
Immersive Software Archaeology

Comprehending Software Structure through Interactive Visualization in Virtual Reality

Adrian Hoff

PhD Thesis

IT UNIVERSITY OF COPENHAGEN

Computer Science Department

Advisors: Christoph Seidl and Mircea Lungu
Submitted: May 2024



Declaration of Work

I declare that this thesis - submitted in partial fulfillment of the requirements for the conferral of PhD, from the IT University of Copenhagen - is solely my own work unless otherwise referenced or attributed. Neither the thesis nor its content have been submitted (or published) for qualifications at another academic institution in Denmark or abroad.

Adrian Hoff

Abstract

Comprehending existing software systems is an activity relevant for, e.g., preparing the re-engineering of a legacy system. It is often complicated by missing or incomplete documentation and the absence of a system's original developers, making source code the only reliable source of information. However, exploring a software system solely by reading through its source code is a tedious and challenging task that, if possible at all, requires significant stamina and motivation. Visualizations can offer an overview of otherwise complex source code bases, making them a valuable tool for assisting software engineers in this exploration process.

In this thesis, I present work on utilizing virtual reality (VR) to provide teams of software engineers with interactive, synchronized multi-user visualizations of source code, for the purpose of exploring software architecture while taking notes on insights and planning future actions. To foster software architecture exploration, the presented VR visualization methods are based on results from automated software clustering techniques. Further, they introduce concepts for mixing automatically interpreted freehand drawing in VR with multimedia annotations, so that engineers can explore software systems in long-lived sessions.

I demonstrate the presented visualization methods in tool implementations and assess via empirical studies their suitability for assisting engineers in comprehending software systems. The results of the studies show that VR software visualization is suitable for assisting engineers – especially those with less experience – in comprehending a subject system's architectural structure. Regarding note-taking, the studies demonstrate that freehand sketching in VR is useful for capturing high-level views on system architecture, while multi-media annotations (such as audio recordings) are a valuable means for more general notes. Lastly, the studies show that software engineers value synchronized multi-user VR software visualization for focused exploration sessions with vivid communication where even with little training and despite their unfamiliarity with VR, engineers are able to gain correct insights into a subject system.

Resumé

At forstå eksisterende softwaresystemer er en aktivitet som opstår, for eksempel, når man forbereder re-design af et ældre system. Aktiviteten bliver dog ofte kompliceret af manglende eller ufuldstændig dokumentation, og fraværet af de oprindelige udviklere. Dette gør kildekoden til den eneste pålidelige informationskilde, men at udforske et softwaresystem ved kun at læse kildekoden, er en tidskrævende og udfordrende opgave. Visualisering, der giver et overblik over den komplekse kildekode, er derfor et værdifuldt værktøj til softwareingeniører i denne udforskningsproces.

I denne afhandling præsenterer jeg brug af Virtual Reality (VR) til at give hold af softwareingeniører interaktive og synkroniserede multi-bruger visualiseringer af kildekoden. Disse visualiseringer kan bruges til at udforske softwarearkitekturen, mens ingeniørerne tager noter omkring indsigter og handleplaner. For at fremme udforskning af den overordnede arkitektur er de præsenterede VR-visualiseringsmetoder baseret på resultater fra automatiserede software-clustering-teknikker. Derudover introducerer metoderne koncepter til at blande automatisk tolkede frihåndstegninger i VR med multimedie-annoteringer, så ingeniører kan udforske softwaresystemer i længere sessioner.

Jeg demonstrerer de præsenterede visualiseringsmetoder gennem værktøjsimplementeringer, og vurderer gennem empiriske studier hvor egnet metoderne er til at hjælpe ingeniører med at forstå softwaresystemer. Studierne viser, at VR-softwarevisualisering er særligt velegnet til at hjælpe ingeniører - især dem med mindre erfaring - med at forstå et systems arkitektur. Studierne viser også, at frihåndstegning i VR er nyttigt til at opnå et overblik af systemarkitekturen, mens multimedie-annoteringer, såsom lydoptagelser, er værdifulde midler til generelle noter. Endelig viser det sig, at ingeniører værdsætter synkroniserede multi-bruger VR-visualiseringer, da de muliggør fokuserede udforsknings-sessioner med livlig kommunikation. Selv med lidt træning og på trods af manglende erfaring med VR kan ingeniører således opnå værdifuld indsigt i et systems struktur.

Zusammenfassung

Bestehende Softwaresysteme zu verstehen ist entscheidend, beispielsweise zur Vorbereitung der Überarbeitung eines Altsystems. Häufig fehlen jedoch vollständige Dokumentation oder die ursprünglichen Entwickler, sodass der Quellcode die einzig verlässliche Informationsquelle ist. Die Erkundung eines Softwaresystems allein durch Lesen des Quellcodes ist jedoch mühsam und herausfordernd, sodass dies viel Ausdauer und Motivation erfordert. Visualisierungen, die einen Überblick über den komplexen Quellcode bieten, sind daher wertvolle Hilfsmittel für Softwareingenieure.

In dieser Dissertation präsentiere ich Methoden zur Nutzung von Virtual Reality (VR), um verteilten Teams von Softwareingenieuren interaktive, synchronisierte Quellcode Visualisierungen zu bieten zur Erkundung von Softwarearchitektur und Erstellung von Notizen über gewonnene Erkenntnisse und Pläne. Um die Erkundung auf Architekturebene zu fördern, basieren die vorgestellten VR-Visualisierungsmethoden auf Ergebnissen von automatisierten Software-Clustering-Techniken. Darüber hinaus führen sie Konzepte ein, um automatisch interpretierte Freihandzeichnungen in VR mit Multimedia-Anmerkungen zu kombinieren, sodass Ingenieure Softwaresysteme in längeren Sitzungen erkunden können.

Ich demonstriere die vorgestellten Visualisierungsmethoden anhand von Tool-Implementierungen und bewerte ihre Eignung zur Unterstützung von Ingenieuren beim Verständnis von Softwaresystemen durch empirische Studien. Die Studienergebnisse zeigen, dass VR-Softwarevisualisierung besonders für weniger erfahrene Ingenieure geeignet ist, um die Architektur eines Systems zu verstehen. Die Studien zeigen auch, dass Freihandskizzen in VR nützlich sind, um Übersichten der Systemarchitektur zu erfassen, während Multimedia-Anmerkungen, wie Audioaufnahmen, wertvolle Hilfsmittel für allgemeine Notizen darstellen. Schließlich wird deutlich, dass Ingenieure synchronisierte Multi-User VR-Visualisierungen schätzen, da sie fokussierte Erkundungssitzungen und lebhaftere Kommunikation ermöglichen. Selbst mit wenig Training und trotz fehlender Erfahrung mit VR können Ingenieure so wertvolle Einblicke in die Struktur eines Systems gewinnen.

Acknowledgments

Many people contributed to the fact that I write this short section of acknowledgments – too many to enumerate all by name. In the following, I go through notable mentions in somewhat chronological order and ask for forgiveness from everyone I left out: sorry!

I thank my wonderful family, that is, my parents Martin and Christiane and my sister Julika, for their continuous and unconditional support and love in everything I do. I will never forget how my father taught me the fundamentals of programming in BASIC. Dankeschön!

I thank Ina Schaefer for introducing me to the world of academia through a position as a research assistant where I met Christoph Seidl, my future PhD supervisor, and Michael Nieke, a future collaborator and friend. With her institute at TU Braunschweig, Ina provided me with a great research environment, which laid the first stones of my academic path. Most notably, Ina funded a trip to my first international research conference as a graduate student in 2019 – I know now that this was not something many graduate students get to do. This experience played a pivotal role in making up my mind about whether I wanted to continue on a journey in academia after my Master's degree and I am grateful for it.

I thank my supervisor Christoph Seidl who played a key role in initiating my academic path years before the start of my PhD studies. To me, Christoph has, without exception, been a supportive, reliable, and inspiring mentor full of ideas and never short on feedback. I enjoyed working with Christoph very much and I am grateful for the many things I was able to learn from him.

I thank Mircea Lungu for taking over the role of supervisor under unexpected circumstances and fulfilling it as the supportive academic guide he is. I especially thank Mircea for many inspiring chats and valuable discussions – I gained a lot through these!

I thank Michele Lanza for hosting me for several months during my stay at his research institute in Lugano and for his active and invaluable role in our joint collaboration, which greatly influenced this thesis. I am grateful for Michele's interest in my work and for his support in establishing connections to other researchers that, without his active role, I don't think I would have made – it is *greatly* appreciated!

I thank all the fantastic people I had the honor to meet during my time as a PhD student at the IT University of Copenhagen. First and foremost, this includes the Software Quality Research (SQUARE) group for great discussions, lovely events – especially our annual trips – and their relentless commitment to an oddly specific lunchtime rule (11:23#). Most notably, I thank Laura Weihl for being a great colleague and friend, always prepared to lend a sympathetic ear. I thank my Danish pals Morten Clausen and Kasper Berthelsen for helping me translate relevant text snippets to Danish (such as the afore-printed abstract).

I thank Paolo Tell, Jesus Gonzalez-Barahona, and Craig Anslow for taking the time to review this thesis and providing me with feedback.

Last but far from least, I thank my girlfriend Malena for supporting me all the way through my PhD studies. It is due to Malena that I made it through the various deadlines for conference submissions – and the hand-in of this thesis – without serious (known) damage. Malena assisted me in improving and debugging concepts and code, tested VR interactions (especially those that required collaborators) and, thus, often acted as a first pilot of my studies, catching many important imperfections. Multiple figures printed in this thesis show virtual avatars of Malena exploring software in VR. Most importantly, however, Malena helped me find a balance between research and private life which I believe was the most relevant contribution to this thesis.

Thank you!

Contents

I	Summary	1
1	Introduction	3
2	Background	7
2.1	Software Visualization	7
2.2	Software Visualization: 2D vs. 3D	8
2.3	X-Reality for 3D Software Visualization	12
2.4	Software Visualization in 2D	15
2.5	Software Visualization in 3D	15
2.5.1	Visual Metaphors in 3D Software Visualization	18
2.5.2	Software Architecture Visualizations in 3D	19
2.6	Software Visualization in XR	22
2.6.1	Note Taking in Long-Lived XR Software Exploration Sessions . . .	22
2.6.2	Collaborative Software Exploration via XR Visualization	23
3	Problem Definition	25
3.1	Research Questions	26
3.2	Thesis	26
4	Solution Overview	27
4.1	Methodology	28
4.2	Contributions per Paper	29
4.2.1	Towards Immersive Software Archaeology: Regaining Legacy Systems' Design Knowledge via Interactive Exploration in Virtual Reality (Paper A)	29

4.2.2	Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality (Papers B and C)	32
4.2.3	Uniquifying Architecture Visualization through Variable 3D Model Generation (Paper D)	38
4.2.4	Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality (Paper E)	41
4.2.5	Collaborative Software Exploration and Note-Taking (Papers F and G)	47
4.3	Summary of Contributions	54
5	Conclusion	57
5.1	Fostering Exploration of Software Architecture	57
5.2	Note-Taking in Virtual Reality Software Visualization	59
5.3	Collaborative Software Exploration in Virtual Reality	62
6	Future Work	63
6.1	Further Empirical Studies on VR Software Visualization	63
6.2	VR Visualizations of Execution Behavior	64
6.3	Code Editing in X-Reality	64
II	Papers	87
A	Towards Immersive Software Archaeology	89
A.1	Introduction	90
A.2	State of the Art	91
A.3	Immersive Software Archaeology	92
A.3.1	Immersion: Experiencing the System	93
A.3.2	Exploration: Navigation and Orientation	95
A.3.3	Guidance: Fostering Understanding	96
A.3.4	Coalescence: Mental & Technical Back-Link	97
A.4	Evaluation	98
A.5	Conclusion and Future Work	98
B	Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in VR	101
B.1	Introduction	102
B.2	State of the Art	103
B.3	Immersive Software Archaeology	105
B.3.1	Automated Software Structure Analysis	106
B.3.2	Immersive Virtual Reality Visualization	109

B.4	Evaluation	115
B.4.1	Subject System	115
B.4.2	Software Exploration Tools	116
B.4.3	Experiment Procedure	117
B.4.4	Tasks	118
B.4.5	Participants	120
B.4.6	Findings	121
B.4.7	Threats to Validity	126
B.5	Conclusion and Future Work	127
C	ISA: Exploring Software Architecture and Design in Virtual Reality	135
C.1	Introduction	136
C.2	State of the Art	138
C.3	Immersive Software Archaeology	138
C.3.1	Automated Architecture and Design Analysis in Eclipse	139
C.3.2	Virtual Reality Exploration of Architecture and Design	140
C.4	Evaluation	144
C.5	Conclusion and Future Work	145
D	Uniquifying Architecture Visualization through Variable 3D Model Generation	149
D.1	Introduction	150
D.2	Background	151
D.2.1	Software Visualization	151
D.2.2	Variability Engineering	152
D.3	Variable 3D Model Generation	153
D.4	Outstanding Challenges	157
D.5	Conclusion & Future Work	159
E	Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality	163
E.1	Introduction	164
E.2	Related Work: Preparing Re-Engineering	165
E.2.1	Software Visualization	166
E.2.2	Reflexion Models	166
E.2.3	Freehand Sketching on Whiteboards and Paper	167
E.3	Freehand Reflexion Models in VR	168
E.3.1	Freehand Sketching of Current and Future States	169
E.3.2	Semi-Automated Interpretation of Freehand Sketches	172
E.3.3	Visual Superimposition of Code-Level References	174
E.3.4	Integration with IDE and Automated Code Generation	175
E.4	Evaluation	176

E.4.1	Study Phases	177
E.4.2	Development Phases	179
E.4.3	Participants' Answers	181
E.4.4	Answers to Research Questions	186
E.4.5	Reflections on the Evaluation	188
E.5	Conclusion and Future Work	189
F	Collaborative Software Exploration with Multimedia Note Taking in Virtual Reality	197
F.1	Introduction	198
F.2	Background and Related Work	200
F.2.1	Software Visualization	200
F.2.2	Software Documentation and Note Making	201
F.3	Collaborative Software Exploration and Note Taking in VR	201
F.3.1	Collaborative Interactive VR Visualization	201
F.3.2	Collaborative Note-Taking on VR Multi-Media Whiteboards	207
F.4	Case Study with Practitioners	211
F.4.1	Tool Implementation	211
F.4.2	Case Study Procedure	211
F.4.3	Subject System	213
F.4.4	Participants	213
F.4.5	RQ ₁ : How do engineers explore and take notes?	214
F.4.6	RQ ₂ : What strengths and weaknesses do engineers perceive?	216
F.4.7	RQ ₃ : What type of insights do engineers extract?	218
F.4.8	Discussion of Results and Lessons Learned	220
F.4.9	Reflections and Threats to Validity	222
F.5	Conclusion and Future Work	223
G	ISA: Collaborative Exploration and Note Taking in Virtual Reality	231
G.1	Introduction and Related Work	232
G.2	Collaborative Software Exploration in VR	232
G.2.1	Usage from a User's Point of view	233
G.2.2	Tool Architecture	237
G.3	Case Study with Practitioners	239
G.4	Conclusion and Future Work	240

List of Figures

4.1	Overview of the contributions presented in this thesis.	27
4.2	Overview of how Papers B and C contribute to the overall thesis. . .	33
4.3	Screenshots of our prototype implementation presented in Paper C	35
4.4	Screenshots of the semantic zoom presented in Papers B and C . .	36
4.5	Overview of how Paper D contributes to the overall thesis.	38
4.6	Example language definition for trophy landmarks	39
4.7	Overview of how Paper E contributes to the overall thesis.	42
4.8	Screenshots of taking notes on software structures in VR	43
4.9	Screenshots of notes taken on a VR whiteboard in an IDE	45
4.10	Overview of how Papers F and G contribute to the overall thesis. . .	48
4.11	Screenshots from our collaborative visualization tool (Paper G) . .	49
A.1	Overview of our envisioned method	92
B.1	Overview of our method for utilizing software architecture recovery	105
B.2	Exemplary clustering procedure next to resulting structure model .	108
B.3	Transformation of a software model to our solar system metaphor	110
B.4	Screenshots of our prototype implementation ISA	111
B.5	Overview of the experiment procedure	117
B.6	Quantitative evaluation results from different experiment phases .	121
B.7	Data structure meta model	132
B.8	Additional bar charts	133
C.1	Overview of ISA from a user's perspective	137
C.2	Screenshots of ISA's extensible Eclipse plugins	141
C.3	Screenshots of ISA's VR application	143
D.1	Example visualization of two architectural elements	150

D.2	Overview of our method for generating “uniquifying” landmarks . . .	153
D.3	Example problem and solution space artifacts	154
D.4	Example of the compositional aspect of our variant derivation . . .	156
D.5	Example of multiple visually distinct results	157
E.1	Freehand Sketching in VR	165
E.2	Overview of our method	168
E.3	Screenshots of an implementation of our method in VR	170
E.4	Meta model for the sketched diagram structure of our method . . .	172
E.5	Structure of our iterative evaluation	176
E.6	Overview of participants’ core statements during the study phases	180
F.1	Screenshot taken from a user’s point of view	199
F.2	Example folder sphere hierarchy depicting our color scheme	203
F.3	Screenshots from an implementation of our visualization method	204
F.4	Screenshots from an implementation of our note making method .	208
F.5	Screenshot of the synchronized view on notes in an IDE	210
F.6	Procedure of our case study	211
F.7	Timeline depicting participant activity in the two VR sessions . . .	215
G.1	Overview of collaborative VR software exploration and note taking	234
G.2	Architectural overview of ISA	238

List of Tables

2.1	Categorization of software visualizations	9
2.2	Overview of existing 3D software visualization techniques	16
4.1	Overview of Papers A-G and how they contribute to RQ ₁ -RQ ₄	55
B.1	State-of-the-art VR software visualization tools	116
B.2	Shortened version of the tool session tasks of our experiment	119
E.1	Shortened Version of the Overarching Tasks of the Case Study	178
F.1	Stacked bar chart with participant responses from VR sessions	217
F.2	Stacked bar chart with feedback from original developers	219

Part I

Summary

Introduction

This thesis is based on a collection of published conference papers. It is divided into two parts and structured as follows.

Part 1 – Summary

In Part 1, I summarize the contributions made in this thesis in the context of related scientific work.

- In Chapter 2, I motivate the work presented in this thesis by summarizing state-of-the-art software visualization with focus on 3D visualizations and mediums for displaying these, while pointing out gaps in the body of research.
- In Chapter 3, I state the research questions addressed in this work based on the gaps highlighted in Chapter 2 and formulate my thesis.
- In Chapter 4, I summarize the contributions made throughout the conference papers included in this thesis and how they relate to the research gaps and questions.
- In Chapter 5, I answer the research questions formulated in Chapter 3 and thus conclude my thesis.
- Lastly, in Chapter 6, I discuss future work.

Part 2 – Papers

In Part 2, I provide one chapter for each of the conference papers included in this thesis. I contain their manuscripts as printed in the respective conference proceedings.

Paper A: Adrian Hoff, Michael Nieke, and Christoph Seidl. *Towards Immersive Software Archaeology: Regaining Legacy Systems' Design Knowledge via Interactive Exploration in Virtual Reality*. Published in the Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), August 19–28, Athens, Greece

Paper B: Adrian Hoff, Lea Gerling, and Christoph Seidl. *Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality*. Published in the Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 2022), October 2–3, Limassol, Cyprus

Paper C: Adrian Hoff, Christoph Seidl, and Michele Lanza. *Immersive Software Archaeology: Exploring Software Architecture and Design in Virtual Reality*. Published in the Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024), March 12–15, Rovaniemi, Finland

Paper D: Adrian Hoff, Christoph Seidl, and Michele Lanza. *Uniquifying Architecture Visualization through Variable 3D Model Generation*. Published in the Proceedings of the 17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2023), January 25–27, 2023, Odense, Denmark

Paper E: Adrian Hoff, Christoph Seidl, Mircea Lungu, and Michele Lanza. *Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality*. Published in the Proceedings of the 39th IEEE Working Conference on International Conference on Software Maintenance and Evolution (ICSME 2023), October 1–6, Bogotá, Colombia

Paper F: Adrian Hoff, Mircea Lungu, Christoph Seidl, and Michele Lanza. *Collaborative Software Exploration with Multimedia Note Taking in Virtual Reality*. Published in the Proceedings of the 46th International Conference on Program Comprehension (ICPC 2024), April 15–16, Lisbon, Portugal

Paper G: Adrian Hoff, Mircea Lungu, Christoph Seidl, and Michele Lanza. *Immersive Software Archaeology: Collaborative Exploration and Note Taking in Virtual Reality*. Published in the Proceedings of the 46th International Conference on Program Comprehension (ICPC 2024), April 15–16, Lisbon, Portugal

Background

In the following, I summarize existing research in the field of software visualization and demonstrate gaps in its body of work. In Section 2.1, I provide a general introduction to software visualization. In Section 2.2, I discuss trade-offs between 2D and 3D visualization, motivating the need for 3D display mediums. In Section 2.3, I introduce XR hardware as a medium for displaying 3D visualizations and their trade-offs with traditional 2D screens. In Section 2.4, I briefly summarize 2D software visualization. In Section 2.5, I summarize software visualization in 3D with a focus on employed visualization metaphors and their ability to represent software architecture. In Section 2.6, I discuss the use of XR for software visualization purposes.

2.1 Software Visualization

Visualization is a powerful tool for working with complex data. It utilizes humans' visual cognitive abilities by visually presenting data, metrics on data, relationships between data points, and so on [1, 2]. This makes visualization particularly suitable for establishing an overview of otherwise overwhelming amounts of information. One successful area of application is software visualization, where source code and other artifacts model large-scale systems comprised of semantically rich structures with complex interrelations and behavior. Software engineers use such visualizations to explore and comprehend a subject system (or parts of it), while being provided with a visual overview and access to information on its structure, behavior, evolution, or combinations of these [3]. However, devising software visualization concepts suitable

for assisting engineers in their tasks is far from trivial, which gave rise to a wide variety of techniques, tools, and studies.

“Software is very difficult to visualize. Whether one diagrams control flow, variable-scope nesting, variable cross-references, dataflow, hierarchical data structures, or whatever, one feels only one dimension of the intricately interlocked software elephant. If one superimposes all the diagrams generated by the many relevant views, it is difficult to extract any global overview.”

— Frederick P. Brooks, Jr. [4]

A Categorization of Software Visualization. Software visualizations encode the formless concepts expressed in source code (such as variables, functions, classes, and relations between them) visually. This process requires what, in this thesis, I refer to as a visual metaphor [5], i.e., a mapping of intangible software concepts to visual elements such as dots, lines, icons, or even 3-dimensional buildings, islands, forests, and more. Visual metaphors for software visualization can be abstract – such as graphs with nodes and edges [6, 7, 8, 9] or hierarchical tree maps [10, 11] – or real-world inspired – such as metaphors encoding software characteristics as forests [12, 13, 14], cities [15, 16, 17, 18, 19], islands [20, 21], or planets [22, 23, 24].

Depending on the visual metaphor used, software visualizations can be distinguished into using two or three spatial dimensions, where 3D visualizations can further be subdivided by display medium into using 2D standard screens or x-reality (XR, also referred to as “extended reality”) – an umbrella term for virtual reality (VR) and augmented reality (AR) [25]. Generally, 2D visualizations tend to use abstract metaphors, while 3D visualizations lean towards using real-world inspired metaphors, most notably the information city metaphor [26, 27, 28, 29]. Orthogonal to that, visualizations are distinguished by the software characteristics they encode into structure, behavior, and evolution [3]. Table 2.1 provides an overview of this categorization of software visualizations.

2.2 Software Visualization: 2D vs. 3D

Comparisons between 2D and 3D visualizations, highlighting their respective strengths and weaknesses as well as when to use which and how have been subject to research since the late 1980s years. The results presented in that line of research are partially conflicting [30]. In the following, I elaborate on the

Table 2.1: Categorization of software visualizations by visualized software characteristics according to Diehl [3] (rows) and combinations of dimensionality and medium (columns). Table cells mark whether the type of visualization is included in this chapter’s overview of software visualizations.

