, to mitigate biases in our analysis, we recorded and transcribed video and audio footage of each session to analyze participants' feedback verbatim and in the context of what they were doing.

V. Conclusion and Future Work

We presented a method for freehand sketching views and plans on architecture-level software structures in virtual reality. Our method integrates with the model structure of an embedding VR software visualization to automatically (a) augment sketches with information on relations between depicted elements and (b) provide instant conformance checks of sketches with the represented source code.

We evaluated our method in a qualitative study with 8 software engineering practitioners from 4 companies across 3 countries. Our results show that participants' main point of critique was a perceived lack ot detail due to the high level of abstraction employed by our method. For the same reason, however, they strongly emphasized obtaining a good overview over the structures depicted in their drawings. All in all, our method was perceived as flexible, efficient, and powerful with a high potential value for communication purposes and eased collaborative efforts.

In future work, we plan to conduct an empirical long-term study on the usage of VR freehand sketching of architectural views in industrial software development to observe practical impacts and benefits with larger participant numbers than presented in this work. That includes further investigations of our method for facilitating the collaboration between practitioners in exploring and re-documenting software systems.

We further plan to investigate supporting re-engineers with enacting changes planned with our VR freehand sketching method. In a first instance, this includes extending the currently available code generation capabilities of our method, so that re-engineers can start preparing source code changes based on drawn diagrams from within VR. Detailed code edits on the level of individual statements, however, should be done in a traditional 2D-screen IDE. Thus, we further plan to extend our method such that it transfers changes planned in a freehand sketch into lists of action items displayed in the IDE.

Acknowledgements

Hoff and Seidl are supported by the DFF (Independent Research Fund Denmark) within the project "Immersive Software Archaeology (ISA)" (0136-00070B). Lanza is supported by the Swiss National Science Foundation (SNSF) through the project "INSTINCT" (Project No. 190113).

REFERENCES

[1] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. [Online]. Available: http://scg.unibe.ch/download/oorp

[2] M. Beck, J. Trumper, and J. Dollner, "A visual analysis and design tool for planning software reengineerings," in *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Sep. 2011.

[3] L. H. Rosenberg and L. E. Hyatt, "Software re-engineering," *Software Assurance Technology Center*, 1996.

[4] R. Koschke and D. Simon, *Hierarchical Reflexion Models*, Dec. 2003.

[5] W. Lowe, M. Ericsson, J. Lundberg, and T. Panas, "Software Comprehension – Integrating Program Analysis and Software Visualization."

[6] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold, "Real-time reflexion modelling in architecture reconciliation: A multi case study," *Information and Software Technology*, vol. 61, 2015.

[7] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, Jan. 2007.

[8] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, 1995.

[9] M. Romanelli, A. Mocci, and M. Lanza, "Towards visual reflexion models," in *Proceedings of ICPC 2015 (23rd International Conference on Program Comprehension*. IEEE CS Press, 2015, pp. 277–280.

[10] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: how and why software developers use drawings," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Apr. 2007.

[11] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th international conference on Software engineering*, May 2006.

[12] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek, "How software designers interact with sketches at the whiteboard," *IEEE Transactions on Software Engineering*, vol. 41, no. 2, 2014.

[13] U. Dekel and J. D. Herbsleb, "Notation and Representation in Collaborative Object-Oriented Design: An Observational Study," 2007.

[14] B. Tversky, "What do Sketches say about Thinking?" 2002.

[15] Y. Y. Wong, "Rough and ready prototypes: lessons from graphic design," in *Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems - CHI '92*, 1992.

[16] S. Baltes and S. Diehl, "Sketches and Diagrams in Practice," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014.

[17] R. Müller and D. Zeckzer, "Past, Present, and Future of 3D Software Visualization - A Systematic Literature Analysis:," in *Proceedings of the 6th International Conference on Information Visualization Theory and Applications*, 2015.

[18] M. Lanza and S. Ducasse, "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, Sep. 2003.

[19] M. Lungu, M. Lanza, and O. Nierstrasz, "Evolutionary and collaborative software architecture recovery with Softwarenaut," *Science of Computer Programming*, vol. 79, Jan. 2014.

[20] R. Minelli and M. Lanza, "SAMOA – A Visual Software Analytics Platform for Mobile Applications," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013.

[21] M. Balzer and O. Deussen, "Hierarchy Based 3D Visualization of Large Software Structures," in *IEEE Visualization*, 2004.

[22] ——, "Level-of-detail visualization of clustered graph layouts," in *6th International Asia-Pacific Symposium on Visualization*, 2007.

[23] J. Maletic, "Visualizing Object-Oriented Software in Virtual Reality," 2001.

[24] R. Brito, A. Brito, G. Brito, and M. T. Valente, "GoCity: Code City for Go," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2019.

[25] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live trace visualization for comprehending large software landscapes: The ExplorViz approach," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2013.