Spatial Dimensions		2D		3D		
		Standard Screen		VR	XR AR	
Display and Interaction Medium		Standard Screen		VR	XR AR	
Software Characteristics	Structure	×	✓	✓	✓	
	Behavior	×	✓	✓	✓	
	Evolution	×	✓	✓	✓	

Legend	✓ core topic of thesis	✓ included in Chapter 2	× not included in Chapter 2
---------------	------------------------	-------------------------	-----------------------------

trade-offs between 2D and 3D software visualization techniques using standard 2D computer screens.

Today’s Hardware – 2D Screens. Visualizations in 2D have benefits over 3D visualizations in terms of the hardware used to display and interact with them. Today’s display mediums are two-dimensional, i.e., computer screens, phone displays, light projectors, and so on. This fits the dimensionality of 2D visualizations, whereas displaying a 3D visualization on a 2D medium requires a projection that removes one dimension, effectively throwing away depth information. Along with geometry occlusion (visual elements being hidden behind larger, occluding ones), this causes problems for viewers with understanding 3D scenes and interacting with them [6, 31].

Ease of Tool Building. From a tool builder’s perspective, constructing a 2D visualization is less challenging than constructing a 3D visualization for practical reasons. It requires less complicated algorithms, coding frameworks, and assets. In 2D, relevant visualization steps include plotting lines, positioning and overlaying 2D images, and so on. Creating a 3D visualization on the other hand requires more complex and computationally heavy tasks such as (programmatically) creating and laying out 3D elements, providing shading functions to define the visual appearance of elements, and so on. Additionally, achieving good performance in a 3D visualization on low-end hardware – and

thus high refresh rate and smooth experiences – can be challenging, especially with large quantities of visual elements to display at once [30]. While tool support for creating visualizations in both 2D and 3D exists and continues to improve, the kind of work associated with creating 3D visualizations requires more know-how and time investment.

Information Mapping. A key consideration in information visualization is representing all desired information (expressiveness) in a legible fashion (effectiveness) [32]. When visualizing semantically rich data (such as on software), it is thus advisable to categorize and rank information by importance and to use more than one visual parameter to encode relevant information [33]. Possible visual parameters are the position, scale, color, or shape of elements, among others.

Many use cases in software visualization require information to be mapped to more than two visual parameters [34] – to achieve the expressiveness desired in a visualization. However, while it is possible to encode information via non-spatial visual parameters such as shape, color, and texture, doing so effectively is far from trivial [35, 33]. For instance, shapes of visual elements become increasingly hard to distinguish when scaled small. The number of colors clearly distinguishable for viewers is limited, potentially harmed further by effects such as color blindness.

This means that spatial parameters such as position and scale along different axes are valuable for encoding information. When comparing 2D and 3D visualizations, a third spatial dimension is thus a valuable asset – an opportunity and challenge alike.

Real-World Metaphors. Semantically rich data can be represented via visual metaphors that use objects from the real world to represent data, e.g., buildings in a city or trees in a forest. Because viewers already know the adopted real-world concepts and how these are related, information can be communicated without spending much effort on understanding the encoding.

While real-world inspired metaphors can also be enacted in the form of more abstract 2D representations [36, 37], visualizations in 3D resemble the physical world more than visualizations in 2D. Because the gap between visualization and the real world is smaller, 3D visualizations can go further with adopting concepts and enhancing details. One indicator of this is the fact that in software visualization, real-world-inspired metaphors are primarily used in 3D [30].

Scene Layout and Use of Space. Positioning and scaling a large number of elements in a 2D software visualization can use up large areas and thus scale poorly with the size of a subject system [34]. Users must then choose to either inspect small excerpts of a system or get an overview of the entire visualization where details are not legible. This is a relevant problem known to users of many 2D visualizations and tools, impeding usability. Through the use of a third spatial dimension, 3D visualizations can achieve a significantly higher density of information [38] while not decreasing the space between individual elements and, thus, potentially allowing for easier access to both overview and details.

Line Occlusion. Drawing lines between visual elements is a common practice in visualizations to represent relations, e.g., for calls between functions in a software system. A relevant aspect of this is drawing the lines so that they can be understood and retraced (expressiveness). This is significantly harder to perform in 2D, where with a growing number of lines, these inevitably overlap and create visual clutter [34]. Visualizations in 3D are in favor here: they can utilize the third spatial dimension to effectively avoid overlap, e.g., using simple spline techniques [39]. However, it should be noted that displaying 3D visualizations via 2D projections (which constitute a major display technique for 3D software visualization) reduces this benefit. When inspecting a steady frame, it causes a comparable kind of incomprehensible line chaos as in 2D visualizations. One way to address this is by letting the viewer manipulate the camera view and thus benefit from motion parallax effects [40, 41].

Summary. For many information visualization purposes, using 2D representations is desirable [30]. A prime reason for this is that 2D visualizations fit the most common kinds of display mediums (computer screens, phones, etc.). For visualizations of semantically rich data, such as information on software systems, representations in 3D offer advantages over 2D due to the additional spatial dimension available for encoding information. However, due to the dimensionality mismatch when displaying them on 2D mediums, 3D visualizations have limitations regarding user interaction, resulting in a trade-off between 2D and 3D [42]. Overcoming this limitation requires mediums able to display 3D scenes in all three spatial dimensions.

2.3 X-Reality for 3D Software Visualization

In Section 2.2, I established that 3D visualizations can have benefits over 2D visualizations, especially when they represent complex and large data sets such as on software systems. However, when displaying a 3D visualization on a 2D screen, problems with user interaction and spatial understanding restrain potential benefits. In the following, I summarize x-reality (XR) technology with a focus on VR and how it can help with addressing these problems in a software visualization context.

What is XR, VR, AR, and MR? There exist no clear definitions for the terms XR, VR, AR, and MR and how to differentiate these from one another [43, 44]. In the following, I describe the terms in agreement with their most common use.

X-reality (XR, sometimes referred to as “extended reality” [44]) is an umbrella term for various forms of realities – the “X” in XR can be considered a placeholder. The common approach in XR technology today is using head-mounted devices to visualize computer-generated virtual 3D geometry, while sensors track users’ movement in the physical world to adjust and thus align their view on the virtual 3D geometry. Modern XR hardware tracks users’ movement with six degrees of freedom (6 DOF), i.e., three degrees for 3D position and three degrees for 3D rotation. Depending on the technology used, other parts of the body are tracked to let users interact with the presented 3D scene, i.e., mostly their hands via either physical controllers or hand tracking and gestures. The most prominent and relevant forms of XR today are virtual reality on one side and augmented/mixed reality on the other.

Augmented Reality (AR) and Mixed Reality (MR) visually blend the physical, real world with a virtual one [25]. Users are presented with a view of their physical real-world environment where virtual 3D objects are embedded into. While the difference between AR and MR is blurry – often, the terms are used interchangeably – one popular distinction is that MR visualizations provide more immersion into a virtual world [43], e.g., by visually altering objects from the physical world, whereas AR purely superimposes virtual objects.

Virtual Reality (VR) immerses users into an alternative, simulated reality where, in contrast to AR and MR, they are shut off from their physical surroundings [25]. Due to this difference, VR and AR/MR differ in the kind of user experiences they achieve. By immersing users into a purely virtual environment, VR applications have more control over the user experience, while reducing distractions from the physical world [45]. This makes VR a promising

medium for software visualization purposes where engineers should engage in focused exploration sessions.

XR Hardware Today. Today, the landscape of XR devices is diverse. Nevertheless, XR is a niche technology, adopted and pushed primarily by enthusiasts and far from being a commodity. For instance, in a 2023 survey, less than 2% of users on the gaming platform Steam (especially popular for PC VR gaming) stated to own a VR headset¹. In comparison with the 2D standard screen, XR hardware is far less accessible, mature, and thus practical. Although the price for XR hardware dropped significantly over the past years, it is safe to say that XR headsets require more financial investment than 2D screens – especially in the context of software visualization where XR hardware is an additional investment to 2D screens. Further, XR hardware of today struggles with practical aspects. Limitations in computational power and battery lifetime limit the scope of application for stand-alone devices, while cable-bound headsets restrict user movement. Even worse, users often suffer from motion sickness and fatigue [45, 46, 47], due to the weight of today’s XR headsets, poor display resolution in cheap models, tracking inaccuracies (e.g., in poorly lit environments), and low refresh rates (potentially worsened by limitations in computational power).

Navigation and Interaction. One accelerator for users’ spatial understanding in 3D visualizations is control over the viewing angle [48]. Users want to see and interact with objects in certain ways. Using 2D interfaces to achieve this is difficult. For instance, moving 3D objects via a 2D interface requires constraints (e.g., to movement on a plane), because information on the third dimension is not available. For the same reason, most 3D visualizations constrain user navigation, e.g., by limiting rotation or movement to one or two degrees of freedom [49, 50].

XR has the potential to provide better interaction with 3D scenes as compared to using 2D interfaces [51]. With modern hardware, users can freely change their point of view by moving their heads and interact with elements via their hands with potentially 6 degrees of freedom. This allows for powerful interaction. However, just using XR does not make a visualization more comprehensible or easier to interact with. Achieving this requires addressing aspects of navigation and interaction, while being intuitive to use, e.g., for movement over longer distances (beyond simple head movement) or user

¹<https://www.statista.com/statistics/265018/>

interactions for complex operations (such as manipulating views on source code in a software visualization).

User Performance in Software Visualization. Various studies investigated quantitative differences in user performance between using 3D visualizations in XR (particularly VR) as compared to 2D standard screens. While the results of these studies are partially contradicting, there is a trend towards XR yielding better results than the 2D standard screen for software exploration purposes.

Rüdel et al. performed a study measuring user performance (speed) in solving software comprehension tasks in a city metaphor visualization in which they compared VR with the standard 2D screen [52]. In their study, participants using 2D screens performed better than those using VR. Moreno-Lumbreras et al. performed a study in which they compared data visualizations on a 2D screen with a VR version and concluded they could not find significant differences for most use cases – although VR seems to perform better for complex tasks that involve relationships between multiple visualizations [53, 45]. On the other hand, early studies on user performance and spatial understanding in VR data visualization found that immersive mediums can achieve better results than the 2D standard screen [41, 54, 55, 56, 57]. Similarly, more recent studies on VR software visualization performed comparisons between VR and 2D screens for software comprehension tasks, concluding that VR can be faster [27] and more efficient in detecting outliers according to software metrics [58].

Conclusion. There are a number of factors influencing the outcome of studies alike to those presented above. Among the most relevant is the quality of both XR hardware and users' familiarity with these. XR hardware today is a niche technology that still combats severe issues such as user fatigue and motion sickness. Further, only a fraction of potential users are familiar with XR technology, which is a relevant factor [52]. That poses a significant obstacle to adopting XR technology in software engineering practice. However, although this puts XR at a disadvantage in comparison with the well-established, ubiquitous 2D standard screen, the results summarized above indicate potential in XR technology – even in its immature state today. With potential future innovations steadily decreasing the impact of current limitations, XR visualization studies today provide valuable insights into establishing software engineering practices of tomorrow.

2.4 Software Visualization in 2D

A majority of existing software visualizations use two spatial dimensions to map information. They represent software elements (such as classes, functions, attributes, etc.) by positioning and scaling visual elements on a flat surface, coloring and texturing them, drawing lines between elements to show relations, among others.

Generally, 2D visualizations represent software via abstract shapes. Among the most popular 2D visual metaphors in software visualization are graphs with nodes and edges representing software elements and relations between them [6, 8, 59, 60, 61]. Other visualization techniques include hierarchical edge bundling, where software elements are arranged in a circle and emphasis is put on relations between these [62, 63]. Hierarchical maps are popular for illustrating structural aspects of a subject system such as the size of its sub-components [64, 10, 11, 65]. Other approaches use notation-based metaphors, e.g., leaning on domain standards such as UML [66]. For evolutionary and behavioral aspects, matrices [67] and timelines [68, 69, 61] are popular constituents of visual metaphors. Only few real-world inspired 2D metaphors exist, such as for visualizing software structure via a topography-inspired metaphor [36, 37].

2.5 Software Visualization in 3D

With advances in computer hardware in the 1990s – most notably graphics cards becoming commodity hardware, allowing for efficient parallel processing of 3D scenes – a rapidly increasing number of new software visualization techniques were proposed that represent software systems in three spatial dimensions [70]. Table 2.2 provides an overview of that field along with the appearance dates of respective publications roughly indicating time ranges of active research. While more tools exist that are suitable for representing software, such as general-purpose graph visualizations, Table 2.2 focuses on 3D visualization methods explicitly tailored for representing software characteristics.

Table 2.2: Overview of existing 3D software visualization techniques and their key characteristics relevant to this thesis. “XR” stands for “X-Reality” and indicates whether a technique is designed for head-mounted x-reality. “Collaboration” indicates whether a technique supports multiple collaborating users at the same time. “Note-Taking” indicates whether a technique encompasses note-taking capabilities.

Name	Publications & Appearance Date	XR	Collaboration	Note-Taking	Metaphor(s) (separated by comma)
Cone Trees	1991 [71]	×	×	×	Abstract Tree
Information Cube	1993 [72]	×	×	×	Nested Cubes
Avatar	1995 [73]	✓	×	×	Matrices and Plots
FileVis	1998 [74]	×	×	×	Nested Boxes
GraphVisualizer3D	1993-2000 [54], [75], [76], [77]	×	×	×	Graphs in Hierarchical Boxes
Software World	1999-2000 [78], [79]	×	×	×	Realistic City
3DSoftVis	2000 [80]	×	×	×	Abstract Trees, 3D Timeline
Imsovision	2001 [81, 82]	✓	(✓)	×	Graph meets Abstract City
Component City	2002 [83]	×	×	×	Realistic City
Graham et al.	2004 [22]	×	×	×	Solar System
Balzer & Deussen	2004 [9], [84]	×	×	×	Graph of Hierarchical Spheres meets Abstract City
CodeCrawler	2003-2005 [85], [86], [87], [88]	×	×	×	Abstract Tree
TraceCrawler	2005-2006 [7], [89]	×	×	×	Abstract Tree
Vizz3D	2003-2007 [90], [91], [92], [93]	×	×	×	Graph, City
Balzer & Deussen	2007 [94]	×	×	×	Clustered Graph
Langelier et al.	2005-2008 [95], [96]	×	×	×	Abstract City

Name	Publications & Appearance Date	XR	Collaboration	Note-Taking	Metaphor(s) (separated by comma)
Source Viewer 3D	2003-2009 [97], [98], [99], [100], [101], [102]	×	×	×	Abstract City
EvoSpaces	2007-2009 [103], [104], [105]	×	×	×	Realistic City
CodeCity	2007-2011 [106], [15], [107], [108], [109], [16]	×	×	×	Abstract City
CodeTrees	2012 [12], [13]	×	×	×	Realistic Forest
SkyscrapAR	2012 [110]	(✓)	×	×	Abstract City
EvoStreets	2010-2013 [111], [112]	×	×	×	Abstract City
SynchroVis	2013 [113]	×	×	×	Abstract City
SeeIT 3D	2013 [114]	×	×	×	Abstract City
Würfel et al.	2015 [115]	×	×	×	Abstract City
CuboidMatrix	2016 [116]	×	×	×	Matrix of Boxes
Walls, Pillar, and Beams	2016 [117]	×	×	×	Matrix of Points
FlyThruCode	2016-2017 [23], [24]	✓	×	×	Solar System
CityVR	2017 [17]	✓	×	×	Abstract City
VR City	2017 [18]	✓	×	×	Abstract City
Schreiber and Brüggemann	2017 [118]	×	×	×	Abstract Graph
CodePark	2017 [119]	×	×	×	Abstract City
GoCity	2019 [120]	×	×	×	Abstract City
CodeHouse	2019 [121]	✓	×	×	House
PerfVis	2019 [122]	✓	×	×	Abstract City

Name	Publications & Appearance Date	XR	Collaboration	Note-Taking	Metaphor(s) (separated by comma)
GetaViz	2017-2020 [123], [124]	×	×	×	Various
Jung et al.	2020 [125]	✓	✓	×	Abstract City with Graphs
SEE	2021-2022 [126], [127], [128]	✓	✓	×	Various Abstract Cities
M3tricity	2021-2022 [129], [130], [131]	×	×	×	Abstract City
BabiaXR	2021-2023 [132], [133], [134]	✓	×	×	Various
VR-GitCity	2023 [135]	✓	×	×	Abstract City
Li et al.	2023 [14]	✓	×	×	Realistic Forest
DGT-AR	2023 [136]	✓	×	×	Abstract Graph
IslandViz	2018-2023 [137], [138], [139], [20], [21]	✓	×	×	Islands with Abstract Cities
ExplorViz	2013-2023 [140], [141], [142], [143], [144], [145], [146]	✓	✓	×	Abstract City
Immersive Software Archaeology	2021-2024 [147], [148], [149], [150], [151], [152]	✓	✓	✓	Solar System, Hierarchical Spheres containing Cylinders

2.5.1 Visual Metaphors in 3D Software Visualization

The City Metaphor. Table 2.2 shows that among existing 3D visualizations, the city metaphor is – by far – the most popular, although there are significant differences between each implementation of it. One general distinction can be made between more abstract implementations with simple cuboids

as buildings [15] on one side and implementations aiming to provide viewers with visually realistic buildings on the other (e.g., including real-world building features such as doors, windows, and so on) [90]. In comparison, abstract implementations noticeably outnumber more realistic-looking versions. A conceptually related, yet less often used visual metaphor is that of trees forming forests [13, 14].

Graph Visualizations. Another type of popular metaphor for software visualization in 3D and 2D alike is various forms of graphs, especially in tree-shaped layouts [71, 75, 87, 89]. These are of an abstract nature, consisting of primitive shapes.

Combinations of Visualization Metaphors. Existing visual metaphors in 3D visualization are often combined to form new metaphors [153]. A popular combination is graphs and cities – in the sense of augmenting primarily city metaphoric visualizations with graph structures spanning from building to building [104] or vice versa, i.e., graph based visualizations with city-like structures as their nodes [9, 81, 140].

2.5.2 Software Architecture Visualizations in 3D

When trying to comprehend a subject system from its source code alone, it is challenging to gain an overview of its structure. That is, for one, due to the vast amount of information to analyze and comprehend in any non-trivial software system. For another, it is because even with programming languages considered “high-level” (such as Java or C#), source code explicitly models information on a relatively low abstraction level when considering the structure of an entire system.

Various 3D software visualization techniques were proposed for visualizing software architecture [5, 35, 154, 155, 156, 157]. However, these consider a system’s folder organization as a high-level software architecture structure, often visualizing folders via secondary visual elements (e.g., as districts in a city visualization).

What is Visualized as Software Architecture Structure?

Existing 3D software visualizations use simple-to-extract information as a system’s architectural structure. Mostly, they assume file system folder hierarchies as an adequate model of an architectural structure and visualize these,

e.g., as districts in a city or higher-level nodes in a tree. A notable exception is the IslandViz tool [138] which displays software architecture based on a system's organization into plugins. However, to achieve that, IslandViz assumes explicitly modeled architecture via an OSGi plugin system specification – a strong assumption that holds only for a fraction of systems.

In long-running systems, exposed to effects of architectural drift and erosion over years of evolution [158, 159], folder structure can deviate heavily from an organization into cohesive collections of files, classes, or similar source code containers. Thus, reliable information on a system's architectural structure is only implicitly available in source code, e.g., in the form of complex relationships between multiple classes in an object-oriented system.

A large variety of software clustering techniques exist that (semi-)automatically estimate high-level system structure beyond file system information [160, 161, 162, 163]. Among these are techniques for determining a hierarchical organization in cohesive collections of files, classes, etc. – a potentially valuable source of information for providing engineers with an overview of a system's architecture. However, to the best of my knowledge, combining automated software clustering techniques with 3D visualization was not studied in the past.

Research Gap 1

There exist no studies on combining results from automated software clustering techniques with 3D software visualization.

How is Software Architecture Structure Visualized?

Most 3D visualizations represent architectural information via secondary visual elements. For instance, a popular way to visualize folder structure in the city metaphor is by distributing buildings for classes or files into city districts and coloring the ground respectively, e.g., in different shades of gray. Although this allows for inspection of higher-level software structure, its encoding is subtle and subordinate to lower-level information such as metrics driving the scale of buildings. Only few techniques exist that represent higher-level software structure as primary visual elements, e.g., folders as hierarchically nested spheres [9]. In the realm of XR software visualization, the body of existing work on this is even thinner. Most notably, ExplorViz uses nested boxes containing city-like structures to show software execution behavior on a system level and IslandViz visualizes OSGi plugins as islands in a virtual ocean [138]. These techniques are valuable for their respective intended use

cases. However, further research into interaction and visualization metaphors is required, e.g., to study how to guide engineers into more detailed inspections of system structure on demand – both IslandViz and ExplorViz lock users on a fixed abstraction level.

Research Gap 2

There is a lack of studies on the impact of using 3D software visualizations for presenting software architecture as their primary first-level visual structure.

Repetitive Visual Patterns in Software Visualization

Software visualizations use procedural techniques for generating visual elements, following intentionally simple rules for determining their shape. Thereby, they encode information on a software element (e.g., a function) in the visual element representing it. For instance, to provide engineers with an overview of the structural size of software elements, visualization elements could be scaled based on a respective software metric, e.g., buildings in a city metaphor visualization scaled according to the number of lines of code of the represented file. This applies to visualizations in 2D and 3D alike. However, while useful for communicating information visually, the approach causes repetitive patterns, especially for large-scale subject systems (which constitute a prime use case for 3D visualization, cf. Section 2.2). Thus, visualizations of large-scale systems tend to result in large-scale landscapes of similarly shaped visual elements. This makes it hard for viewers to distinguish between representations for different parts in a subject system, impeding their orientation and ability to relate information (e.g., “Is this what I have seen before?”).

Related work shows that visually distinct landmarks (i.e., noticeable and recognizable structures) help viewers with their orientation in information visualizations [164, 165, 166]. However, existing software visualizations do not exploit this phenomenon. Doing so would require a method for generating potentially large quantities of visually distinct landmarks and placing them in suitable locations throughout the visualization, thus “uniquifying” different parts of a visualized system. The challenge in this is related to the problem itself, i.e., potential methods for generating landmarks are prone to producing repetitive outcomes – after all, they are procedures for generating visual elements, too. Further, it is not trivial to achieve control over a 3D geometry generation process such that it reliably produces sufficiently unique outcomes. How can developers of visualizations control what “sufficiently

unique” means for their purposes? To the best of my knowledge, there exist no techniques for reliably generating visually distinct landmarks in quantities large enough to “uniquify” potentially large-scale visualizations (such as of a large-scale software system).

Research Gap 3

There exist no methods in 3D software visualization to counteract repetitive visual patterns impeding viewers’ ability to recognize different parts and, thus, orient in a subject system.

2.6 Software Visualization in XR

Table 2.2 shows that a majority of 3D software visualizations, especially work before circa 2016, use standard 2D computer screens as output medium (see column “XR”). Noticeable exceptions to this are Avatar [73] (ca. 1995) and Imsovision [81] (ca. 2001), virtual reality software visualization concepts that use image projection on walls in a room-scale setup while tracking users’ head and hands (CAVE [167]).

The main starting point for research on XR software visualization is 2016, around the same time as big technology companies started launching XR headsets at significantly lower prices than those of existing enthusiast-oriented models in an attempt to make these commodity hardware [168]. Since then, a variety of different studies on XR software visualization have been published with a majority of these being about VR technology. However, while these grant valuable insights into XR and VR as a medium for software visualization, there is a lack of studies on pivotal aspects that I highlight in the following.

2.6.1 Note Taking in Long-Lived XR Software Exploration Sessions

A pivotal aspect of exploring and comprehending an unfamiliar software system is taking notes on findings as, otherwise, engineers risk losing valuable insights – especially in long-lived exploration sessions. This is true for all software exploration means alike. However, existing XR software visualization tools do not support engineers in taking notes of any kind (see column “Note Taking” in Table 2.2). With existing techniques, the only way for engineers to take notes during XR exploration sessions is by falling back to external note-taking means, either via third-party applications installed in their XR device

or via traditional note-taking means such as pen and paper or whiteboards. For VR visualizations, this is especially harmful because it requires engineers to disrupt their immersion and exploration.

Research Gap 4

There exist no studies on taking notes during long-lived software exploration sessions in VR visualizations, posing the risk of losing valuable findings.

2.6.2 Collaborative Software Exploration via XR Visualization

Software exploration is an activity often conducted by teams of collaborating engineers, e.g., when preparing for the re-engineering of a software system. However, because commonly used tools such as IDEs are designed for single-user scenarios, they do not adequately support software exploration in a collaborative setting. That is, they lack the means for multi-user interaction and, even when somehow operated by more than one user, impede engineers in synchronizing their actions and views on the source code.

Collaborative visualization concepts for 2D exist. Notable examples are Churrasco [169, 170], a web-browser-based visualization for analyzing software evolution, and SourceViz [171], a visualization for co-located exploration sessions via multi-touch displays.

Compared to 2D mediums, XR provides the potential for a unique form of collaborative software exploration where engineers simultaneously investigate synchronized visual elements from different angles. At the same time, XR techniques (especially VR) can achieve a strong feeling for the presence of other users, even when they are physically separated, by visualizing avatars for them and updating their movement in real time. Thus, there is a striking correlation in 3D software visualization between support for real-time collaboration and XR (see column “Collaboration” in Table 2.2); if 3D visualizations support collaborative exploration, they are designed for use in virtual reality.

However, all in all, there exist only a few visualization techniques for collaborative software exploration in XR. The Imsovision project [81], early work on utilizing VR technology for software exploration purposes, includes plans for synchronizing and rendering collaborators in VR. However, it is unclear whether these were eventually implemented. Both Jung et al. [125] and the ExplorViz tool [144, 146] present collaborative VR software exploration techniques for investigating the execution behavior of a system. SEE [126] is a

project on collaborative VR software exploration with particular focus on detecting software clones and architectural drift. While these contributions provide valuable insights, many facets of collaborative software visualization remain open, e.g., engineers' exploration behavior or note-taking during long-lived collaborative sessions.

Research Gap 5

There is a lack of studies on using VR visualization methods for collaborative software exploration, leaving open relevant aspects such as engineers' exploration behavior.

Problem Definition

In Chapter 2, I outlined existing research on software visualization with a focus on 3D metaphors and XR as a medium. Further, I highlighted research gaps in the body of existing work:

- Research Gap 1:** There exist no studies on combining results from automated software clustering techniques with 3D software visualization. (Section 2.5.2)
- Research Gap 2:** There is a lack of studies on the impact of using 3D software visualizations for presenting software architecture as their primary first-level visual structure. (Section 2.5.2)
- Research Gap 3:** There exist no methods in 3D software visualization to counteract repetitive visual patterns impeding viewers' ability to recognize different parts and, thus, orient in a subject system. (Section 2.5.2)
- Research Gap 4:** There exist no studies on taking notes during long-lived software exploration sessions in VR visualizations, posing the risk of losing valuable findings. (Section 2.6.1)
- Research Gap 5:** There is a lack of studies on using VR visualization methods for collaborative software exploration, leaving open relevant aspects such as engineers' exploration behavior. (Section 2.6.2)

3.1 Research Questions

With my work, I address one central research question:

How can VR software visualization assist engineers in exploring the architectural structures of software systems in prolonged sessions with peers?

To answer this question, I define a series of subordinate research questions, providing insights into solutions for the gaps highlighted in Chapter 2 (and listed above).

- RQ₁**: How can software be visualized in VR so that it fosters engineers' exploration of architectural structure? (Research Gaps 1, 2, and 3)
- RQ₂**: What are strengths and weaknesses of using a VR software visualization for exploring architectural structure? (Research Gaps 1 and 2)
- RQ₃**: How can engineers be supported in taking notes during software exploration sessions in a VR visualization without disrupting their immersive exploration? (Research Gap 4)
- RQ₄**: How do teams of software engineers explore systems in a collaborative multi-user VR visualization? (Research Gap 5)

3.2 Thesis

In an attempt to answer the research questions formulated in Section 3.1, I formulate the following thesis:

VR visualization is suitable for exploring unfamiliar software systems because it provides engineers with an overview of system architecture, an environment fostering collaborative work with peers, and powerful interaction mechanisms, e.g., for querying detailed information or for taking notes on findings.

In Chapter 4, I elaborate on the contributions made in Papers A-G and how these answer RQ₁-RQ₅. I summarize these results in Chapter 5, providing agglomerated answers to each research question, and concluding on how these support my thesis.

Solution Overview

Papers A-G present concepts, tools, and empirical studies on utilizing VR to explore software structure through immersive visualizations with multi-media note-taking capabilities and support for synchronous collaborative usage with peers. Figure 4.1 provides a high-level overview of these contributions and how they relate. Engineers explore visualizations of a system's architecture and detailed structure in VR while profiting from a system-level overview of architectural structure. In parallel to that, they take notes on their findings directly in VR, assisted with conformance checks and support for multi-media recordings. Distributed teams of engineers can collaborate in these activities through real-time synchronization.

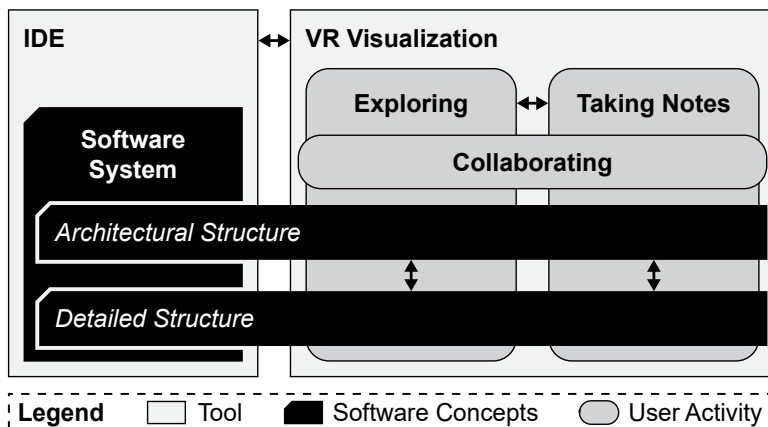


Figure 4.1: Overview of the contributions presented in this thesis.

In the following, I summarize the contributions made throughout Papers A-G, thus expanding on the concepts and activities illustrated in Figure 4.1. I briefly touch upon the applied research methodology (Section 4.1) before summarizing each paper (Section 4.2) and finally providing an overview of their contributions in the form of a concise table (Section 4.3).

4.1 Methodology

In the following, I briefly summarize the research methods applied throughout Papers A-G as referred to in their summary in Section 4.2.

Tool Prototyping. I study concepts designed to help engineers understand software systems. To evaluate these concepts, I implement them in tools and assess their strengths and weaknesses through empirical studies with software engineers. These tool implementations, which encompass all concepts presented in Papers A-G, are available in an open-source repository¹ and serve as proof-of-concept demonstrations.

Controlled Experiments. A well-established and thorough form of empirical assessment with human participants is controlled experiments. These investigate research questions in a well-defined (controlled) setting where the researcher stipulates all external influences and variables except those under investigation. Thereby, the researcher can conclude that differences in participants' behavior and answers are in a causal relationship with the unconstrained, isolated variables under investigation, yielding strong empirical evidence [172]. Paper B presents a controlled experiment.

Case Studies. Another type of empirical assessment with humans is case studies. Based on a set of predefined research questions, these investigate phenomena in a realistic setting [173], yielding large quantities of data and anecdotal evidence even with smaller groups of participants than feasible for controlled experiments. This is valuable in empirical software engineering. In comparison to controlled experiments, case studies provide results that are generally more open, which can generate a deeper understanding of a studied subject [174]. Papers E and F present different case studies.

¹<https://gitlab.com/immersive-software-archaeology/>

“Knowledge is more than statistical significance.”
— Runeson and Höst [174]

The case study presented in Paper F demonstrates the open nature of the approach. Following an incremental and user-centered approach, I collected primarily qualitative feedback from practitioners, analyzed the results, improved my concepts and prototype, and continued with the next iteration.

4.2 Contributions per Paper

In the following, I summarize Papers A-G. For each paper, I provide a summary of contributions along with a brief statement on how these relate to the research questions defined in Chapter 3.

4.2.1 Towards Immersive Software Archaeology: Regaining Legacy Systems’ Design Knowledge via Interactive Exploration in Virtual Reality (Paper A)

Paper A is a conceptual contribution presenting a vision for a VR software visualization method addressing Research Gap 2. It presents concepts for assisting engineers in exploring (legacy) software systems they are not familiar with via interactive visualizations in immersive virtual reality. I presented Paper A at a general software engineering conference to engage with a broad expert audience and to gather feedback. With Papers B-G, I follow up on the concepts presented in Paper A.

Results

In Paper A, I envision a method for visualizing large-scale software systems in virtual reality. The envisioned method aims to provide engineers with a higher-level architecture overview of a system while using visualizations of execution behavior and code quality metrics to guide engineers to interesting parts of a system where they can query details on fine-grain information on demand, e.g., textual views on source code for a particular function.

Immersion through Real-World Metaphor. I envision a metaphor that maps software concepts (functions, classes, etc.) to 3D visual elements commonly known from the real world, i.e., planets, cities, buildings, and their inhabitants. Thereby, the metaphor builds on engineers’ familiarity with the

associated concepts and relationships between these, e.g., a planet contains cities, inhabitants live in buildings, and so on. With this metaphor, I propose to visualize the structure, behavior, and quality of subject software systems.

Visualizing Software Structure. To visualize a system's structural properties, the metaphor represents coarse-grain components (e.g., root-level source folders) as planets. Sub-components of a system are represented as cities on a planet, where buildings represent elements on the level of files, classes, structs, or similar concepts. I propose to encode function-level structural information on each function/method by representing these as the floors of a building, scaled according to software metrics capturing their structural size.

Visualizing Software Behavior. Visualizing the execution behavior of a system helps identify heavily frequented, and thus critical, sections in its source code. In a software re-engineering effort, these might require special attention from engineers. I propose to visualize execution behavior as an integral part of the solar system metaphor visualization summarized above. I envision previously recorded execution traces to be represented as inhabitants of the solar system, wandering from building to building, through cities, and across planets. Thereby, inhabitants represent specific calls through the system, carrying information that alters the state of the subject system. Users could inspect these aspects to gain an understanding of the inner workings of the system.

Visualizing Software Quality. Visualizing software quality is useful for pointing engineers to re-engineering opportunities. To that end, I propose to encode established metrics from existing software quality assessment tools (e.g., SonarQube²) into the visual representation of software elements. For instance, the facade of a building (representing a class/file level element, cf. above) could receive a brittle texture to communicate poor code quality based on a respective software metric. Effects such as litter spread throughout a city could indicate a module-level smell.

Guidance. With the concepts summarized above, my envisioned method aims to guide engineers through a visualization of a subject system and, thus, foster their exploration. Observable phenomena subtly draw their attention to points of interest, e.g., a particularly large building that represents an overly large class/file, a brittle facade or litter in the streets of a city indicating poor

²<https://www.sonarsource.com/>

code quality, or a gathering of inhabitants in certain locations in the city indicating a performance bottleneck.

Overview and Details. With the envisioned method, I aim to assist engineers in exploring software systems with a focus on architectural elements. I envision a method fostering a top-down exploration style that focuses on a high-level overview first and then provides engineers with access to details of interest on demand (Schneiderman's mantra [175]). I plan for respective concepts on the aspects of software structure, behavior, and quality.

Regarding software structure, I envision a semantic zoom that enables users to expand upon more detailed information, e.g., on the members contained in a Java class. In the metaphor summarized above, engineers could zoom in from a system level (where they see planets) to a city level where they see buildings larger in scale and detail, presenting further semantic information. For instance, in a visualization of Java code, individual methods of a class could be visualized as building floors indicating through geometric details whether they are encapsulated, abstract, etc.

Regarding software behavior, I envision a mechanism enabling engineers to manipulate execution time when replaying recorded traces through a system, i.e., pausing, rewinding, speeding up/down, etc. This is useful for carefully examining both short-lived and prolonged processes.

Regarding software quality, I envision access to explicit raw metric information, such as yielded directly from the underlying analysis tool. That is, besides encoding the results in the form of phenomena such as building facade textures or litter in a city, I envision a user interface with ground-truth information.

Interaction. For cases where textual information is required, I envision diegetic user interfaces, i.e., interactable elements residing in the virtual world of the VR visualization – as integral parts of it. For example, engineers could inspect source code or ground-truth information on software quality metrics via a tablet screen attached to their virtual arm that they can dock to elements in the visualization and thus query information. While being VR compatible, this integrates with the metaphor and maintains engineers' immersion.

Mental & Technical Back-Link. The goal of my envisioned VR visualization method is to assist engineers in exploring a subject system. A pivotal part of that is establishing a link between the mental model built in the visualization and the system's ground-truth source code. To that end, I plan to connect the

proposed visualization with an IDE. Based on that, I envision mechanisms that enable engineers to mark interesting parts of a subject system and to create annotations on them. For instance, I propose to address this via sound recordings attachable to visual elements that are shared between the VR world and IDE.

Collaboration. Software exploration is an activity often conducted by teams of engineers. These either collaboratively explore a system simultaneously or they each explore at different times. In Paper A, I propose concepts for supporting engineers in the latter use case. I envision a record and replay mechanism on VR exploration sessions, that stores a guided tour through a system. Such tours are (asynchronously) replayed by collaborators, e.g., to be introduced into a system they do not know yet, or receive a status update on specific new parts or aspects of a system.

Addressed Research Questions

With the contributions summarized above, Paper A envisions concepts for answering RQ₁. Based on these visions, I presented more detailed concepts, prototype implementations, and empirical studies in Papers B-G.

The remainder of Section 4.2 summarizes the results of pursuing the vision laid out in Paper A. However, these deviate slightly from the vision presented in Paper A. That is, over the course of studying software structure visualizations, I noticed striking research gaps in the body of existing work (summarized in Chapter 2). For that reason, I pushed back plans on supporting engineers in exploring system behavior and quality to future work, venturing deeper into aspects of exploring software structure. For a more detailed discussion on future work, refer to Chapter 6.

4.2.2 Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality (Papers B and C)

Papers B and C address Research Gap 1 and Research Gap 2 (Section 2.5.2) via a method for (a) automatically analyzing a software system's structure and (b) presenting engineers the results in an interactive VR software visualization. Figure 4.2 outlines this contribution in the scope of this thesis.

While Paper B focuses on concepts developed for the presented method, Paper C reports on the technical aspects of a tool implementation. Further,

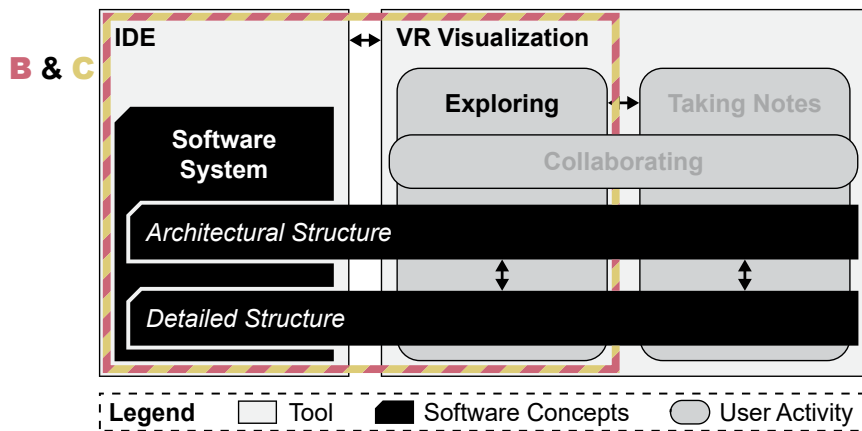


Figure 4.2: Overview of how Papers B and C contribute to the overall thesis.

Paper B reports on a controlled experiment comparing the presented VR visualization method with an existing state-of-the-art VR software visualization and, as a baseline, with the open-source Eclipse IDE.

Results

In the following, I summarize the results presented in Papers B and C.

Fully Automated Software System Analysis. The analysis presented in Paper B is split into two stages, i.e., (i) a parsing phase establishing a model on the level of classes/files and their content and (ii) a clustering technique to estimate an organization of the subject system's class/file level elements into a hierarchy of cohesive clusters. Because I chose Java as the target language for the tool implementation presented in Paper C, I use object oriented-concepts such as classes, methods, etc. in the following summary. Nevertheless, the summarized method is generally applicable to structured programming languages with modularization concepts.

Parsing Stage. In its first stage, the analysis parses the system's source code to collect explicitly modeled information on the structure of classes. Thereby, it establishes a model of the subject system's classes and inheritance relations as well as information on their members (return type and signature of methods as well as name and type of fields). Further, the analysis summarizes each method by calculating two metrics, i.e., one for complexity based on control flow splits in the method's body and one for its number of expressions. The model resulting from this first analysis stage captures all references between

classes and members in the system (e.g., a constructor accessing a field or calling a method, a class implementing an interface, or a method referencing a type).

Automated Clustering Stage. In its second stage, the analysis conducts a fully automated clustering procedure. It utilizes the model constructed in the above-mentioned first stage to automatically determine an organization of the subject system's classes into a hierarchy of cohesive clusters, i.e., collections of classes that are strongly interconnected among one another, but weakly connected to other clusters. This process is based on references between source code elements as captured in the first analysis stage, i.e., type references, method/constructor calls, and field accesses, thus maximizing the cohesiveness of clusters. Thereby, the software analysis presented in Paper B is able to estimate an architectural organization of a subject system's source code, independently from potentially flawed and/or outdated folder hierarchies.

The presented procedure utilizes existing software clustering techniques in a novel way, providing users with control over the minimum and maximum size of clusters in the resulting hierarchy. Further, users can influence the clustering outcome by blending between different similarity measures used to determine the cohesiveness between classes and by defining minimum and maximum cluster sizes. The provided options are useful for visualization purposes where, for instance, constraining cluster sizes to certain boundaries is useful for layouting purposes.

Interactive VR Visualization. Papers B and C present a visualization concept and tool that visualizes the results from the above-summarized analysis in virtual reality. Engineers navigate the VR visualization and interact with its 3D elements to explore a subject system visually, supported by an architectural overview, a semantic zoom gradually revealing lower-level information, relationship graphs summarizing references in the system's source code, and – when desired – ground-truth textual representations of software elements.

Visual Metaphor with Architectural Overview. To visualize a subject system's structure, the method presented in Papers B and C represents intangible programming language concepts via a solar system metaphor. Figure 4.3 provides an overview of this metaphor. On an architectural level @-©, cohesive clusters of classes resulting from the above-summarized analysis are represented as planets (top-level clusters) containing hierarchies of continents and

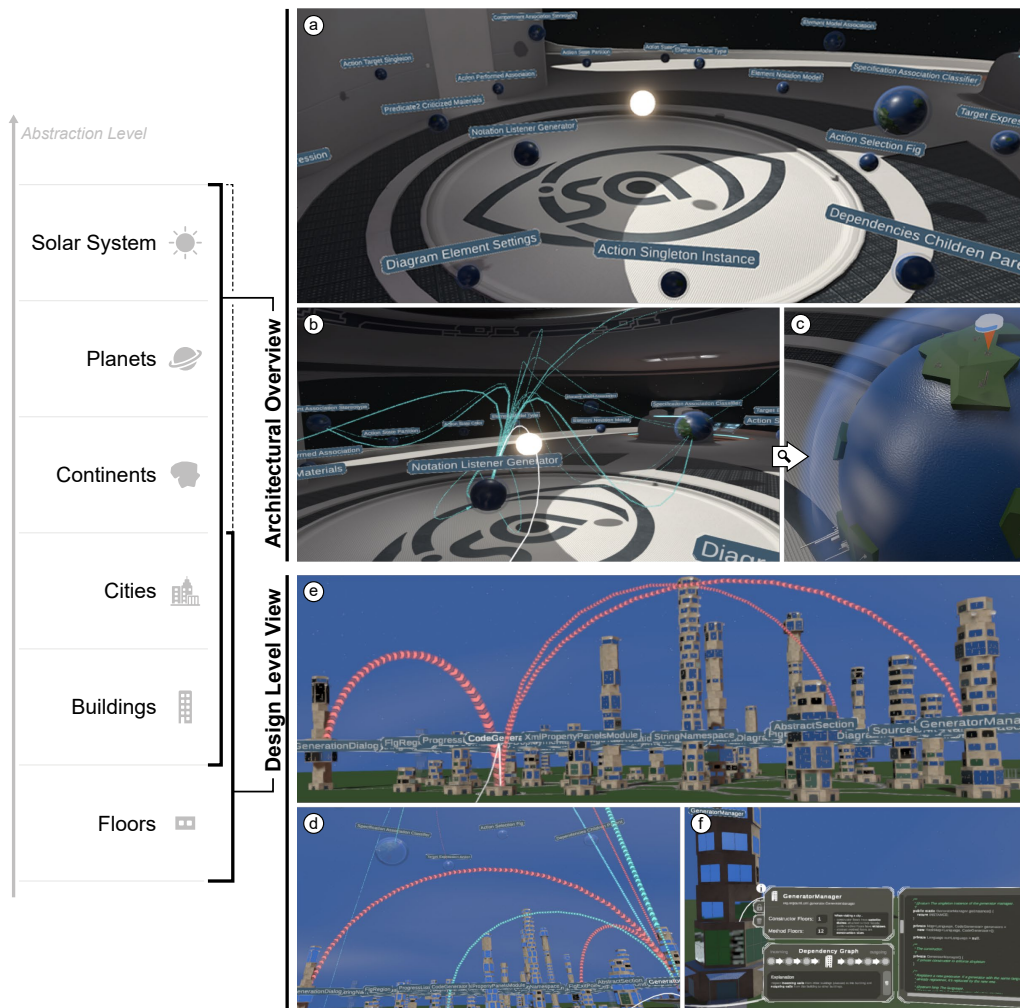


Figure 4.3: Screenshots of the prototype implementation presented in Paper C showing an example system with ~ 1.800 Java classes on different levels of abstraction. The upper area shows architecture level while the bottom area shows class and method level.

sub-continent (intermediate-level clusters). Bottom-level clusters, i.e., those containing classes directly, are visualized as cities (d)-(f) of buildings which represent classes, interfaces, enums, records, etc. Thereby, the visualization provides an explicit overview of a system’s architectural structure. Engineers explore a subject system by navigating through its solar system representation in VR and interacting with visual elements as described in the following.

Semantic Zoom. To emphasize architectural structure and foster a top-down exploration, the visualization locates engineers on the level of planets when

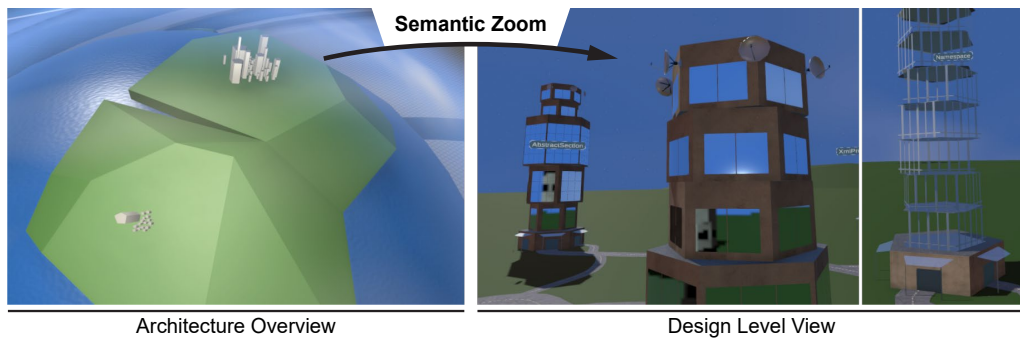


Figure 4.4: Example screenshots of the semantic zoom presented in Papers B and C. Engineers can switch the level of abstraction from architecture level to class/file level and vice versa. When zooming in, buildings are not only larger in scale, they are also enhanced with semantics by being composed of individual floors with further geometric features (e.g., windows that indicate a method is publicly accessible).

starting their exploration (see ①-③ in Figure 4.3). There, engineers can inspect planets, their continents and sub-continent hierarchies, and small-scaled versions of each city located on the surfaces of planets. The left-hand side of Figure 4.4 provides an optically zoomed-in view of a city located on a planet's surface in the architectural overview.