[26] M. Lanza, H. Gall, and P. Dugerdil, "EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009.

[27] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "CityVR: Gameful Software Visualization," 2017.

[28] J. Vincur, P. Navrat, and I. Polasek, "VR City: Software Analysis in Virtual Reality Environment," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2017.

[29] R. Wettel and M. Lanza, "Visualizing Software Systems as Cities," in *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Jun. 2007.

[30] ——, "CodeCity: 3D visualization of large-scale software," in *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, 2008.

[31] R. Wettel, M. Lanza, and R. Robbes, "Software systems as cities: a controlled experiment," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011.

[32] A. Hoff, M. Nieke, and C. Seidl, "Towards immersive software archaeology: regaining legacy systems' design knowledge via interactive exploration in virtual reality," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[33] M. Shahin, P. Liang, and M. A. Babar, "A systematic review of software architecture visualization techniques," *Journal of Systems and Software*, vol. 94, Aug. 2014.

[34] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, Apr. 2001.

[35] G. Murphy and D. Notkin, "Reengineering with reflexion models: a case study," *Computer*, vol. 30, no. 8, Aug. 1997.

[36] C. Ackermann, M. Lindvall, and R. Cleaveland, "Towards Behavioral Reflexion Models," in *2009 20th International Symposium on Software Reliability Engineering*, Nov. 2009.

[37] S. Herold, S. Counsell, M. English, M. Ó Cinnéide, and J. Buckley, *Detection of Violation Causes in Reflexion Models*, Feb. 2015.

[38] M. Romanelli, A. Mocci, and M. Lanza, "Towards Visual Reflexion Models," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015.

[39] E. Baniassad and G. Murphy, "Conceptual module querying for software reengineering," in *Proceedings of the 20th International Conference on Software Engineering*, Apr. 1998.

[40] R. A. Bittencourt, G. J. d. Santos, D. D. S. Guerrero, and G. C. Murphy, "Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques," in *2010 17th Working Conference on Reverse Engineering*, Oct. 2010.

[41] A. Le Gear, J. Buckley, J. Collins, and K. O'Dea, "Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling," in *2005 International Symposium on Empirical Software Engineering, 2005.*, Nov. 2005.

[42] T. Olsson, M. Ericsson, and A. Wingkvist, "To automatically map source code entities to architectural modules with Naive Bayes," *Journal of Systems and Software*, vol. 183, 2022.

[43] Z. T. Sinkala and S. Herold, "InMap: automated interactive code-to-architecture mapping," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, Mar. 2021.

[44] ——, "Towards Hierarchical Code-to-Architecture Mapping Using Information Retrieval," 2021.

[45] C. D. Hundhausen, "Using end-user visualization environments to mediate conversations: a 'Communicative Dimensions' framework," *Journal of Visual Languages & Computing*, vol. 16, no. 3, Jun. 2005.

[46] J. D. Herbsleb, B. Laboratories, and S. Boulevard, "Metaphorical representation in collaborative software engineering," 1999.

[47] N. Mangano, T. D. LaToza, M. Petre, and A. van der Hoek, "Supporting informal design with interactive whiteboards," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Apr. 2014.

[48] M. Suwa and B. Tversky, "External Representations Contribute to the Dynamic Construction of Ideas," in *Diagrammatic Representation and Inference*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds., 2002, vol. 2317.

[49] M. Suwa, J. Gero, and T. Purcell, "Unexpected discoveries and S-invention of design requirements: important vehicles for a design process," *Design Studies*, vol. 21, no. 6, Nov. 2000.

[50] D. A. Schön, *The reflective practitioner: How professionals think in action*, 2017.
[51] W. Ju, B. A. Lee, and S. R. Klemmer, "Range: exploring implicit interaction through electronic whiteboard design," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, Nov. 2008.
[52] T.-J. Nam, "Collaborative Design Prototyping Tool for Hardware Software Integrated Information Appliances," in *Virtual Reality*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and R. Shumaker, Eds., 2007, vol. 4563.
[53] E. D. Mynatt, T. Igarashi, W. K. Edwards, and A. LaMarca, "Flatland: new dimensions in office whiteboards," in *Proceedings of the SIGCHI conference on Human factors in computing systems the CHI is the limit - CHI '99*, 1999.
[54] J. Luo, "VR-Notes: A Perspective-Based, Multimedia Annotation System in Virtual Reality," 2020.
[55] R. Chung, P. Mirica, and B. Plimmer, *InkKit: a generic design tool for the tablet PC*, Jan. 2005.
[56] J. Grundy and J. Hosking, "Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007.
[57] C. H. Damm, K. M. Hansen, and M. Thomsen, "Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard," 2000.
[58] A. Hoff, L. Gerling, and C. Seidl, "Utilizing software architecture recovery to explore large-scale software systems in virtual reality," in *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022.