Engineers can semantically zoom in on a bottom-level component by visiting the city representing it (right-hand side of Figure 4.4). Thereby, buildings increase noticeably in scale, while being semantically enriched with more information. That is, buildings are composed of individual floors for each of the represented classes' methods. To provide a visual summary of methods and thus classes, the shape of each floor is determined by metrics measuring method complexity (floor radius) and the number of contained expressions (floor height).

Relationship Graphs. To provide engineers with an overview of the relationships between visualized software elements, the method presented in Papers B and C visualizes references in the source code as animated arced lines between the corresponding visual elements, forming a relationship graph. References from one software element (e.g., a class or entire cluster) to another are visualized as an arced line between the two respective visual elements (e.g., building, city, or entire planet). Figure 4.3 ②, ④, and ⑤ show examples of this. Engineers interactively blend relationships between elements in and out by interacting with a graphical user interface in VR (Figure 4.3 ⑥ illustrates the user interface for buildings in a city).

Source Code as Text. With the above concepts, the method presented in Papers B and C provides engineers with an interactive visual overview of a subject system's structure, including explicitly an architectural view. Nevertheless, engineers can access the ground-truth information on that visual overview, i.e., the system's source code, via a user interface in VR (see Figure 4.3 (f)).

Controlled Experiment. In Paper B, I report on a controlled experiment with 54 participants (students and scientific staff of the IT University of Copenhagen) assessing the quality of the above-summarized concepts in a comparison with a state-of-the-art VR software visualization and a standard IDE. Participants in the experiment used one of the three methods to solve tasks while talking out loud and thus providing feedback. I observed how they approached each task and what results they produced to draw conclusions on the strengths and weaknesses of the three methods in assisting participants with relevant software exploration tasks. That is, I investigated in what sense the methods assisted participants in accessing and relating information on elements in the subject software system.

Accessing Information A key aspect of software exploration is accessing information such as identifying a set of classes implementing certain functionality. The experiment shows that, for the provided tasks, visualizing high-level software structure based on results from clustering techniques helped participants in accessing information, as it groups together software elements based on their interrelations, e.g., as buildings in the same city or continents on the same planet. As a result, engineers gain easier access to correlated information in comparison to the other methods under test which use the system's folder structure to organize classes.

Relating Information Relating information is a crucial task in software exploration. I distinguish it into relating software elements horizontally (same abstraction level), such as identifying what methods are called from an inspected method, and relating software elements vertically (different abstraction levels) as in identifying containment relations, e.g., what classes should be contained in a software module. To perform these tasks, participants in the IDE need to read through source code. While this provides more experienced developers with deeper insights into implementation details, IDE participants generally perform worse in relating their acquired pieces of knowledge than participants using the visualizations. This is especially notable for

less experienced developers. The experiment shows that an explicit visualization of references between software elements, such as provided by the relationship graphs in the presented method, helps identify horizontal relations, especially on an architectural level. For relating software elements vertically, the experiment shows that both visualizations under test (i.e., our method and the state-of-the-art visualization) provided participants with an overview of the system’s structure resulting in a better understanding of the dimensions of architectural structures than those from participants working in the IDE on a textual representation of source code.

Addressed Research Questions

With the results summarized above, Papers B and C contribute to RQ₁ and RQ₂. The papers present concepts and a tool implementation for VR software visualization with an emphasis on software architecture structure (RQ₁). Further, Paper B presents a controlled experiment comparing participants’ ability to perform software comprehension tasks in VR software visualizations compared to a traditional IDE (RQ₂).

4.2.3 Uniquifying Architecture Visualization through Variable 3D Model Generation (Paper D)

In Paper D, I address Research Gap 3 (Section 2.5.2) via concepts for generating visually distinct 3D landmarks which, when integrated into a software visualization, can be used to “uniquify” otherwise similar-looking parts. The contribution made is primarily conceptual and technical, aimed at providing

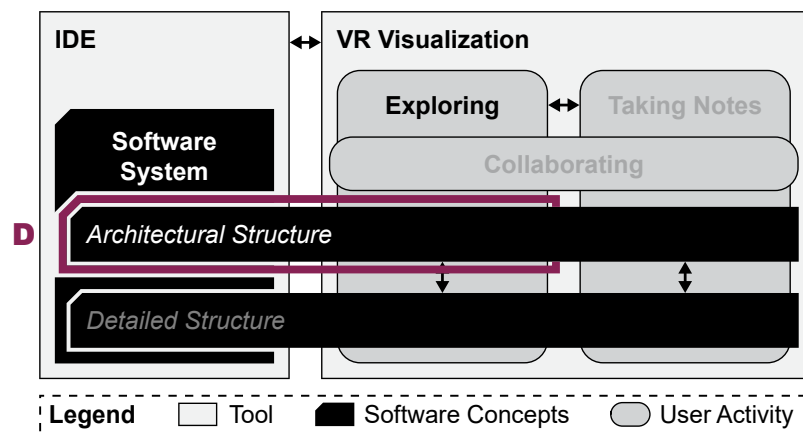


Figure 4.5: Overview of how Paper D contributes to the overall thesis.

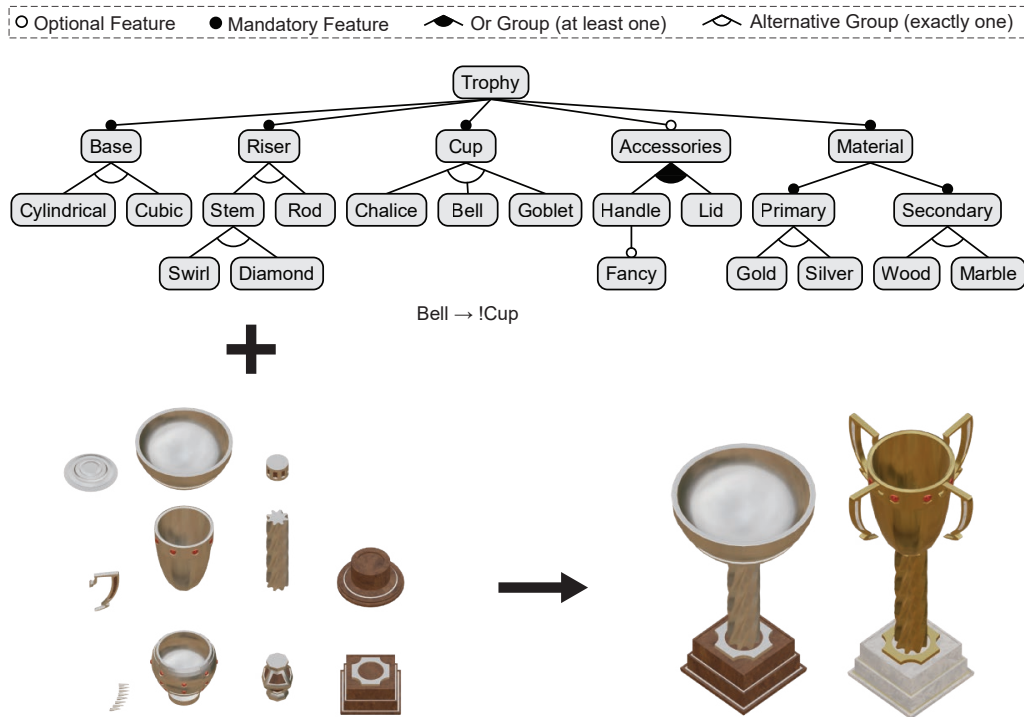


Figure 4.6: Example language definition for landmarks resembling sports trophies. A feature model captures the configuration rules (problem space), partial 3D geometry defines pieces to be assembled (solution space). A hybrid variant derivation procedure generates final landmarks by (1) composing partial 3D geometry and (2) transforming visual characteristics such as material.

developers of 3D software visualizations with a solution to a non-trivial problem. Figure 4.5 highlights this contribution in the context of this thesis.

Results

The method proposed in Paper D is centered around defining landmarks as a visual design language using concepts from variability engineering. Developers of software visualizations model a design language expressing landmarks via two kinds of artifacts:

1. A feature model (top part of Figure 4.6), i.e., a tree-shaped model where nodes are selectable configuration options. Feature models encode semantics on the relationship between features, e.g., two features being mutually exclusive, thus providing fine-grain control over the space of valid configuration options of the design language (problem space).

2. A set of partial 3D geometry, where each piece corresponds to a feature from the feature model (bottom left part of Figure 4.6). This partial geometry expresses the space of composable elements (solution space).

A concrete landmark is generated by first deriving a valid configuration of the design language (problem space), i.e., a valid selection of features, and then assembling the corresponding geometry pieces (solution space).

From Software To Valid Landmark Configuration (Problem Space). I propose a method for deriving valid configurations of a design language (defined as summarized above) based on a collection of source code elements – whose representation in a visualization should be unquified. First, the method generates a software descriptor, deterministically encoding simple high-level characteristics via the number of contained elements and a hash over its element names. For instance, this method can be used on a folder to generate a descriptor that is robust against minor changes in the contents of constituent files while sensitive to higher-level changes such as the deletion of a file. Second, the method uses deterministic steered random sampling on the design language’s feature model, driven by the previously calculated software descriptor. The result is a valid configuration, i.e., a valid selection of features in the feature model (a “word” of the design language) that is mapped to a selection of partial geometry.

From Landmark Configuration to 3D Geometry (Solution Space). As a last step, the method presented in Paper D includes a hybrid variant derivation mechanism that takes a valid configuration of the design language as input to generate the landmark geometry described by it. I refer to the mechanism as “hybrid” because it subsequently employs two variant derivation techniques. First, the presented method composes the pieces of partial geometry included in the given configuration. This composition uses a hooks-and-slots mechanism to determine how pieces are matched. That is, each piece of geometry is annotated with one root hook and arbitrarily many slots. The latter mark the position and rotation for child elements to be hooked into (where to place a child piece’s root hook). Second, the presented method transforms the visual characteristics of the resulting geometry. For now, this transformation can be used to change the material of geometry pieces based on a simple naming scheme.

Tool Implementation. I implemented the concepts presented in Paper D as an open-source tool³ using the Unity 3D engine⁴ (a popular tool among developers of 3D visualizations) and FeatureIDE (“an extensible framework for feature-oriented software development”⁵). The tool can be integrated into existing visualizations – either directly when made with Unity, or in adapted form when using other technology.

Addressed Research Questions

Maintaining viewers’ orientation in a visualization is crucial as, otherwise, they are hampered in establishing a coherent mental model of the subject system. With the work summarized above, Paper D contributes to achieving this by augmenting visualizations with distinct, remarkable landmarks. Thereby, the paper contributes to RQ₁.

4.2.4 Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality (Paper E)

Paper E addresses Research Gap 4 (Section 2.6.1) via concepts and a tool implementation for supporting engineers in taking notes about software structures via freehand sketching in VR with automatic interpretation, integration with ground-truth source code, and conformance checks. Further, the paper presents an iterative case study through which I assessed the strengths and weaknesses of the presented VR note-taking method while improving it over the course of each iteration. Figure 4.7 outlines this contribution in the scope of this thesis.

Results

In the following, I summarize the results presented in Paper E.

Freehand Sketching with Automated Interpretation and Conformance Checks in Virtual Reality. The note-taking method presented in Paper E extends VR software visualizations with a virtual whiteboard on which engineers

³<https://gitlab.com/immersive-software-archaeology/variable-3d-landmarks/>

⁴<https://unity.com/>

⁵<https://featureide.github.io/>

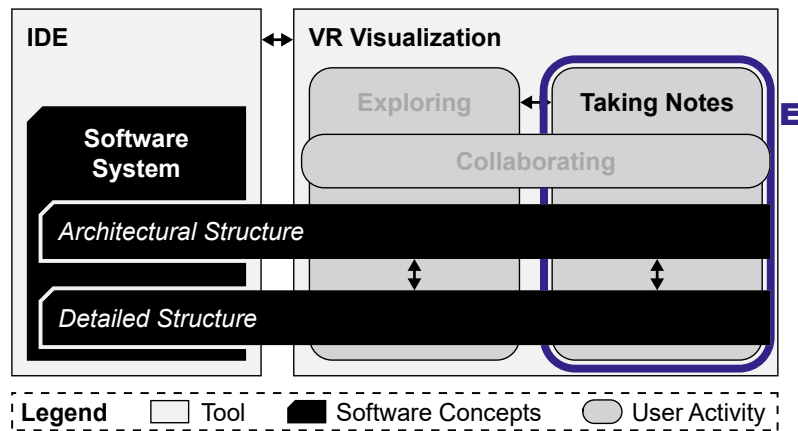


Figure 4.7: Overview of how Paper E contributes to the overall thesis.

can pin software elements from the visualization and freely hand-draw arbitrary sketches. The method aims primarily at taking notes on architecture-level software structures, e.g., on how a collection of Java classes interacts with a library. Figure 4.8 illustrates that concept via screenshots from a tool implementation.

Pinning Software Elements. Engineers grab architecture-level elements from the software visualization embedding the virtual whiteboards and pin them on a whiteboard $\diamond 1$. For instance, in the tool implementation used in the case study, users pin Java classes, interfaces, and folders/clusters – they can choose whether they wish to work with folders/packages or based on the results of the software clustering technique presented in Paper B when importing a system into the VR visualization.

Pinning a visual elements onto a whiteboard leaves behind a small pin representing the visual element. Such pins include a small avatar version that helps engineers with distinguishing between pins for different elements $\diamond 5$. The pinning mechanism enables engineers to swiftly add software elements to a diagram when working in a VR visualization, while at the same time establishing a direct link to the software element. The note-taking method automatically provides engineers with information on references between pinned elements by blending in arced, semi-transparent lines between them (see Figure 4.8).

Interacting with Pins. Engineers position and move pins freely on virtual whiteboards by grabbing them with their virtual hands and releasing them at the desired location $\diamond 6$. Further, they access detailed information on pins



Figure 4.8: Screenshots of the VR method for taking notes on software structures in VR as presented in Paper E.

by tapping on them with their virtual fingertips $\diamond 2$. That opens a user interface $\diamond 3$. For instance, the user interface displayed in $\diamond 3$ contains information on a pinned Java package. For one, it shows metrics on the number of contained classes and sub-packages. Further, it provides a list of references pointing to the pinned element as well as a list of references declared within the pinned element pointing to other elements in the system. Thereby, user interfaces for pins provide engineers with navigation shortcuts to related elements on the whiteboard directly, without needing to return to the visualization. Lastly, pins for packages/clusters can automatically be replaced with pins for their constituent elements. In the tool implementation, users do this by pressing a button in the user interface (see arrow pointing from $\diamond 3$ to $\diamond 4$).

Freehand Drawing. Engineers annotate arrangements of pins by grabbing a virtual pen $\diamond 11$ and drawing freely $\diamond 8$ - $\diamond 10$ – as they would on a regular real-world whiteboard. The virtual pen distinguishes between three drawing modes for the freehand strokes it registers on a virtual whiteboard:

- *Uninterpreted Drawing:* Pen strokes simply change the whiteboard's surface texture $\diamond 8$, there is no further interpretation (as opposed to the other two drawing modes explained beneath). This mode is useful for handwriting, drawing icons, etc.
- *Module Drawing:* Pen strokes are treated as outlines around pins for defining so-called “modules”, i.e., collections of pinned software elements $\diamond 9$. The note-taking method automatically determines which pins are contained in a sketched module outline and stores the results in a model. Engineers can interact with outlined modules by tipping into their area, thereby opening a user interface similar to that for software elements.
- *Arrow Drawing:* Pen strokes are interpreted as arrows between modules drawn on the whiteboard, connecting the two modules closest to the starting and end position of the arrow pen stroke $\diamond 10$ (self-references are possible). Reference lines between pins in modules connected by a drawn arrow automatically adopt the color of the drawn arrow $\diamond 7$. Thereby, the note-taking method provides engineers with conformance checks between their sketches and the ground-truth source code; if no reference line across two modules changes color after connecting them with a hand-drawn arrow, the engineer knows that the arrow does not correspond to a reference relation in the source code.

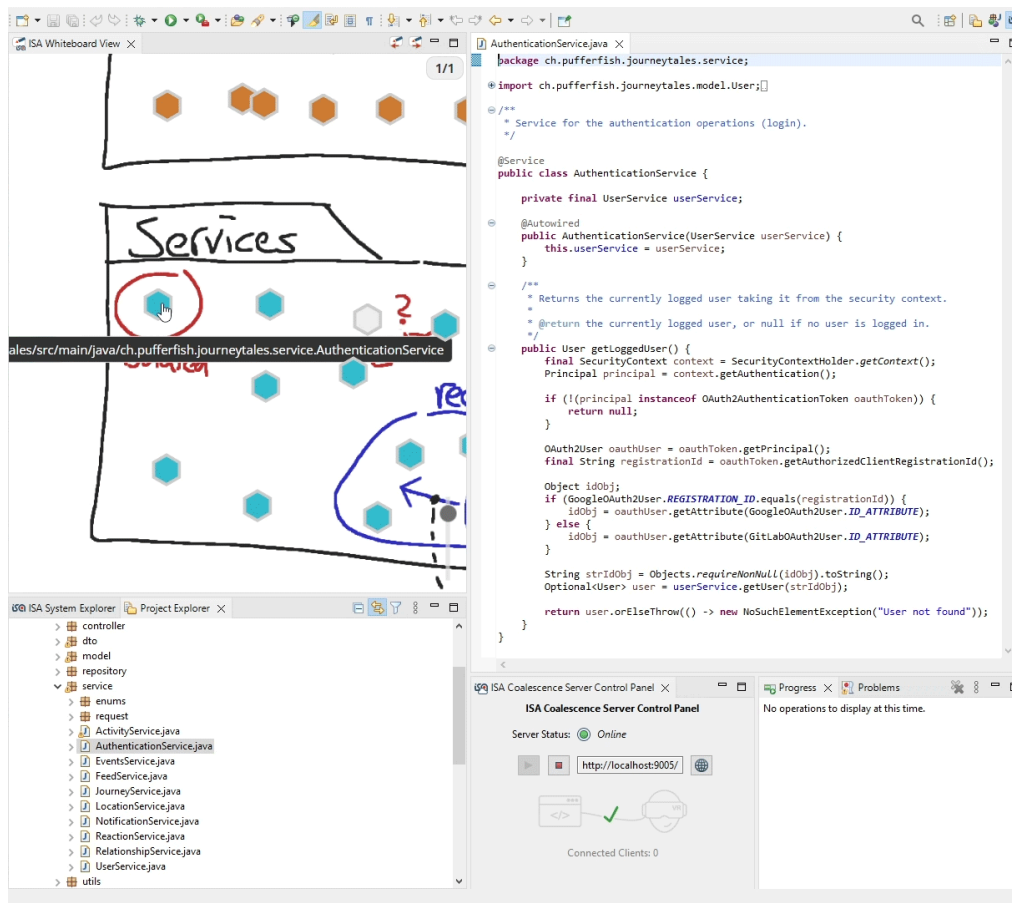


Figure 4.9: Screenshots of notes taken on a virtual whiteboard in VR displayed in an IDE (top left area). Users can interact with the whiteboard in the IDE, e.g., to open a pinned element’s source code in text form (top right area).

Access to Notes Outside of VR. Engineers create and edit notes in VR using the above-summarized mechanisms. However, a key use case for accessing notes is during programming tasks (e.g., to implement a planned change captured in notes on a whiteboard) – an activity commonly carried out in traditional 2D screen IDEs and (at least for now) not in VR. To grant engineers access to notes created on VR whiteboards in that scenario, the note-taking method provides them via a web interface, easing integration into existing tools such as an IDE. While freehand drawn notes can be adapted directly to 2D, three-dimensional structures are removed, i.e., relationship lines between pins that would cause visual clutter as well as avatar structures that require a third spatial dimension.

In my prototype implementation and its integration into the Immersive Software Archaeology tool, VR whiteboards are accessible via HTTP in the

browser and, based on that, directly in the Eclipse IDE for convenient access. Figure 4.9 illustrates that; the top left area of the figure shows a window in the IDE embedding a web browser displaying a virtual whiteboard. In the tool implementation, whiteboards receive live updates when edited in VR. Further, they provide engineers with features such as goto navigation, i.e., mouse-clicking on a pin opens the file of the pinned software element (e.g., a class) in the IDE (top right area of Figure 4.9).

Iterative Case Study with Practitioners. I conducted an iterative case study with seven software engineering practitioners from four companies to assess the quality of the VR note-taking method while improving it throughout the study iterations. In each iteration, participants used the VR note-taking method in individual sessions to solve tasks on analyzing a software system they were not familiar with and persisting the results. While participants were working on these tasks, I went through a catalog of open questions with them to gather feedback and suggestions for improvement. At the end of each iteration, I analyzed participants' responses to improve the note-taking concepts and implemented these improvements in tool prototype used in the study, before continuing with the next iteration. In total, the study consisted of three iterations.

In the paper, I summarize participants' feedback by grouping their statements into recurring topics. Based on that, I discuss how the VR note-taking method assists engineers in representing software architecture structures. Further, I investigate how the method assists engineers in reflecting on their notes in the sense of "Is this what the system looks like?" and "Should the system really look like this?"

Representing Software Structure. In the first iteration of the case study, participants left negative feedback on various issues emerging, such as problems with certain VR interactions – particularly, with virtual pens accidentally clipping through whiteboards while drawing. I addressed these problems by refining the note-taking concepts and tool implementation for all subsequent study iterations. On the other hand, participants positively mentioned intuitive interaction and the high flexibility offered by the VR note-taking method. As a side note, it should be mentioned that all tasks provided to participants were concerned with software architecture structure, as that is the primary target of the method. For capturing more fine-grain structure (such as on the level of functions/methods, or even statements) or other aspects of software, such as the behavior of a system as in data "flowing" through it, the results

might vary. For instance, participants mentioned operating on a relatively high abstraction level that would not allow them to capture interactions between individual methods as easily as between classes. However, for capturing views on an architectural level, participants perceived the workflow of the VR note-taking method, i.e., primarily pinning and drawing, as powerful and intuitive.

Reflecting on Software Structure. Participants' feedback related to reflecting on what they have noted on VR whiteboards was centered around two main points of feedback. First, by design, the level of abstraction the VR note-taking method focuses on primarily benefits an architectural overview and reflections on it (in alignment with the comments from above). Second, for that purpose, however, participants positively emphasized the overview of relationships between architectural elements they were provided with – especially when comparing the VR note-taking method with their usual workflow in IDEs or with freehand sketching on a physical whiteboard. Participants account this to the automatically shown reference lines between pins and to the conformance checks these provide based on hand-drawn arrows.

Addressed Research Questions

Paper E contributes to RQ₃ via concepts and a tool implementation for note-taking in VR software visualizations. Further, it provides insights into the strengths and weaknesses of the method via an iterative case study with software engineering practitioners.

4.2.5 Collaborative Software Exploration and Note-Taking (Papers F and G)

Papers F and G address Research Gap 4 (Section 2.6.1) and Research Gap 5 (Section 2.6.2) via a method for a synchronized collaborative VR software visualization method with multi-media note-taking capabilities. The presented method employs a metaphor representing software architecture as its primary visual structure, thus addressing Research Gap 2. Further, Paper F reports on a case study that investigates how collaborative VR software visualizations are used by practitioners when exploring an unfamiliar system together.

Figure 4.10 highlights these contributions in the scope of this thesis. Paper F presents concepts and a case study while Paper G focuses on technical details of a tool implementation.

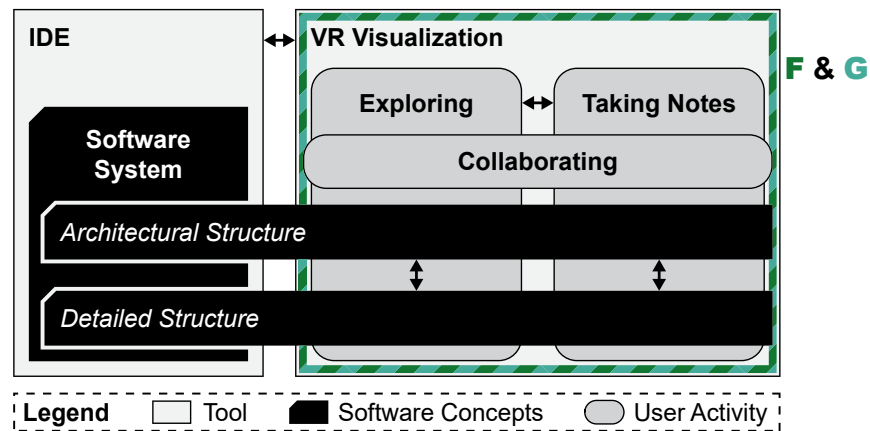


Figure 4.10: Overview of how Papers F and G contribute to the overall thesis.

Results

In the following, I summarize the results presented in Papers F and G.

Collaborative Software Visualization in VR. Papers F and G present a collaborative VR software visualization technique that combines and enhances concepts presented in the papers summarized above. Multiple engineers can simultaneously immerse into one synchronized visual representation of a subject software system and thus explore its structure while taking notes on findings, planning changes, and so on. Figure 4.11 provides example screenshots of this from the tool implementation presented in Paper G. Because I used Java as a target language in the tool implementation, I use Java terminology in the summary beneath. However, the summarized concepts can be applied to other programming languages – object-oriented languages require no adaptations.

Folders as Semi-Transparent Spheres. Software elements on an architectural level are represented as semi-transparent spheres (Figure 4.11 (a) and (b)). Conceptually, these represent folders, Java packages, clusters resulting from an analysis such as that presented in Paper B, or other hierarchical organizations of source elements. For brevity, and to remain consistent with the terminology used in Papers F and G, I refer to them as “folder spheres” in the following.

To help engineers distinguish between different folder spheres, these receive a unique color that is determined by their position in the system’s hierarchy of folders, packages, clusters, etc. To foster a top-down exploration

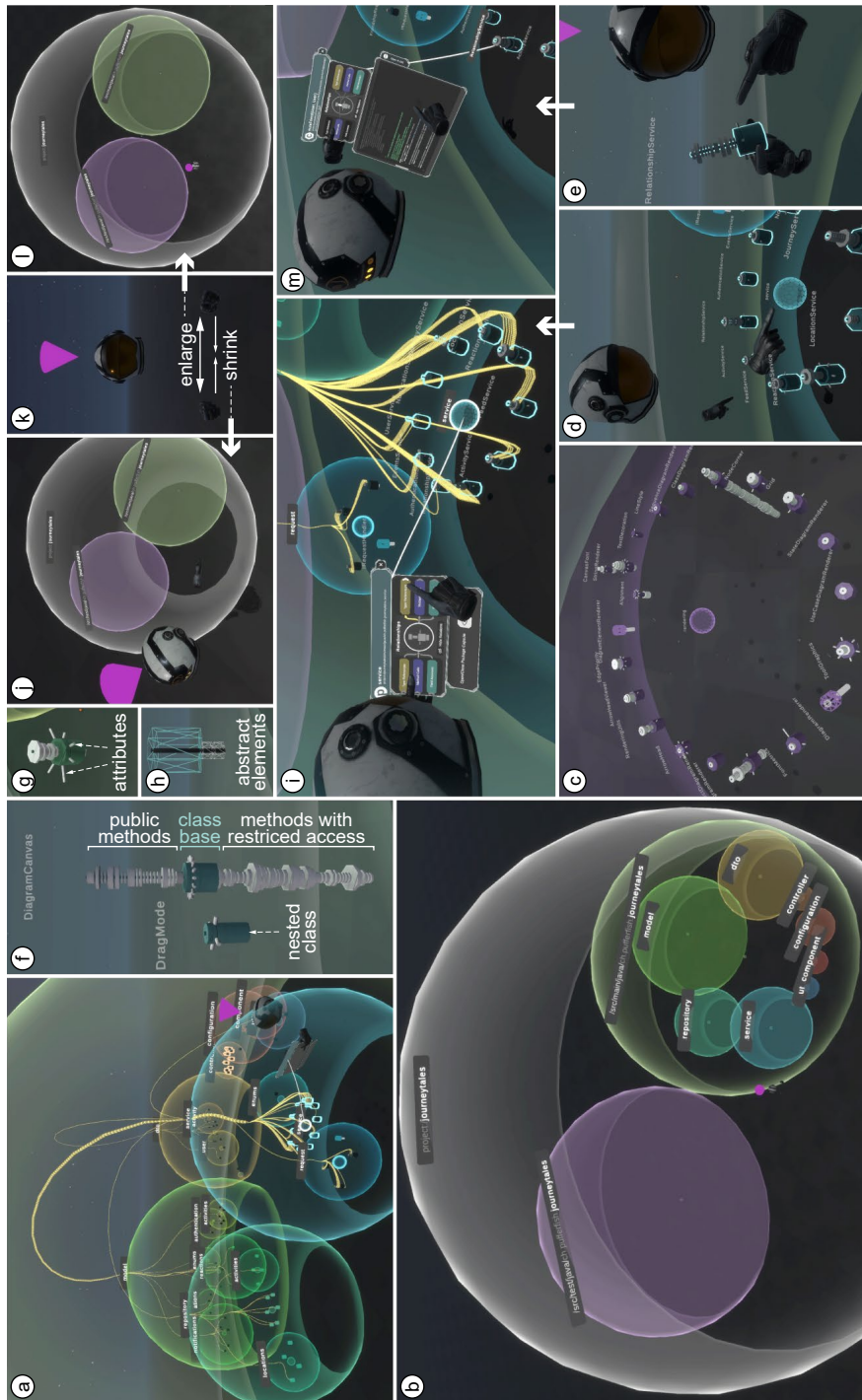


Figure 4.11: Screenshots taken from the tool presented in Paper G implementing the method presented in Paper F. Folders/packages are represented as colored nested spheres (a-c). Classes, interfaces, etc. and their members are represented as stacks of cylinders (class cylinders), encoding metainformation and structural metrics via location, shading, and form (f-h). Engineers can interact with elements (d, e) to blend in visual relationships graphs (i, a), read code (m), or scale the visualization up and down (j-l).

approach and to reduce the amount of information presented to engineers, folder spheres initially hide their contained elements. Engineers interact with them to reveal content of interest on demand. For instance, in Figure 4.11 (b), compare the purple folder sphere on the left-hand side (closed, hiding its content) with the green folder sphere on the right-hand side (opened, showing contained sub-spheres). Thereby, the method presented in Papers F and G adopts the semantic zoom presented in Papers A-C seamlessly and more gradually.

Classes as Stacks of Cylinders. A software element on the level of files or classes is represented as a stack of cylinders (see Figure 4.11 (f)-(h)), in short, referred to as a “class cylinder”. Each class cylinder consists of three types of visual constituents:

- One “base cylinder” that represents the class itself, adopting its color from the direct parent folder sphere and thus communicating the containment relationship. The surface of a base cylinder representing an interface is rendered in a wireframe look, indicating its abstract nature (h).
- An arbitrary amount of cylinders for methods and constructors (referred to as “method cylinders” for brevity). Their shape is determined similarly to how building floors are shaped in the visualization metaphor proposed in Paper B (Section 4.2.2). That is, the height is determined by counting the expressions contained in the visualized method/constructor. The diameter is determined from a metric on cognitive complexity [176]. To further provide a brief overview of the API of a class, cylinders for system-wide accessible methods (`public` keyword in Java) stack on top of the base cylinder, while encapsulated methods (`private`, `protected`, or `package visible` in Java) stack beneath the base cylinder. Similar to base cylinders, the surface of a method cylinder representing an abstract method or interface method is rendered in a wireframe look (h).
- An arbitrary amount of “attribute spikes” representing class attributes/-variables, originating from the base cylinder. Attribute spikes communicate accessibility via their length, i.e., publicly accessible attributes are noticeably longer than encapsulated ones (see Figure 4.11 (g)).

By visualizing classes with the proposed technique, engineers are provided with a visual overview of their content (the example shown in Figure 4.11 (f)

illustrates this well). Figure 4.11 © shows class cylinders within their containing folder sphere. They are arranged in a circle parallel to the floor to provide engineers with convenient access.

Interaction with Visual Elements. Engineers interact with folder spheres, class cylinders and their constituent elements, and the entire system via different interaction mechanisms. To access and reveal further information, they tip on the visual element of interest with their virtual index finger (Figure 4.11 ④ and ⑤). That causes folder spheres to reveal their content and class cylinders (and their constituent elements) to show a user interface with additional information, including the element's source code ⑥. Further, engineers optically zoom in and out via hand gestures ⑦-⑧.

Relationship Graph. The method presented in Papers F and G provides engineers with an overview of references between software elements similar to the method presented in Paper B. However, the relationship graphs presented in Papers F and G provide (a) more fine-grain control over what is displayed and (b) more detailed information on which elements are referenced. That is, the relationship graphs presented in Paper F distinguish between different types of references, enabling engineers to blend these in separately. In the tool implementation presented in Paper G, this distinction is made between type references, method calls, and field accesses in Java source code.

Further, the individual reference lines originate from and point to representations of particular class members. For instance, a reference line showing a call between two methods originates from the cylinder representing the calling method, pointing to the cylinder representing the called method. Engineers control the relationship graphs via the user interface they can open on elements (summarized above and shown in Figure 4.11 ⑨ and ⑩).

Synchronization and Presence of Collaborators. Engineers see their collaborators in the shared, synchronized virtual space around them. For that purpose, the visualization method updates and displays collaborators' heads and hands in real-time. The visualization further synchronizes the grabbing of elements (e.g., Figure 4.11 © shows a collaborator currently holding a class cylinder), interactions with user interfaces such as when scrolling through source code, and blending in or out references in the relationship graph.

Collaborative Note-Taking in VR. The VR visualization method presented in Papers F and G integrates and extends the VR note-taking method presented in Paper E in two aspects.

First, it provides all the functionality presented in Paper E in a collaborative setting where whiteboards are shared between them. Engineers can each pin elements to the same whiteboard and pick up a pen and draw while seeing the results of whiteboard edits made by their peers. Thereby, they can collaboratively take notes of findings and complement each others' views. Paper G explains how the collaborative VR note-taking method verifies edits performed by collaborating engineers to ensure their consistent execution.

Second, the method presented in Papers F and G extends the VR whiteboards presented in Paper E with additional means for taking notes. Engineers record audio notes via a virtual microphone, enabling them to attach replayable audio pins to whiteboards. Further, they take in-visualization screenshots via a virtual camera, enabling them to attach specific views (as bitmaps) on parts of the visualization to whiteboards. These screenshots act as temporal savepoints because engineers can restore the visualization to the state it was in when using the camera, instantly restoring the scaling of elements, open/close states of each folder sphere, and the visibility of lines in the relationship graph. These new features provide engineers with a means for effectively expressing thoughts and insights.

Case Study with Practitioners. In Paper F, I present a case study with four software engineering practitioners assessing the strengths and weaknesses of the visualization method summarized above. In pairs of two, the engineers collaboratively explored a software system they had not seen before to gather insights into its inner workings. I did not interfere with that exploration except by responding to technical questions raised by the participants regarding tool functionality and VR hardware.

During the collaborative exploration sessions, I screen-recorded participants' activity in VR. Further, I gathered their verdict on the method via an anonymized, individually filled-in post-session questionnaire. Lastly, I collected information on the quality of participants' insights by forwarding them to the subject system's original developers. By analyzing this data, I answered research questions related to using a collaborative VR software exploration method. In the following, I summarize the results.

How do engineers explore and take notes? I analyzed participants' activity from the recordings of each VR exploration session to create a detailed time-

line. Thereby, I identified patterns in participants' behavior. It shows that engineers using a collaborative VR software exploration tool oscillate between examining architectural structure in the visualization and reading through details of source code as text. When taking notes on findings, they primarily use handwriting and audio recordings. In the study, participants required practice to become used to handwriting on a virtual whiteboard and thus used it only to annotate individual words, outline pins, and draw arrows, relying on audio recordings to capture longer thoughts. Over the course of both sessions, I observed ample communication between participants, i.e., ongoing discussions and synchronization of insights and assumptions, interrupted by short phases of individual exploration.

What strengths and weaknesses do engineers perceive? Based on participants' answers to the post-session questionnaire, I extracted their verdict on different exploration and note-taking mechanisms and how to improve these. Overall, participants felt more supported in exploring the subject system than in taking notes on their findings. They especially valued the aspect of exploring an unknown system in a collaborative VR tool thanks to the overview. For note-taking, participants state that handwriting longer notes on VR whiteboards is not desirable. Generally, they perceive issues with handwriting legible notes due to their lack of VR experience. For capturing longer and more general thoughts, they favor audio recordings, strongly wishing for automated speech-to-text transformation. However, the virtual whiteboards were perceived as overall very useful. Further, participants rated the collaborative aspect of the method as highly positive.

What type of insights do engineers extract? I collected insights into the subject system's source code gathered by both pairs of participants and identified patterns in these. The results show that participants' insights are mainly focused on system/architecture level concerns – which is in line with the observations on participants' exploration style summarized above. Further, I forwarded all insights to the subject system's original developers to assess their correctness and relevance. As per the verdict of the original developers, participants' insights were correct (average of 4.43 on a scale from 1 to 5) with varying relevance (average of 3.61 on a scale from 1 to 5). I conclude that while further studies are needed to investigate these results in more detail, the correctness of participants' notes along with their feedback and the above-summarized observations are promising.

Addressed Research Questions

With the contributions summarized above, Papers F and G contribute to RQ₁-RQ₄. They present a new VR software visualization method along with a case study on its use by practitioners, thus contributing to RQ₁ and RQ₂. Further, they extend the note-taking method presented in Paper E, thus contributing to RQ₃. Lastly, Paper F investigates how teams of engineers utilize collaborative VR visualizations for software exploration (RQ₄).

4.3 Summary of Contributions

Table 4.1 summarizes the contributions made in Papers A-G and maps these to research questions (Chapter 3) they provide insights to.

Table 4.1: Overview of Papers A-G and how they contribute to answering RQ₁-RQ₄.

	Paper A	Papers B & C	Paper D	Paper E	Papers F & G
RQ₁	Concepts for VR visualization of software architecture	Concepts and tool prototype for: <ul style="list-style-type: none"> • Fully-Automated Software Clustering • VR Software Visualization (Solar System Metaphor) 	Concepts and tool prototype for generating 3D landmarks in large-scale software visualizations	X	Concepts and tool prototype for VR Visualization (Spheres and Cylinders Metaphor)
RQ₂	X	Controlled experiment comparing VR software visualization to standard IDE	X	X	Case study investigating collaborative software exploration in VR
RQ₃	X	X	X	Concepts and tool prototype for freehand note-taking in VR with conformance checks Iterative case study assessing strength and weaknesses	Concepts and tool prototype for collaborative multi-media note-taking in VR Case study assessing strength and weaknesses
RQ₄	X	X	X	X	Concepts and tool prototype for collaborative software exploration via synchronized visualization in VR Case study investigating collaborative software exploration in VR

Conclusion

In the following, I summarize how the work presented in Chapter 4 provides answers to the research questions formulated in Chapter 3. Thereby, I conclude whether the thesis formulated in Section 3.2 holds:

VR visualization is suitable for exploring unfamiliar software systems because it provides engineers with an overview of system architecture, an environment fostering collaborative work with peers, and powerful interaction mechanisms, e.g., for querying detailed information or for taking notes on findings.

5.1 Fostering Exploration of Software Architecture

Paper A describes a vision for a VR software visualization method using a solar system metaphor for representing architectural software elements (modules, folders, packages, clusters, etc.). That vision includes concepts for enabling engineers to switch between abstraction levels and to access details on demand, most notably through a relationship graph explicitly visualizing source code references.

Papers B and C extend that vision into concrete concepts and provide a tool implementation of these. In doing so, Paper B presents an automated software clustering procedure as the basis for VR visualization that enables engineers to inspect architecture beyond mere folder structure. Further, Paper B reports on a controlled experiment assessing the suitability of the above

concepts for assisting engineers in software comprehension tasks. The experiment compares the method envisioned and implemented in Papers A-C with another state-of-the-art VR software visualization tool and a traditional 2D screen IDE. This allows for conclusions on differences between comprehending subject systems in an IDE as compared to VR software visualizations.

Paper D presents a method for using unique, remarkable landmarks in 3D software visualizations for breaking up repetitive patterns in their landscapes of visual elements – a problem stemming from the intentionally simple building rules in 3D visualizations that impedes viewers' orientation.

Papers F and G present another VR software visualization metaphor encoding software elements and their properties via abstract spherical and cylindrical shapes. This metaphor adopts the concepts presented in Papers A-C while providing enhancements to both the semantic zoom and the relationship graph. Further, Paper F presents a case study investigating strengths and weaknesses of the presented VR visualization method for software exploration.

RQ₁: *How can software be visualized in VR so that it fosters engineers' exploration of architectural structure?*

The two methods presented in Papers B and F visualize software architecture as their first-level visual structure in an immersive VR environment. In both methods, engineers interact with visual elements in VR, e.g., to navigate through the visualization or to query further information on demand. The controlled experiment presented in Paper B shows that, for this purpose, estimating architecture via clustering can be valuable, because it groups together visual representations of software elements that are strongly related (in the sense of references in source code), easing the access to information.

In software architecture visualizations in VR suffering from repetitive visual patterns, landmarks can be used to make different parts of the visualization more unique and remarkable.

Lastly, the experiment presented in Paper B shows that visualizing relationships is valuable for engineers. In the methods presented in Papers B and F, this is achieved through lines between visual elements. By utilizing three spatial dimensions, VR visualizations can effectively avoid line overlapping while stereoscopic vision and motion parallax effects enable engineers to perceive this naturally. The experiment demonstrates that source code reference visualizations help engineers with relating software elements, especially on higher abstraction levels such as “How do the classes in this Java package work together?”.

RQ₂: *What are strengths and weaknesses of using a VR software visualization for exploring architectural structure?*

The controlled experiment presented in Paper B suggests that VR software visualizations are suitable for gaining an overview of complex structures in a subject system. They provide engineers with a bird's eye perspective of a system's architecture which must largely be mentally imagined when reading through raw textual representations of source code in an IDE.

While this overview comes at a cost of insights into details, it helps with comprehending software architecture. The experiment presented in Paper B shows that, because the visual summary provided in a visualization answers many high-level questions without reading through code, engineers are less familiar with implementation details as if they needed to extract the big picture by reading through source code as text. However, participants using a VR visualization develop a better understanding of the dimensions of software structures on an architectural level when compared to participants using an IDE. Visualizations of references between software elements play a pivotal role in this. In textual representations of source code, references are not directly visible and thus hard to overview and cumbersome to navigate whereas, in a VR visualization, they can be directly displayed and interacted with while avoiding line overlap. The case study presented in Paper F reflects this; engineers feel strongly supported in exploring software architecture due to the overview provided in VR.

Conclusion. Regarding RQ₁ and RQ₂, I conclude that my thesis holds. VR Visualization is useful for providing engineers with a bird's eye perspective on a subject system. Estimating a system's architectural structure via automated clustering is a beneficial first step for that purpose because it allows for sensible arrangements of a system's artifacts beyond potentially outdated and inaccurate folder structure. Further, representations of relationships between software elements are helpful, especially on higher abstraction levels. VR is a suitable medium for software visualization because it provides powerful inspection and interaction possibilities that utilize three spatial dimensions.

5.2 Note-Taking in Virtual Reality Software Visualization

Paper E presents a method for extending existing VR software visualizations with note-taking capabilities. The presented concepts are centered around the idea of providing engineers with a virtual whiteboard on which they take

notes by pinning visual elements (e.g., a sphere representing a Java package) and drawing freely as they would on a physical whiteboard in the real world. Based on these notes, the presented concepts include automated checks providing engineers with feedback on the conformance of their sketches to the source code structure these depict. I assessed the strengths and weaknesses of the note-taking method in an iterative case study, throughout which I extended and improved its concepts.

Papers F and G extend the whiteboard note-taking method presented in Paper E with capabilities for recording audio notes and creating screenshots in VR. Further, Paper F presents a case study assessing the extended note-taking method (including the concepts presented in Paper E), thereby yielding more insights into the strengths and weaknesses of note-taking in VR.

RQ₃: *How can engineers be supported in taking notes during software exploration sessions in a VR visualization without disrupting their immersive exploration?*

The note-taking concepts presented in Papers E, F, and G are designed to integrate with a VR software visualization and its model of a system's source code, thus establishing a direct link between notes taken and the software structure they describe. They are examples of how VR provides potential for novel interaction metaphors with source code.

One of the presented VR note-taking concepts enables engineers to grab elements from the VR visualization (i.e., in Paper E, Java classes and packages) and to pin them to a virtual whiteboard via hand movement, enabling further tool support. For instance, it allows explicitly visualizing references between pinned elements via lines between them or augmenting pins with visual summaries of the pinned element (similarly to how the element is represented in the visualization).

Along with the pinning interaction summarized above, Paper E presents concepts for VR freehand drawing on virtual whiteboards. Engineers outline pins on a whiteboard and draw arrows between outlines to take notes on a system's structure and relationships between parts of it. Because these freehand-drawn notes are created in a purely virtual realm, the method is able to establish a model capturing the semantics of freehand-drawn lines. Again, this enables further tool support and automated analysis, e.g., for conformance checks between the drawn lines and references in the subject system's source code.

Lastly, Papers F and G present means for swiftly capturing general thoughts and insights via audio recordings and screenshots. Audio record-

ings are a convenient way for engineers to capture longer thoughts and insights. They have a potential for further utilization, e.g., via natural language processing and conversion to written form or as input for other tools. Screenshots are useful for swiftly capturing a view in a VR visualization. Similarly, they have potential for further utilization, e.g., besides acting as purely visual artifacts, screenshots can act as temporal snapshots for saving the state of a visualization at the time point of their creation.

The VR note-taking and interaction concepts summarized above enable engineers to capture insights and plans, especially on a subject system's architectural structure. While notes taken in this manner are three-dimensional when editing and interacting, it is possible to convert them to two-dimensional representations without introducing clutter. In the method presented in Papers E, F, and G, this process removes information – such as relation lines between pins which, in 2D, would lead to inevitable chaos. Nevertheless, it allows mirroring notes to a 2D standard screen, e.g., for integration into an IDE, while explicitly maintaining a connection to the depicted source code elements.

The case studies presented in Papers E and F show that, although VR-specific technical challenges to note-taking do exist, engineers perceive note-taking in VR as overall suitable and intuitive. In the concepts and prototypes presented in Papers E, F, and G, challenges and problems include, most notably, hands and pens clipping through whiteboards and issues with legible VR handwriting in general – both accelerated by engineers' inexperience with VR.

On the other hand, engineers described the note-taking approach of pinning elements from the visualization and annotating them via freehand drawing as flexible, powerful, and visually intuitive. VR screenshots were perceived as useful, although they were not used in the case study presented in Paper F. Audio recordings were rated useful in combination with further natural language processing such as an automated conversion to textual form. All in all, both studies show that VR is a suitable medium for assisting engineers in taking notes on software structures or plans for changes to these as well as for reflecting on notes taken.

Conclusion. Regarding RQ₃, I conclude that my thesis holds. The above means demonstrate how VR and its possibilities to interact with visualizations can be utilized to enable engineers to take notes during software exploration sessions without disrupting their immersion and focus on the task. While

challenges in technical implementations remain, engineers perceive the concepts as suitable for capturing software structure, especially on an architectural level.

5.3 Collaborative Software Exploration in Virtual Reality

Papers F and G present concepts and technical details on a collaborative VR software visualization method immersing multiple engineers into a shared and synchronized visualization of a subject system. Further, Paper F presents a case study that investigates engineers' exploration behavior in a collaborative VR visualization as well as the quality of insights they gather about a subject system they are not familiar with.

RQ₄: *How do teams of software engineers explore systems in a collaborative multi-user VR visualization?*

Currently, VR is a niche technology. In the case study presented in Paper F, engineers were not experienced in using VR except for rare exceptional occurrences. As a result, engineers self-assessed that they were slow in executing tasks due to their unfamiliarity with using VR and the interaction mechanisms employed in the study.

Despite the above limitation, engineers stated they felt strongly supported in collaboratively exploring the subject system, especially its architectural structure. They engage in vivid communication, e.g., to synchronize with their peers on new findings or to challenge previous assumptions. Such phases of communication alternate with short periods of individual detailed inspection, mostly in the form of briefly reading through source code. Mainly, however, engineers explore the subject system via the visualized structure and via sketches on VR whiteboards. Further, in the entirely unguided exploration conducted throughout the study, engineers yielded correct results as per the verdict of the original developers of the subject system.

Conclusion. Regarding RQ₄, I conclude that my thesis holds. Despite their unfamiliarity with VR as a medium, engineers are able to utilize VR software visualization in collaborative sessions with peers where they engage in vivid communication and gain a correct understanding of the visualized system.

Future Work

In this chapter, I touch upon future work in XR software visualization that extends upon the research presented in this thesis. I elaborate on further studies assessing concepts presented in Papers A-G, visualizations of software execution behavior, and code editing capability in XR software visualization.

6.1 Further Empirical Studies on VR Software Visualization

While the experiments and studies presented in Papers A-G provide answers to multiple as of before unaddressed research questions in VR software visualization, there remains a variety of further open questions that require empirical studies.

One unaddressed facet of software exploration in VR is its influence on engineers' knowledge retention, i.e., their ability to remember insights into a subject system's source code over longer periods of absence. For instance, a controlled experiment could be useful for comparing VR software visualization with the traditional IDE in this matter.

Regarding note-taking, further studies are needed to investigate engineers' usage of notes taken in VR when implementing changes in an IDE. For instance, it is unclear how useful engineers perceive different kinds of notes taken in VR for carrying out this activity.

Lastly, it requires further studies to extend our understanding of collaborative software exploration via multi-user VR visualization. Especially, studies with larger team sizes are needed to clarify remaining questions. How suitable are existing concepts for supporting teams of more than two engineers

in exploring software? How does the exploration behavior change when larger teams explore software compared to pairs of developers, e.g., regarding communication? What additional features are useful or even necessary?

6.2 VR Visualizations of Execution Behavior

In the work presented in this thesis, I focus on visualizations of software structure. These play a pivotal role as, often, they act as a basis for visualizing other software characteristics such as execution behavior, code quality, or system evolution. Combining visualizations of these different kinds of software characteristics provides engineers with more information, assisting them in investigating a subject system.

Visualizations of a system's execution behavior – as in messages being passed through its code structure – provide engineers with relevant insights. Paper A outlines an idea where concrete messages in a recording of a system's execution are visualized as entities traveling from one structural element to another. That is useful for identifying parts of a system that are heavily executed and thus relevant when trying to understand the system's inner workings. In addition to that, VR visualizations of software execution behavior could enable engineers to interact with messages to inspect the data being passed through the system.

Another approach could be to attach VR visualizations to the execution environment of a subject system and, based on that, to provide engineers with VR interaction features similar to how they are present in debuggers. Engineers could set breakpoints to halt the execution at certain locations in the code, modify the content of messages being passed through a system at runtime, or artificially trigger the execution of a method to understand its effects.

More tools and empirical studies are needed to investigate whether and how similar methods can be utilized to assist engineers in comprehending software systems.

6.3 Code Editing in X-Reality

Tools and solutions presented in this thesis are view-only, and intended for pure software exploration and comprehension. However, upon discovering a re-engineering opportunity during an exploration session, being able to perform edits directly from within XR – while benefiting from a visual overview of the subject system – can be helpful for engineers.

Because AR and MR visualization tools leave engineers in their real-world surroundings, these can implement code editing by enabling engineers to use physical keyboards to perform textual changes to source code, similar to how they would do it in an IDE. While feasible interaction-wise, this kind of low-level editing does not necessarily match the strong suit of system-level visualizations and the exploration style these foster. Instead, code edits in XR, especially VR, could be provided based on interactions with elements of the visualization.

Simple refactorings such as relocating existing software elements could be directly supported via XR interaction, e.g., relocating a building in a city metaphor to move a file from one folder to another. More advanced operations, including the creation of entirely new functionality, would require more complex techniques. For creating new functionality, XR visualizations could enable engineers to spawn new visual elements (e.g., a new building in a city metaphor), initialized without content. Based on that, visualizations could utilize generative artificial intelligence to fill in empty structures created in that manner, e.g., based on speech queries recorded in XR. This process could be guided via further context, e.g., by processing users' hand gestures, gaze, and location in the system, among others. For instance, if the engineer is located in a "util" package, staring at a particular visual element, this information could be used to assist the code generation process by putting a spoken query into context. Researching methods akin to that briefly sketched above could yield relevant results for future development environments.

Bibliography

- [1] Mackinlay Card. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.
- [2] Colin Ware. *Information visualization: perception for design*. The Morgan Kaufmann series in interactive technologies. Morgan Kaufman, San Francisco, CA, 2004.
- [3] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.
- [4] Frederick Brooks and H Kugler. *No silver bullet*. April, 1987.
- [5] Vladimir Averbukh, Natalya Averbukh, Pavel Vasev, Ilya Gvozdarev, Georgy Levchuk, Leonid Melkozerov, and Igor Mikhaylov. Metaphors for Software Visualization Systems Based on Virtual Reality. In *Augmented Reality, Virtual Reality, and Computer Graphics*, volume 11613, pages 60–70. Springer International Publishing, Cham, 2019. Series Title: Lecture Notes in Computer Science.
- [6] Catherine Plaisant, Jesse Grosjean, and Benjamin B Bederson. Space-Tree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation. 2002.
- [7] O. Greevy, M. Lanza, and C. Wyseier. Visualizing Feature Interaction in 3-D. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, Budapest, Hungary, 2005. IEEE.
- [8] M. Lanza and S. Ducasse. Polymetric views - A lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

-
- [9] M. Balzer and O. Deussen. Hierarchy Based 3D Visualization of Large Software Structures. In *IEEE Visualization 2004*, pages 4p–4p, Austin, TX, USA, 2004. IEEE Comput. Soc.
 - [10] Danny Holten, Roel Vliegen, and Jarke J. Van Wijk. Visual Realism for the Visualization of Software Metrics. In *In Proceedings of Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 1–6. Springer, 2005.
 - [11] Danny Holten, Roel Vliegen, and Jarke van Wijk. Visualization of Software Metrics using Computer Graphics Techniques. January 2006.
 - [12] Ugo Erra and Giuseppe Scanniello. Towards the visualization of software systems as 3D forests: the CodeTrees environment. page 8, 2012.
 - [13] Ugo Erra, Giuseppe Scanniello, and Nicola Capece. Visualizing the Evolution of Software Systems Using the Forest Metaphor. In *2012 16th International Conference on Information Visualisation*, pages 87–92, Montpellier, France, July 2012. IEEE.
 - [14] Dian Li, Weidong Wang, and Yang Zhao. Intelligent Visual Representation for Java Code Data in the Field of Software Engineering Based on Remote Sensing Techniques. *Electronics*, 12(24):5009, 2023. Publisher: MDPI.
 - [15] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, page 921, Leipzig, Germany, 2008. ACM Press.
 - [16] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 551, Waikiki, Honolulu, HI, USA, 2011. ACM Press.
 - [17] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. CityVR: Gameful Software Visualization. page 5, 2017.
 - [18] Juraj Vincur, Pavol Navrat, and Ivan Polasek. VR City: Software Analysis in Virtual Reality Environment. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 509–516, Prague, Czech Republic, July 2017. IEEE.

-
- [19] David Moreno-Lumbreras, Jesus M. Gonzalez-Barahona, Gregorio Robles, and Valerio Cosentino. The influence of the city metaphor and its derivatives in software visualization. *Journal of Systems and Software*, 210:111985, April 2024.
- [20] Elke Franziska Heidmann, Annika Meinecke, Lynn von Kurnatowski, and Andreas Schreiber. Towards visualization of evolution of component-based software architectures in VR. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, pages 61–64, Porto Portugal, March 2020. ACM.
- [21] Elke Franziska Heidmann, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. Visualization of Evolution of Component-Based Software Architectures in Virtual Reality. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 12–21, Adelaide, Australia, September 2020. IEEE.
- [22] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics. page 7, 2004.
- [23] Roy Oberhauser, Christian Silfang, and Carsten Lecon. Code structure visualization using 3D-flythrough. In *2016 11th International Conference on Computer Science & Education (ICCSE)*, pages 365–370. IEEE, 2016.
- [24] Roy Oberhauser and Carsten Lecon. Virtual Reality Flythrough of Program Code Structures. In *Proceedings of the Virtual Reality International Conference - Laval Virtual 2017 on - VRIC '17*, pages 1–4, Laval, France, 2017. ACM Press.
- [25] Philipp A. Rauschnabel, Reto Felix, Chris Hinsch, Hamza Shahab, and Florian Alt. What is XR? Towards a Framework for Augmented and Virtual Reality. *Computers in Human Behavior*, 133:107289, August 2022.
- [26] Andreas Dieberger and Andrew U Frank. A city metaphor to support navigation in complex information spaces. *Journal of Visual Languages & Computing*, 9(6):597–622, 1998. Publisher: Elsevier.
- [27] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. On the use of virtual reality in software visualization:

- The case of the city metaphor. *Information and Software Technology*, 114:92–106, October 2019.
- [28] Clinton L. Jeffery. The City Metaphor in Software Visualization. In *Computer Science Research Notes*. Západoeská univerzita, 2019.
- [29] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. The city metaphor in software visualization: feelings, emotions, and thinking. *Multimedia Tools and Applications*, 78(23):33113–33149, December 2019.
- [30] A.R. Teyseyre and M.R. Campo. An Overview of 3D Software Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, January 2009.
- [31] Mei C. Chuah, Steven F. Roth, Joe Mattis, and John Kolojejchick. SDM: selective dynamic manipulation of visualizations. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 61–70, Pittsburgh Pennsylvania USA, December 1995. ACM.
- [32] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, April 1986.
- [33] Andrian Marcus, Louis Feng, and Jonathan I Maletic. 3D Representations for Software Visualization. page 11, 2003.
- [34] Hideki Koike. The role of another spatial dimension in software visualization. *ACM Transactions on Information Systems*, 11(3):266–286, July 1993.
- [35] Pierre Caserta and O Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, July 2011.
- [36] Adrian Kuhn, Peter Loretan, and Oscar Nierstrasz. Consistent Layout for Thematic Software Maps. page 10, 2008.
- [37] Adrian Kuhn, David Erni, Peter Loretan, and Oscar Nierstrasz. Software Cartography: thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, pages n/a–n/a, 2010.

-
- [38] George Robertson, Mary Czerwinski, Kevin Larson, Daniel C Robbins, David Thiel, and Maarten Van Dantzich. Data mountain: using spatial memory for document management. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 153–162, 1998.
- [39] Marcel Steinbeck and Rainer Koschke. TinySpline: A Small, yet Powerful Library for Interpolating, Transforming, and Querying NURBS, B-Splines, and Bézier Curves. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 572–576, Honolulu, HI, USA, March 2021. IEEE.
- [40] Brian Rogers and Maureen Graham. Motion Parallax as an Independent Cue for Depth Perception. *Perception*, 8:125–34, February 1979.
- [41] C Ware and G Franck. Viewing a graph in a virtual reality display is three times as good as a 2D diagram. page 2, 1994.
- [42] Andy Cockburn and Bruce McKenzie. Evaluating the Effectiveness of Spatial Memory in 2D and 3D Physical and Virtual Environments. *Spatial Cognition*, (4), 2002.
- [43] Maximilian Speicher, Brian D. Hall, and Michael Nebeling. What is Mixed Reality? In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–15, Glasgow Scotland Uk, May 2019. ACM.
- [44] Samuli Laato, Nannan Xi, and Velvet Spors. Making Sense of Reality: A Mapping of Terminology Related to Virtual Reality, Augmented Reality, Mixed Reality, XR and the Metaverse. 2023.
- [45] David Moreno-Lumbreras, Gregorio Robles, Daniel Izquierdo-Cortázar, and Jesus M. Gonzalez-Barahona. Software development metrics: to VR or not to VR. *Empirical Software Engineering*, 29(2):42, March 2024.
- [46] Katherine J. Mimnaugh, Evan G. Center, Markku Suomalainen, Israel Becerra, Eliezer Lozano, Rafael Murrieta-Cid, Timo Ojala, Steven M. LaValle, and Kara D. Federmeier. Virtual Reality Sickness Reduces Attention During Immersive Experiences, June 2023. arXiv:2306.13505 [cs, q-bio].

-
- [47] Daniel Hepperle and Matthias Wölfel. Similarities and Differences between Immersive Virtual Reality, Real World, and Computer Screens: A Systematic Scoping Review in Human Behavior Studies. *Multimodal Technologies and Interaction*, 7(6):56, 2023. Publisher: Multidisciplinary Digital Publishing Institute.
- [48] Geoffrey S. Hubona, Gregory W. Shirah, and David G. Fout. 3D object recognition with motion. In *CHI '97 extended abstracts on Human factors in computing systems looking to the future - CHI '97*, page 345, Atlanta, Georgia, 1997. ACM Press.
- [49] Andrew Hanson, Eric Wernert, and Stephen Hughes. Constrained Navigation Environments. May 2000.
- [50] George G. Robertson, Stuart K. Card, and Jack D. Mackinlay. Information visualization using 3D interactive animation. *Communications of the ACM*, 36(4):57–71, April 1993.
- [51] Jesus M Gonzalez-Barahona. Software development in the age of LLMs and XR. IDE 24, Lisbon, Portugal, 2024.
- [52] Marc-Oliver Rudel, Johannes Ganser, and Rainer Koschke. A Controlled Experiment on Spatial Orientation in VR-Based Software Cities. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 21–31, Madrid, September 2018. IEEE.
- [53] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Lanza Michele, and Jesus M. Gonzalez-Barahona. CodeCity: On-Screen or in Virtual Reality? other, September 2021.
- [54] Colin Ware, David Hui, and Glenn Franck. Visualizing Object Oriented Software in Three Dimensions. page 10, 1993.
- [55] Pourang Irani and Colin Ware. Diagrams based on structural object perception. In *Proceedings of the working conference on Advanced visual interfaces - AVI '00*, pages 61–67, Palermo, Italy, 2000. ACM Press.
- [56] M. Tavanti and M. Lind. 2D vs 3D, implications on spatial memory. In *IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.*, pages 139–145, San Diego, Ca, USA, 2001. IEEE.

-
- [57] Dheva Raja, Doug Bowman, John Lucas, Chris North, and Virginia Tech. Exploring the benefits of immersion in abstract information visualization. *8th Int'l Immersive Projection Technology Workshop (IPT '04)*, January 2004.
- [58] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. On the Impact of the Medium in the Effectiveness of 3D Software Visualizations. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 11–21. IEEE, 2017.
- [59] Jabier Martinez, Tewfik Ziadi, Raul Mazo, Tegawende F. Bissyande, Jacques Klein, and Yves Le Traon. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *2014 Second IEEE Working Conference on Software Visualization*, pages 50–59, Victoria, BC, Canada, September 2014. IEEE.
- [60] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, August 2014.
- [61] Carol V. Alexandru, Sebastian Proksch, Pooyan Behnamghader, and Harald C. Gall. Evo-Clocks: Software Evolution at a Glance. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 12–22, Cleveland, OH, USA, September 2019. IEEE.
- [62] Martin Beck, Jonas Trumper, and Jurgen Dollner. A visual analysis and design tool for planning software reengineerings. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8, Williamsburg, VA, USA, September 2011. IEEE.
- [63] Lucian Voinea and Alexandru C. Telea. Visual Clone Analysis with SolidSDD. In *2014 Second IEEE Working Conference on Software Visualization*, pages 79–82, Victoria, BC, Canada, September 2014. IEEE.
- [64] Roberto Minelli and Michele Lanza. SAMOA A Visual Software Analytics Platform for Mobile Applications. In *2013 IEEE International Conference on Software Maintenance*, pages 476–479, September 2013. ISSN: 1063-6773.

- [65] Nicolas Anquetil, Anne Etien, Gaelle Andreo, and Stephane Ducasse. Decomposing God Classes at Siemens. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 169–180, Cleveland, OH, USA, September 2019. IEEE.
- [66] Andrzej Zalewski, Szymon Kijas, and Dorota Sokoowska. *Capturing Architecture Evolution with Maps of Architectural Decisions 2.0*, volume 6903. September 2011. Pages: 96.
- [67] Sébastien Rufiange and Guy Melançon. *AniMatrix: A Matrix-Based Visualization of Software Evolution*. December 2014. Journal Abbreviation: Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014 Publication Title: Proceedings - 2nd IEEE Working Conference on Software Visualization, VISSOFT 2014.
- [68] Johannes Feiner and Keith Andrews. RepoVis: Visual Overviews and Full-Text Search in Software Repositories. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–11, Madrid, September 2018. IEEE.
- [69] Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. Performance Evolution Matrix: Visualizing Performance Variations Along Software Versions. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 1–11, Cleveland, OH, USA, September 2019. IEEE.
- [70] Peter Young. Three Dimensional Information Visualisation. 1996.
- [71] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone Trees: animated 3D visualizations of hierarchical information. In *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91*, pages 189–194, New Orleans, Louisiana, United States, 1991. ACM Press.
- [72] Jun Rekimoto and Mark Green. The information cube: Using transparency in 3d information visualization. In *Proceedings of the Third Annual Workshop on Information Technologies & Systems (WITS93)*, volume 13, pages 125–132, 1993.
- [73] Daniel A Reed, Keith A Shields, Will H Scullin, Luis F Tavera, and Christopher L Elford. Virtual reality and parallel systems performance analysis. *Computer*, 28(11):57–67, 1995. Publisher: IEEE.

-
- [74] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, Ischia, Italy, 1998. IEEE Comput. Soc.
- [75] G. Franck and C. Ware. Representing nodes and arcs in 3D networks. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 189–190, St. Louis, MO, USA, 1994. IEEE Comput. Soc. Press.
- [76] Glenn Franck, Monica Sardesai, and Colin Ware. Layout and structuring object oriented software in three dimensions. In *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*, page 22, 1995.
- [77] Colin Ware, Glenn Franck, Monica Parkhi, and Tim Dudley. Layout for Visualizing Large Software Structures in 3D. 2000.
- [78] Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. page 8, 1999.
- [79] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205, London, UK, 2000. IEEE Comput. Soc.
- [80] Claudio Riva. Visualizing software release histories with 3DSoftVis. In *INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, volume 22, pages 789–789. Citeseer, 2000.
- [81] J.I. Maletic, J. Leigh, and A. Marcus. Visualizing Software in an Immersive Virtual Reality Environment. page 6, 2001.
- [82] J.I. Maletic, J. Leigh, A. Marcus, and G. Dunlap. Visualizing object-oriented software in virtual reality. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 26–35, Toronto, Ont., Canada, 2001. IEEE Comput. Soc.
- [83] Stuart M Charters, Claire Knight, Nigel Thomas, and Malcolm Munro. Visualisation for Informed Decision Making; From Code to Components. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 765–772, 2002.

-
- [84] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *IEEE TCVG*, 2004.
 - [85] Michele Lanza. CodeCrawler - A Lightweight Software Visualization Tool. 2003.
 - [86] M. Lanza. CodeCrawler - polymetric views in action. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 394–395, Linz, Austria, 2004. IEEE.
 - [87] Michele Lanza and Stéphane Ducasse. CodeCrawler An Extensible and Language Independent 2D and 3D Software Visualization Tool. 2005.
 - [88] Michele Lanza, Stéphane Ducasse, Harald Gall, and Martin Pinzger. *CodeCrawler: an information visualization tool for program comprehension*. January 2005. Pages: 673.
 - [89] Orla Greevy, Michele Lanza, and Christoph Wyseier. Visualizing Live Software Systems in 3D. page 10, 2006.
 - [90] Thomas Panas, Thomas Panas, Rebecca Berrigan, and John Grundy. A 3D Metaphor for Software Production Visualization. page 6, 2003.
 - [91] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-configuration of software visualizations with Vizz3D. In *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05*, page 173, St. Louis, Missouri, 2005. ACM Press.
 - [92] Thomas Panas, Dan Quinlan, and Richard Vuduc. Analyzing and Visualizing Whole Program Architectures. page 4, 2007.
 - [93] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating Software Architecture using a Unified Single-View Visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228, Auckland, New Zealand, 2007. IEEE.
 - [94] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *2007 6th International Asia-Pacific Symposium on Visualization*, pages 133–140, Sydney, NSW, February 2007. IEEE.

- [95] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05*, page 214, Long Beach, CA, USA, 2005. ACM Press.
- [96] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Exploring the evolution of software quality with animated visualization. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 13–20, September 2008. ISSN: 1943-6106.
- [97] J.I. Maletic, A. Marcus, and L. Feng. Source Viewer 3D (sv3D) - a framework for software visualization. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 812–813, Portland, OR, USA, 2003. IEEE.
- [98] A. Marcus, L. Feng, and J.I. Maletic. Comprehension of software analysis data using 3D visualization. In *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pages 105–114, Portland, OR, USA, 2003. IEEE Comput. Soc.
- [99] Andrian Marcus, Louis Feng, and Jonathan I Maletic. Source Viewer 3D (sv3D) A System for Visualizing Multi Dimensional Software Analysis Data. page 2, 2003.
- [100] Xinrong Xie, D. Poshyvanyk, and A. Marcus. Support for Static Concept Location with sv3D. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, Budapest, Hungary, 2005. IEEE.
- [101] Xinrong Xie, Denys Poshyvanyk, and Andrian Marcus. 3D visualization for concept location in source code. In *Proceedings of the 28th international conference on Software engineering*, pages 839–842, Shanghai China, May 2006. ACM.
- [102] David Montano, Jairo Aponte, and Andrian Marcus. Sv3D meets Eclipse. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 51–54, Edmonton, AB, Canada, September 2009. IEEE.

-
- [103] Sazzadul Alam and Philippe Dugerdil. EvoSpaces Visualization Tool: Exploring Software Architecture in 3D. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 269–270, Vancouver, BC, Canada, October 2007. IEEE. ISSN: 1095-1350.
- [104] Philippe Dugerdil and Sazzadul Alam. Execution trace visualization in a 3D space. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 38–43. IEEE, 2008.
- [105] Michele Lanza, Harald Gall, and Philippe Dugerdil. EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 293–296, Kaiserslautern, Germany, 2009. IEEE.
- [106] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, Banff, AB, Canada, June 2007. IEEE.
- [107] Richard Wettel and Michele Lanza. CodeCity. 2008. Publisher: Citeseer.
- [108] Richard Wettel and Michele Lanza. Visual Exploration of Large-Scale System Evolution. In *2008 15th Working Conference on Reverse Engineering*, pages 219–228, Antwerp, Belgium, October 2008. IEEE.
- [109] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM symposium on Software visualization - SoftVis '08*, page 155, Ammersee, Germany, 2008. ACM Press.
- [110] Rodrigo Souza, Bruno Silva, Thiago Mendes, and Manoel Mendonça. SkyscrapAR: An Augmented Reality Visualization for Software Evolution. 2012.
- [111] Frank Steinbrückner and Claus Lewerentz. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*, page 193, Salt Lake City, Utah, USA, 2010. ACM Press.
- [112] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, April 2013.

- [113] Jan Waller, Christian Wulf, Florian Fittkau, Philipp Döhring, and Wilhelm Hasselbring. SynchronoVis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. page 4, 2013.
- [114] Bonita Sharif, Grace Jetty, Jairo Aponte, and Esteban Parra. An empirical study assessing the effect of seeing 3D on comprehension. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–10, Eindhoven, Netherlands, September 2013. IEEE.
- [115] H Würfel, M Trapp, D Limberger, and J Döllner. Natural Phenomena as Metaphors for Visualization of Trend Data in Interactive Software Maps. page 8, 2015.
- [116] Teseo Schneider, Yuriy Tymchuk, Ronie Salgado, and Alexandre Bergel. CuboidMatrix: Exploring Dynamic Structural Connections in Software Components Using Space-Time Cube. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 116–125, Raleigh, NC, USA, October 2016. IEEE.
- [117] Yuriy Tymchuk, Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Walls, Pillars and Beams: A 3D Decomposition of Quality Anomalies. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 126–135, Raleigh, NC, USA, October 2016. IEEE.
- [118] Andreas Schreiber and Marlene Bruggemann. Interactive Visualization of Software Components with Virtual Reality Headsets. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 119–123, Shanghai, September 2017. IEEE.
- [119] Pooya Khaloo, Mehran Maghoumi, Eugene Taranta II, David Bettner, and Joseph Laviola Jr. Code Park: A New 3D Code Visualization Tool. *arXiv:1708.02174 [cs]*, August 2017. arXiv: 1708.02174.
- [120] Rodrigo Brito, Aline Brito, Gleison Brito, and Marco Tulio Valente. GoCity: Code City for Go. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 649–653, Hangzhou, China, February 2019. IEEE.
- [121] Akihiro Hori, Masumi Kawakami, and Makoto Ichii. Codehouse: Vr code visualization tool. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 83–87. IEEE, 2019.

-
- [122] Leonel Merino, Mario Hess, Alexandre Bergel, Oscar Nierstrasz, and Daniel Weiskopf. PerfVis: Pervasive Visualization in Immersive Augmented Reality for Performance Awareness. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 13–16, March 2019. arXiv:1904.06399 [cs].
- [123] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and Richard Müller. *GETAVIZ: Generating Structural, Behavioral, and Evolutionary Views of Software Systems for Empirical Evaluation*. September 2017.
- [124] David Baum, Pascal Kovacs, and Richard Müller. Fostering Collaboration of Academia and Industry by Open Source Software. 2020. ISBN: 9783885796947 Publisher: Gesellschaft für Informatik e.V.
- [125] Florian Jung, Veronika Dashuber, and Michael Philippsen. Towards Collaborative and Dynamic Software Visualization in VR:. In *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, pages 149–156, Valletta, Malta, 2020. SCITEPRESS - Science and Technology Publications.
- [126] Rainer Koschke and Marcel Steinbeck. SEE Your Clones With Your Teammates. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pages 15–21, Luxembourg, October 2021. IEEE.
- [127] Rainer Koschke and Marcel Steinbeck. Modeling, Visualizing, and Checking Software Architectures Collaboratively in Shared Virtual Worlds. 2021.
- [128] Falko Galperin, Rainer Koschke, and Marcel Steinbeck. *Visualizing Code Smells: Tables or Code Cities? A Controlled Experiment*. October 2022. Pages: 62.
- [129] Federico Pfahler, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing Evolving Software Cities. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 22–26, Adelaide, Australia, September 2020. IEEE.
- [130] Susanna Ardigo, Csaba Nagy, Roberto Minelli, and Michele Lanza. Visualizing Data in Software Cities. page 5, 2021.

-
- [131] Susanna Ardigò, Csaba Nagy, Roberto Minelli, and Michele Lanza. M3triCity: Visualizing Evolving Software & Data Cities. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 130–133, May 2022. arXiv:2204.10006 [cs].
- [132] David Moreno-Lumbreras, Jesus M Gonzalez-Barahona, and Andrea Villaverde. BabiaXR: Virtual Reality software data visualizations for the Web. 2021.
- [133] David Moreno-Lumbreras, Jesus M. Gonzalez-Barahona, and Gregorio Robles. BabiaXR: Facilitating experiments about XR data visualization. *SoftwareX*, 24:101587, December 2023.
- [134] David Moreno-Lumbreras, Jesus M Gonzalez-Barahona, and Gregorio Robles. Babiaxr: Facilitating Experiments About Xr Data. *Available at SSRN 4333820*.
- [135] Roy Oberhauser. VR-GitCity: Immersively Visualizing Git Repository Evolution Using a City Metaphor in Virtual Reality. 2023.
- [136] Dussan Freire-Pozo, Kevin Cespedes-Arancibia, Leonel Merino, Alison Fernandez-Blanco, Andres Neyem, and Juan Pablo Sandoval Alcocer. DGT-AVisualizing Code Dependencies in AR. In *2023 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023.
- [137] Martin Misiak, Doreen Seider, Sascha Zur, Arnulph Fuhrmann, and Andreas Schreiber. Immersive Exploration of OSGi-Based Software Systems in Virtual Reality. In *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 1–2, Reutlingen, March 2018. IEEE.
- [138] Martin Misiak, Andreas Schreiber, Arnulph Fuhrmann, Sascha Zur, Doreen Seider, and Lisa Nafeie. IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 112–116, Madrid, September 2018. IEEE.
- [139] Andreas Schreiber, Lisa Nafeie, Artur Baranowski, Peter Seipel, and Martin Misiak. Visualization of Software Architectures in Virtual Reality and Augmented Reality. In *2019 IEEE Aerospace Conference*, pages 1–12, Big Sky, MT, USA, March 2019. IEEE.

-
- [140] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, Eindhoven, Netherlands, September 2013. IEEE.
 - [141] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Exploring software cities in virtual reality. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 130–134, Bremen, Germany, September 2015. IEEE.
 - [142] Wilhelm Hasselbring, Alexander Krause, and Christian Zirkelbach. ExplorViz: Research on software visualization, comprehension and collaboration. *Software Impacts*, 6:100034, 2020. Publisher: Elsevier.
 - [143] Alexander Krause-Glau, Malte Hansen, and Wilhelm Hasselbring. Collaborative program comprehension via software visualization in extended reality. *Information and Software Technology*, 151:107007, November 2022.
 - [144] Alexander Krause-Glau, Marcel Bader, and Wilhelm Hasselbring. Collaborative Software Visualization for Program Comprehension. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 75–86, Limassol, Cyprus, October 2022. IEEE.
 - [145] Alexander Krause-Glau and Wilhelm Hasselbring. Scalable Collaborative Software Visualization as a Service: Short Industry and Experience Paper. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 182–187, CA, USA, September 2022. IEEE.
 - [146] Alexander Krause-Glau and Wilhelm Hasselbring. Collaborative, Code-Proximal Dynamic Software Visualization within Code Editors, August 2023. arXiv:2308.15785 [cs].
 - [147] Adrian Hoff, Michael Nieke, and Christoph 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*, pages 1455–1458, Athens Greece, August 2021. ACM.

- [148] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 119–130, Limassol, Cyprus, October 2022. IEEE.
- [149] Adrian Hoff, Christoph Seidl, Mircea Lungu, and Michele Lanza. Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2023.
- [150] Adrian Hoff, Christoph Seidl, and Michele Lanza. Immersive Software Archaeology: Exploring Software Architecture and Design in Virtual Reality. In *International Conference on Software Analysis, Evolution and Reengineering (SANER) 2024*. IEEE, 2024.
- [151] Adrian Hoff, Mircea Lungu, Christoph Seidl, and Michele Lanza. Collaborative Software Exploration with Multimedia Note Taking in Virtual Reality. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC24)*, 2024.
- [152] Adrian Hoff, Mircea Lungu, Christoph Seidl, and Michele Lanza. Immersive Software Archaeology: Collaborative Exploration and Note Taking in Virtual Reality. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC24)*, 2024.
- [153] A.s.K. Wijayawardena, Ruvan Abeysekera, and M.W.P Maduranga. A Systematic Review of 3D Metaphoric Information Visualization. *International Journal of Modern Education and Computer Science*, 15:73–93, February 2023.
- [154] Sheelagh Carpendale and Yaser Ghanam. A Survey Paper on Software Architecture Visualization. page 12, 2008.
- [155] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and Evolution of Software Architectures. page 18 pages, 2012. Artwork Size: 18 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.
- [156] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väättäjä. Software visualization today: systematic literature review. In *Proceedings of the 20th International Academic*

- Mindtrek Conference*, pages 262–271, Tampere Finland, October 2016. ACM.
- [157] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, August 2020.
- [158] Michael L Nelson. A Survey of Reverse Engineering and Program Comprehension. page 8, 2005.
- [159] Harry Sneed and Chris Verhoef. Re-implementing a legacy system. *Journal of Systems and Software*, 155:162–184, September 2019.
- [160] Lifeng Mu, Vijayan Sugumaran, and Fangyuan Wang. A Hybrid Genetic Algorithm for Software Architecture Re-Modularization. *Information Systems Frontiers*, 22(5):1133–1161, October 2020.
- [161] Jiahua Li and Ali Yamini. Clustering-based software modularisation models for resource management in enterprise systems. *Enterprise Information Systems*, pages 1–21, October 2020.
- [162] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 69–78, Florence, Italy, May 2015. IEEE.
- [163] Chun Yong Chong, Sai Peck Lee, and Teck Chaw Ling. Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach. *Information and Software Technology*, 55(11):1994–2012, November 2013.
- [164] Rudy P. Darken and John L. Sibert. A toolset for navigation in virtual environments. In *Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 157–165, Atlanta Georgia USA, December 1993. ACM.
- [165] Rob Ingram and Steve Benford. Legibility Enhancement for Information Visualisation. April 2002.

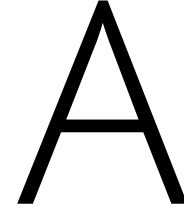
- [166] Norman G. Vinson. Design guidelines for landmarks to support navigation in virtual environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems the CHI is the limit - CHI '99*, pages 278–285, Pittsburgh, Pennsylvania, United States, 1999. ACM Press.
- [167] J. Leigh, A.E. Johnson, M. Brown, D.J. Sandin, and T.A. DeFanti. Visualization in teleimmersive environments. *Computer*, 32(12):66–73, December 1999.
- [168] Ruiqi Zhang. Research on the Progress of VR in Game. *Highlights in Science, Engineering and Technology*, 39:103–110, 2023.
- [169] Marco D'Ambros and Michele Lanza. Distributed and Collaborative Software Evolution Analysis with Churrasco. *Science of Computer Programming*, 75(4):276–287, April 2010.
- [170] Marco D'Ambros and Michele Lanza. A Flexible Framework to Support Collaborative Software Evolution Analysis. In *2008 12th European Conference on Software Maintenance and Reengineering*, pages 3–12, Athens, Greece, April 2008. IEEE.
- [171] Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. Source-Vis: Collaborative software visualization for co-located environments. In *2013 First IEEE Working Conference on Software Visualization (VIS-SOFT)*, pages 1–10, Eindhoven, Netherlands, September 2013. IEEE.
- [172] Andrew J. Ko, Thomas D. LaToza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, February 2015.
- [173] Li Zhang, Jia-Hao Tian, Jing Jiang, Yi-Jun Liu, Meng-Yuan Pu, and Tao Yue. Empirical Research in Software Engineering A Literature Survey. *Journal of Computer Science and Technology*, 33(5):876–899, September 2018.
- [174] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009.
- [175] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, Boulder, CO, USA, 1996. IEEE Comput. Soc. Press.

- [176] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.

Part II



Papers



Towards Immersive Software Archaeology: Regaining Legacy Systems' Design Knowledge via Interactive Exploration in Virtual Reality

Originally published in: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), August 19–28, Athens, Greece

Joint work with: Michael Nieke and Christoph Seidl

Abstract

Many of today's software systems will become the legacy systems of tomorrow, comprised of outdated technology and inaccurate design documents. Preparing for their eventual re-engineering requires engineers to regain lost design knowledge and discover re-engineering opportunities. While tools and visualizations exist, comprehending an unfamiliar code base remains challenging. Hence, software archaeology suffers from a considerable entry barrier as it requires expert knowledge, significant diligence, tenacity, and stamina. In this paper, we propose a paradigm shift in how legacy systems' design knowledge can be regained by presenting our vision for an immersive explorable software visualization in virtual reality (VR).

We propose innovative concepts leveraging benefits of VR for a) immersion in an exoteric visualization metaphor, b) effective navigation

and orientation, c) guiding exploration, and d) maintaining a link to the implementation. By enabling immersive and playful legacy system exploration, we strive for lowering the entry barrier, fostering long-term engagement, strengthening mental-model building, and improving knowledge retention in an effort to ease coping with the increased number of tomorrow's legacy systems.

Video Demonstration – <https://youtu.be/Ma83YbQ6ck0>

A.1 Introduction

Many of today's large-scale productive systems will become tomorrow's legacy systems when companies do not adapt to advances in technology, design knowledge is rendered inaccurate over time, or original developers move on to other projects/companies [1]. While the ongoing use of a legacy system indicates a continued value for its company, coping with new requirements (e.g., to accommodate for increased load or changed legislation) may ultimately require substantial re-engineering. To scope that re-engineering, it is then necessary to regain design knowledge on a legacy system's structure and behavior [2, 3], and to identify re-engineering opportunities, i.e., a potential for quality improvement. For many legacy systems, the only reliable source for regaining design knowledge commonly consists of the system's implementation and its runtime behavior exposed in current use [1, 3]. While tools for analyzing and visualizing a software system's design exist [4, 5], exploring an unfamiliar codebase still requires significant effort, motivation, and diligence, which, at present, makes regaining legacy systems' design knowledge a challenging and tedious procedure.

In this paper, we propose a paradigm shift in how legacy systems' design knowledge can be regained by presenting a vision for an immersive and interactive software visualization in virtual reality (VR) generated from a system's implementation, execution traces, and quality metrics that allows for engaging and playful legacy system exploration. In particular, we aim to capitalize on the benefits of VR by striving to achieve the following objectives:

- Lowering the entry barrier to exploring legacy systems
- Fostering long-term engagement via a drive for exploration
- Strengthening users' mental model of a system's design, behavior, and quality

- Improving knowledge retention even over periods of absence

In the following, we present our vision and future core contributions for what we call *immersive software archaeology*.

A.2 State of the Art

Three principal approaches exist for regaining a software system's design knowledge, i.e., (i) manually browsing through code and models, (ii) guided by knowledgeable programmers who know the subject system, and (iii) computer-aided techniques [2]. As browsing through a large-scale system strictly manually is not feasible and experienced programmers are often not available, computer-aided techniques in form of (semi) automated tools are the only realistically applicable approach. Generally, these fulfill two consecutive tasks. First, they analyze certain aspects of a subject system, e.g., subsystem decomposition, dependency analysis, or metrics calculation [2]. Second, they display respective results as text (tables, lists) or visual 2D/3D representations.

Depending on the purpose of a tool, visual representations can significantly improve the legibility of its output by addressing users' visual cognitive capabilities [6]. Software visualization aids software comprehension by leveraging this benefit and displaying software systems in form of visual representations in 2D (e.g., graphs, diagrams) or 3D space. Doing the latter requires a metaphor that maps the intangible concepts of software (e.g., classes) to visually meaningful objects. These can be subdivided into abstract (e.g., graph or tree-like) and real-world (e.g., cities, islands, solar systems) metaphors [4]. With recent advances in VR, research gained an increased interest in leveraging the technology to improve software comprehension. Studies suggest that VR can help with software comprehension tasks [7] and knowledge retention [8].

We identify shortcomings of existing 3D software visualization methods that we aim to address: A majority of methods rigidly visualize all artifacts of a system at once [4, 5]. This can result in overwhelmingly complex 3D structures (e.g., large 3D cities) and, thus, impaired legibility and comprehensibility. Additionally, a majority of existing concepts use 3D metaphors solely to represent selected fundamental metrics of a system's artifacts (e.g., lines of code for the size of a 3D structure) where an interpretation in terms of quality requires significant software engineering expertise. That constitutes an entry barrier, which is raised further by overloaded visual metaphors and

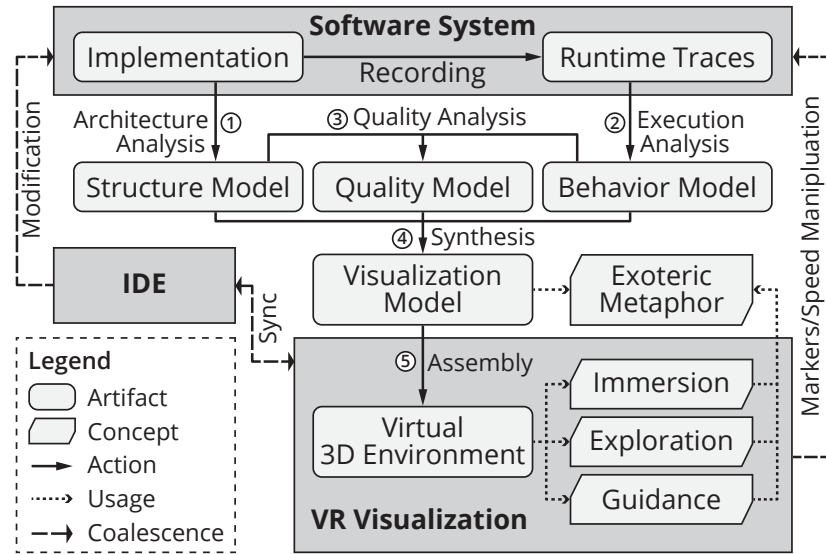


Figure A.1: Overview of our envisioned method. An automated process analyzes a subject system and yields a 3D environment that is explored in VR. The visualization is synchronized with an IDE.

missing explanations. Regarding behavior, existing execution trace visualization approaches focus either on scalability with large systems by providing lucid, yet limited, information or on providing detailed information, which does not scale with large systems. Generally, we observe that most existing methods are constructed to ease the comprehension of only one aspect of a system, i.e., either structure, behavior, or quality. At the same time, most existing tools serve as additional exploration means that depend on simultaneously browsing source code in an IDE. Regarding the potential benefits of immersive capacities offered by modern VR technology, we conclude that current VR methods have yet to overcome the stage of essentially porting existing standard-screen 3D visualization methods into a VR environment.

A.3 Immersive Software Archaeology

To provide a framework for our envisioned contributions, we briefly elaborate on the model-driven process for creating core artifacts of our method as depicted in Figure A.1: Our method will analyze ① a legacy system's implementation for architectural recovery yielding information on hierarchically nested sub-systems, components, etc. and their interrelations, e.g., as mentioned in [9]. Our method will collect behavior information by recording and analyzing ② method-call traces passing through that structure. Our method will

further collect quality information ③ by analyzing the models yielded from ① and ② with regards to technical debt on an architectural level, e.g., by conducting dependency analyses and identifying execution bottlenecks. Using a visual metaphor, our method will synthesize ④ these results into a visualization model, i.e., a data model that contains abstract information on the spatial arrangement and appearance of visual elements that meaningfully represent a subject system. Based on that model, we will then assemble a virtual 3D environment ⑤ to use within our VR visualization.

The models of this process will allow for the representation of object-oriented languages by successively abstracting from specific target programming language features. We will demonstrate our method and its information retrieval mechanisms on the case of legacy systems implemented in Java, e.g., by instrumenting the Java Virtual Machine to transparently record execution traces. However, we design our concepts generalizable for imperative languages that allow deriving data on encapsulation, modularity, and coupling.

To achieve our vision of *immersive software archaeology*, we will draw on existing work on software re-engineering [2, 3], software comprehension [10], software architecture recovery [9, 11], and software visualization [5, 4]. We will devise innovative concepts that immerse re-engineers into a VR representation of a subject system (Section A.3.1), provide means for effective navigation and orientation (Section A.3.2), guide users through understanding both our method and the subject system (Section A.3.3), while maintaining a link to the actual source code (Section A.3.4). The synthesis provided by these concepts will constitute an integrated, interactive, and immersive visualization of a software system's structure, behavior, and quality.

A.3.1 Immersion: Experiencing the System

Recent research suggests that examining a software system is perceived as more satisfactory when using a 3D visualization instead of a text-based IDE [12]. We will extend that idea by exploiting the immersive properties of VR to foster a drive for exploration.

Exoteric Metaphor with Esoteric Properties We will support users' mental-model building by relating to real-world knowledge. Therefore, we will design an exoteric (common knowledge) metaphor to represent a software system's structure, behavior, and quality on various levels of granularity as elements from the physical world. For structural aspects, a component could

be visualized as a planet in a solar system with its sub-components as cities on that planet and classes as buildings populating the respective city, where dimensions and placement indicate artifact size and logical relation. For instance, class buildings could be composed of sub-complexes for each method, shaped according to method properties such as the number of splits in their control flow. For behavioral aspects, the control flow of recorded execution traces could be visualized as abstract inhabitants traveling through the representation along the involved software components. For qualitative aspects, metric values could be conveyed as esoteric properties of our representation (common elements with specialized meaning), e.g., communicating a high degree of coupling via visually complex textures or the presence of architectural smells via litter in the environment.

Unlike with the majority of existing software visualizations, users will not only take the role of an overseeing observer but will also explore the represented structures in first-person view. When roaming freely, users may discover detailed phenomena, e.g., congregations of numerous “inhabitants” indicating heavily frequented software components. When guided along specific paths (cf. Section A.3.3), users can explore common system usage or manually created routes deemed particularly relevant.

Ambient Visuals and Acoustics To subliminally communicate behavior and quality information, our method will incorporate ambient effects based on textures, animations, and sound: We will use ambient visuals to highlight particularly frequented routes through the system (e.g., by worn-out paths), which creates a focal point for exploration. When replayed, execution traces may emit 3D positional sounds to indicate location, frequency, and clusters of events. We will also explore ambient music to convey a subliminal impression of software quality for a visited part of the software system, e.g., by using tense music to accentuate bad software quality.

Diegetic Interfaces We will offer various tools to inspect and interact with the represented software system (cf. Section A.3.2 and Section A.3.4). Instead of breaking with immersion by a mere overlay interface (e.g., screen-space 2D buttons and menus), we will design access to all context-dependent tools as diegetic interfaces that integrate with the metaphor and the 3D representation [13]. Using the above metaphor, users could be provided with an overview of their current location within a subject system via a holographic projection originating from a device attached to their virtual arm. To compare software elements (e.g., classes) with each other, users will be able to

temporarily detach them from their actual environment and transport them to a dedicated comparison environment. Furthermore, users will be able to inspect a system's live behavior by interacting with it directly, e.g., by accessing the public interface of a class to input data and observing how the system reacts.

We will counteract stimulus overflow by offering users to configure what information they are interested in and filtering the VR environment accordingly. For instance, users that are solely interested in behavior can decide to filter out information on quality. We will create predefined profiles for different exploration purposes.

A.3.2 Exploration: Navigation and Orientation

Exploiting VR's capability of immersing users into a virtual environment requires providing means for effective and efficient navigation. For that purpose, our method will feature navigation along structure, behavior, and quality.

Structural Semantic Zoom A Global overview is considered crucial for 3D information visualization [5]. As a starting point, our method will show a subject system on its architectural level. Structure and quality will be visualized on component level, between which recorded traces visualize behavior. As part of our visual metaphor, we envision a step-wise semantic zoom that enables navigating along a system's structure to reveal further details. For instance, zooming into a component will visualize its inner structure, behavior, and quality, i.e., of contained sub-components or classes. A key aspect here is to maintain users' orientation, e.g., with an interface showing the current position in the overall system.

As an example, using the metaphor given in Section A.3.1, a system could be initially visualized on the level of planets (components). Users could be able to zoom into a planet that is then enlarged (optically) and augmented with further semantic details, e.g., abstract representations of its cities (sub-components). Zooming into a city could be realized by letting users walk among its individual buildings (classes). To the best of our knowledge, our envisioned software visualization method is the first to provide such zooming/abstraction functionality based on functional architectural components integrated into a real-world metaphor.

Temporal Behavioral Zoom Cross-cutting to the semantic structural zoom, we envision a temporal zoom that enables users to manipulate the time in which recorded execution traces are represented. Users will be able to speed up, slow down, and freeze time as well as to jump back and forth in time. Extending the idea of time travel debugging, this mechanism will leverage both inspecting short-lived phenomena and surveying lengthy procedures.

Context-Dependent Quality Zoom We will make quality information available on demand by letting users select types of technical debt to be visualized in the currently visited part of a system. Respective information will be represented as exoteric property with esoteric meaning, e.g., a dirty floor within a building as an indicator for a code smell within a class.

A.3.3 Guidance: Fostering Understanding

Our method will progressively self-explain its usage via built-in tutorials. In parallel, it will adaptively guide users through a subject system, according to specific interests, e.g., in its behavior.

Understanding the Method Unlike a majority of existing approaches, we will ease the entry into our method by generating tutorial-style quests from the actual system that incrementally introduce tools and diegetic interfaces, e.g., “locate the class x ” or “find the metric value for y ”. We will maintain users’ immersion by embedding these tutorials into our metaphor, triggering them according to what the user intends to do, and enabling to skip or entirely disable the tutorials. Additionally, if a user inspects the visual representation of a software entity (such as a class), our method will provide hints on how to “read” it, e.g., via a helping companion.

Understanding a System We will help users with understanding a subject system more efficiently by using visual effects and audio to create observable phenomena that draw attention to potentially interesting areas. For instance, we will encode information on the quality of components and classes via the texture of their VR representations. Using the metaphor given in Section A.3.1, a class with an overall high complexity could be represented by a building with a visually complex texture. To guide users concerning behavioral quality (esp. on high abstraction levels), we will offer to augment execution trace routes with visual aids that encode how highly these paths are frequented, e.g., using a heat map color scheme ranging from dark blue to bright red. Our method

will explain the meaning of these phenomena to users, e.g., via a helping companion. Furthermore, we will generate quests to explore the system, e.g., to travel along the most common execution route.

A.3.4 Coalescence: Mental & Technical Back-Link

We will foster a link between a user's mental model formed in our visual metaphor and the underlying implementation of a subject system. Unlike existing software visualizations, we will therefore coalesce and synchronize our visualization with an IDE, e.g., by extending Microsoft's language server protocol¹ to place and query markers for system parts of interest.

Details in VR Our method will enable inspecting implementation details directly from within the VR visualization. For instance, using the metaphor given in Section A.3.1, that could be achieved by providing users with a diegetic interface (e.g., a virtual tablet computer) which, once attached to a class (building), displays its source code. Upon performing changes to artifacts within the IDE, we will update the VR visualization accordingly.

Annotations (POIs) Users will be able to annotate points of interest (POI) throughout the VR environment. These will conserve gained insights in form of user-created text messages and sound recordings, which are related to structures represented within the VR representation. Using the metaphor given in Section A.3.1, POIs could be realized in form of sticky notes that users pin to the facade of a building (class) or, on a higher level of semantic zoom (cf. Section A.3.2), to a planet (component). POIs will be synchronized with the IDE so that markers can be created, inspected, modified, and deleted with the respective software elements.

Record & Replay We will devise a mechanism that allows users to record sequences of previously marked POIs so that they can use contained messages and voice recordings to construct paths through the system. Encoding experiences in that way will allow for building an asynchronous mentor-mentee relationship between collaborators and, thus, ease the exploration of a subject system. We will synchronize recorded paths with the IDE so that they can be recorded and navigated from within both VR and the IDE.

¹<https://langserver.org/>

A.4 Evaluation

We are in the process of refining our ideas and visions into concrete concepts and a prototype. We will use that prototype to empirically evaluate our concepts in a series of controlled experiments which, in turn, will influence further development of our concepts and prototype. As subject systems, we will use multiple real-world legacy systems, e.g., a long-running, undocumented medical supply system provided by one of our industry partners. To consider diverse backgrounds, we will recruit participants with knowledge in software engineering from various countries, already having arranged for participants from Denmark, Germany, and Switzerland. We will divide participants into a treatment group using our immersive VR method as well as two control groups using a) an existing standard-screen 3D visualization tool and b) a standard IDE along with dedicated tools for architecture recovery, quality metric calculation, etc. After a brief training period on either of the tools, we will use two consecutive experiment sessions that are interleaved with two weeks of absence from the subject system. In the first session, we will evaluate our objectives of lowering the entry barrier, maintaining long-term engagement, and easing mental model building by letting participants conduct a series of extensive tasks, e.g. finding certain system aspects or identifying performance bottlenecks. We will measure metrics such as completion time and correctness, collect participants' subjective verdict (e.g., similar to [14]), and observe their behavior. In the second session, we will evaluate participants' knowledge retention by asking questions about the system's structure, behavior, and quality, e.g., by asking them to recall its architecture from memory or to find the same system aspects as in the first session.

A.5 Conclusion and Future Work

With our envisioned method, we aim to reshape the comprehension phase of legacy software system re-engineering. Instead of being labor-intensive, tedious work, we will immerse re-engineers in an engaging, playful exploration that builds a strong mental model of a system while still connecting to its implementation. By making software archaeology less tedious and more accessible to a wider audience, we strive for our method to ease coping with an increased number of legacy systems of the future.

Bibliography

- [1] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How Do Professionals Perceive Legacy Systems and Software Modernization? In *Int. Conf. on Software Engineering (ICSE)*, 2014.
- [2] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse Engineering: A Roadmap. In *Conf. on the Future of Software Engineering*, 2000.
- [3] H. Sneed and C. Verhoef. Re-implementing a Legacy System. *Journal of Systems and Software*, 2019.
- [4] A.R. Teyseyre and M.R. Campo. An Overview of 3D Software Visualization. *Transactions on Visualization and Computer Graphics*, 2009.
- [5] P. Caserta and O Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 2011.
- [6] M. Card. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, 1999.
- [7] R. Wetzel, M. Lanza, and R. Robbes. Software Systems as Cities: A Controlled Experiment. In *Int. Conf. on Software Engineering (ICSE)*, 2011.
- [8] L. Merino, J. Fuchs, M. Blumenschein, C. Anslow, M. Ghafari, O. Nierstrasz, M. Behrisch, and D. A. Keim. On the Impact of the Medium in the Effectiveness of 3D Software Visualizations. In *Work. Conf. on Software Visualization (VISSOFT)*, 2017.
- [9] J. Garcia, I. Ivkovic, and N. Medvidovic. A Comparative Analysis of Software Architecture Recovery Techniques. In *Int. Conf. on Automated Software Engineering (ASE)*, 2013.
- [10] M. J. Pacione, M. Roper, and M. Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *Working Conf. on Reverse Engineering*, 2004.
- [11] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger. Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. In *37th IEEE Int. Conf. on Software Engineering*, 2015.

- [12] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza. On the use of Virtual Reality in Software Visualization: The Case of the City Metaphor. *Information and Software Technology*, 2019.
- [13] P. Salomoni, C. Prandi, M. Rocchetti, L. Casanova, and L. Marchetti. Assessing the Efficacy of a Diegetic Game Interface with Oculus Rift. In *Annual Consumer Communications & Networking Conference (CCNC)*, 2016.
- [14] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. CityVR: Gameful Software Visualization. In *2017 IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, 2017.

Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality

Originally published in: Proceedings of the 10th IEEE Working Conference on Software Visualization (VISSOFT 2022), October 2–3, Limassol, Cyprus

Joint work with: Lea Gerling and Christoph Seidl

Abstract

Exploring an unfamiliar large-scale software system is challenging, especially when based solely on source code. While software visualizations help in gaining an overview of a system, they generally neglect architecture knowledge in their representations, e.g., by arranging elements along package structures rather than functional components or locking users in a specific abstraction only slightly above the source code.

In this paper, we introduce an automated approach for software architecture recovery and use its results in an immersive 3D virtual reality software visualization to aid accessing and relating architecture knowledge. We further provide a semantic zoom that allows a user to access and relate information both horizontally on the same abstraction level, e.g., by following method calls, and vertically across different abstraction levels, e.g., from a class to its containing component. We evaluate our contribution in a controlled experiment contrasting the use-

fulness regarding software exploration and comprehension of our concepts with those of the established CityVR visualization and the Eclipse IDE.

Video Demonstration – <https://youtu.be/wmayYcpL7ZY>

B.1 Introduction

For software engineers, establishing an understanding of a large-scale software system is essential for starting work on a settled project and regaining design knowledge of a legacy system [1, 2]. The exploration of a software system ideally starts with the system’s architecture [3] to gain both an overview of as well as guidance through the system’s coarse-grained structure. However, architecture documentation may be inaccurate even for established projects or outright missing for legacy systems [4, 2], leaving a system’s source code as the only reliable information. Establishing a mental model of a system’s structure from source code alone is tedious and challenging due to large amounts of fine-grained detail and a lack of explicitly represented coarse-grained architectural concerns. While integrated development environments (IDEs) and dedicated analysis tools may foster an inspection and navigation of source code, there are few applied visualization techniques for architectural analysis and synthesis activities [5].

Various forms of software visualization in 2D, 3D, augmented reality (AR), and virtual reality (VR) visually represent coarse-grained structures of a software system to provide an overview and highlight particular phenomena, such as especially large classes. Visualizations in VR seem promising as recent research indicates that they provide for more engaging exploration than both IDEs and standard-screen visualizations [6, 7, 8]. Many visualization techniques use elementary software architecture information in their representation: For example, metric values influence the depiction of individual elements, or the package structure defines the arrangement of elements. Although a system’s internal organization of implementation artifacts can deviate heavily from its actual architecture, especially when the system underwent long-term evolution and, as a side-effect, experienced architectural erosion [9], existing visualization techniques generally do not consider information from advanced architecture recovery, like conceptual components, their dependencies or control flows, and thereby leave a crucial source of information untapped.

In this paper, we present a method for utilizing software architecture recovery to visualize and utilize a system’s architecture as a first-level entity. Our

method allows users to access and navigate information along different abstraction levels via a semantic zoom, from architectural component hierarchies down to classes and methods. On each abstraction level, our method additionally provides users with a visualization of relationships among elements, such as dependencies among components, or calls among methods, which enables users to efficiently retrace and navigate along relationships.

We demonstrate our method via an implementation for immersive VR and evaluate it in an empirical experiment with 54 participants in which we compare its ability to foster accessing and relating information on multiple levels of abstraction with a standard IDE and another state-of-the-art software visualization. Our results show that, compared to the IDE and the state of the art, our approach provides participants with a better overview of a subject system's architecture, while improving their ability to access and relate elements.

The rest of this paper is structured as follows: In Section B.2, we discuss the state of the art in software visualization regarding (its lack of) incorporating software architecture knowledge. In Section B.3, we describe our method for software architecture recovery (SAR) and how we incorporate its results into an immersive 3D virtual reality representation. In Section B.4, we evaluate our contribution in a controlled experiment contrasting its usefulness regarding software exploration and comprehension with those of the established CityVR visualization and the Eclipse IDE. Finally, in Section B.5, we close with a conclusion and an outlook on future work.

B.2 State of the Art

A software visualization provides a visual overview of a subject system [10]. Depending on the purpose of the visualization, it may encompass a system's structure, behavior, evolution, or quality [11, 12]. A visualization uses a metaphor to depict (otherwise non-corporeal) elements of a software system in a coherent setting. While 2D metaphors are mostly abstract, such as graph or tree representations, 3D metaphors may range from abstract to real-world representations, such as cities, planets, or islands [12, 13]. Despite a plethora of different software visualizations, we identify shortcomings regarding their use of architectural knowledge:

Overview of Software Architecture Existing 3D software visualizations do not sufficiently use architecture information as a driving first-level element of their visual structure. Instead, the term “software architecture” is often used

interchangeably with a system's internal organization of implementation artifacts. In consequence, visualization elements are structured according to folder structures, namespaces, or package hierarchies, which, while indicative of a system's design, do not adequately represent a system's architecture, e.g., in terms of its functional components and their connections. This gap also manifests in various surveys on 3D software visualization, where a subject system's architecture beyond folders, namespaces, and packages is not among the explicitly extracted aspects that existing 3D software visualizations address [14, 15, 16, 10, 13, 11].

Accessing Architecture on Various Abstraction Levels Existing 3D software visualizations fixate their view on a system on one abstraction level, usually on the level of files, classes, or methods, where a prime aspect is the visualization of metrics such as lines of code. For the widely used city metaphor [7, 17, 18, 19, 20, 21, 22, 6, 23], this manifests in complex large-scale cities where architectural information is mainly used to determine positions for a large number of buildings, while lower level visual structure is often not available. As a result, this leaves open potential for guiding engineers along the abstraction levels of a subject system's structure altogether.

Relating Architecture Elements There exist only few 3D software visualization approaches that both incorporate a system's architecture while allowing to switch between abstraction levels. Most notably, Balzer et al. [24, 25, 26] use a metaphor of hierarchically nested semi-transparent bubbles, starting on architectural level. Based on that, they establish a semantic zoom that enriches elements with more fine-grained information when moving the virtual camera closer. However, while this can strengthen users' overview on architecture level, including their ability to relate elements, it does not provide them with this overview once zoomed in on a fine-grained level, which has an impact on viewers' orientation and their ability to retrace relations between elements on architecture level.

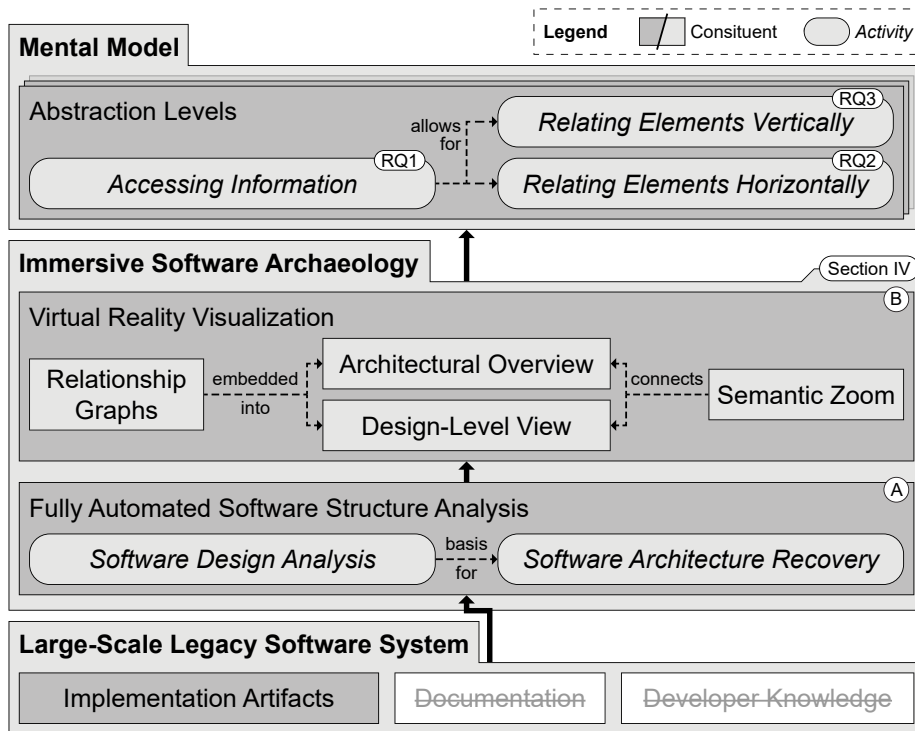


Figure B.1: Overview of our method for utilizing software architecture recovery to provide a visualization with semantic zoom along multiple abstraction levels.

B.3 Immersive Software Archaeology

The goal of our software visualization method is to foster the exploration of an unfamiliar software system by aiding users with accessing and relating information on and across design and architecture level. Figure B.1 shows an overview of that. As a system's implementation artifacts are the only reliable source of information, our method conducts an automated software structure analysis based on only the source code of a system (box A in Figure B.1), yielding the ground-truth structure of a system's design as well as, based on that, an estimation of its higher-level architectural structure. With the design of a system, we refer to its implementation in terms of constructs such as classifiers (i.e., classes, interfaces, etc.) and their constituents – commonly explicit in source code through designated language constructs. With the architecture of a system, we refer to a hierarchical organization of its design-level structure in cohesive components – generally only implicit in source code. While our concepts are applicable for systems implemented in object-oriented programming languages in general, we demonstrate them on Java-based systems and terminology in this paper.

We visualize the structure resulting from our analysis via an immersive software visualization in VR (box ② in Figure B.1). To provide engineers' with access to information on all abstraction levels of the resulting structure, we establish a semantic zoom that lets users interactively switch between abstraction levels while always providing them with an overview. To furthermore foster engineers' ability to retrace relations between elements on different abstraction levels, our method incorporates an interactive visualization of relationship graphs along the semantic zoom. Thereby, we address both horizontal relations on the same abstraction level, such as dependencies among components or calls among methods, as well as vertical relations across different abstraction levels, i.e., containment relations, such as between a component and a classifier.

B.3.1 Automated Software Structure Analysis

Our method encompasses an automated analysis of a system's static structure (cf. ① in Figure B.1). This analysis consists of two subsequent steps which populate a software structure model. The first step is an analysis of the system's design on the basis of its source code (Section B.3.1). The second step is a software architecture recovery procedure based on the results of the recovered design of the system (Section B.3.1).

Software Design Analysis

In the first step of our software structure analysis, our method utilizes a parser to automatically extract design-level information explicitly available in the source code of a subject system. This process lifts information about all classifiers and members of a system into a model structure – an excerpt from our concrete metamodel is available in our online appendix¹ and in Figure B.7 in the appendix of this chapter. For members with statement bodies, the software design analysis gathers metrics such as their respective number of expressions and cognitive complexity [27]. Subsequently, the analysis extracts dependencies among classifiers and calls among members. The resulting model structure encompasses the ground-truth design-level structure of an entire system, including a classifier-level dependency graph and a member-level call graph.

¹<https://gitlab.com/immersive-software-archaeology/publication-vissoft22>

Software Architecture Recovery (SAR)

The second step during our software structure analysis is an SAR that establishes an architecture-level software structure model. For that purpose, we devise an unsupervised software clustering procedure that organizes a subject system's implementation artifacts in a hierarchy of cohesive functional components. Our procedure operates based on the results of the software design analysis described above. However, in contrast to the design analysis, an SAR procedure, automated or not, cannot draw on explicitly available information. Instead, it needs to recover implicit high-level connections between software elements and is therefore driven by heuristics and best guesses [28]. The method we present in this paper serves as an exemplary demonstration of an unsupervised SAR procedure. If a project's specifics call for a dedicated solution, our SAR procedure can be replaced with a suitable alternative procedure yielding a hierarchical organization of implementation artifacts.

A prime goal of our overall software visualization method is to provide engineers with an overview of a system's architecture along multiple levels of abstraction. Because a hierarchical structure supports engineers in their thought process [4], we design our SAR procedure to recover a system's architecture on multiple abstraction levels in form of nested hierarchies of components. To foster the discovery of correlations in the source code of a system, these components should be as cohesive as possible, i.e., they should group together what is strongly interrelated. Depending on the level of abstraction they capture, we distinguish between three kinds of components in our model structure. We define bottom-level components as components that contain classifiers directly but do not contain sub-components. For higher level components, we distinguish between top-level components and intermediate components. We define both as components that do not contain classifiers directly, but instead an arbitrary amount of sub-components. Top-level components are the root components in a component hierarchy, whereas intermediate components represent the abstraction levels in between top-level components and bottom-level components. Intermediate components can be nested, allowing for arbitrarily high hierarchical structures.

Another prime consideration for our SAR procedure is to split a subject system up into components that are small so that their detailed visualization does not overwhelm a viewer, yet large enough to result in component hierarchies as small and simple as possible. To achieve both, we employ a divisive hierarchical clustering technique that can be configured with a hard lower limit and a soft upper limit for cluster sizes. Figure B.2 depicts an example application of that technique. Initially, it groups all classifiers of a system in

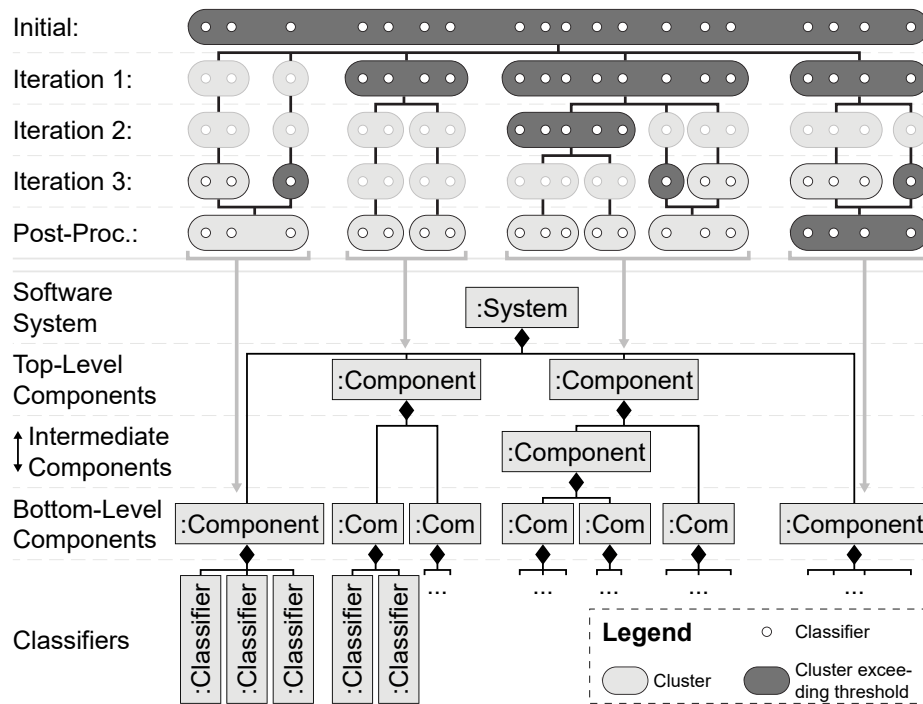


Figure B.2: Upper area: Simplified representation of our exemplary software clustering procedure. Similarities between classifiers are represented via spatial distances. Lower area: Representation of the respective clustering results when populated in a software structure model (member-level structure omitted). Please note how the cluster structure in the upper area is translated into the component structure in the lower area (indicated by light-grey arrows).

one root cluster (cf. first row in the upper area of Figure B.2). It then iteratively breaks down clusters that exceed the upper limit for cluster sizes. Rows 2 to 4 in the upper area of Figure B.2 illustrate these iterations in the given example (the upper limit in the example is set to 3, the lower limit is set to 2). As subroutine for the splits, we choose the DBSCAN algorithm (“Density-Based Spatial Clustering of Applications with Noise” [29]), because thereby (i) we can directly influence the upper limit for cluster sizes, (ii) we can detect noise, i.e., in our case, classifiers which are loosely coupled with the rest of the system and, therefore, need special treatment, and (iii) we calculate clusters purely based on the similarity between their containment, which, in our case, ensures cohesiveness among the clustered classifiers.

To achieve a high degree of cohesiveness within the clusters computed by our technique, we measure the similarity between clusters in terms of the summarized weight of their dependencies. These are calculated by aggregating the weight of all direct dependencies between their contained classifiers.

Once the dividing iterations of our technique are completed, the resulting hierarchy contains clusters that lay within the specified limits for cluster sizes as well as noise singleton clusters (cf. row 3 in the upper area of Figure B.2). As a last step, our technique therefore performs a post-processing procedure that merges noise singleton clusters with neighboring clusters. The bottom row in the upper area of Figure B.2 shows the result of that process in the given example. These clusters are then translated into a hierarchy of components accordingly, as exemplarily depicted in the lower area of Figure B.2 (note the arrows connecting clusters to component counterparts).

Cluster Labeling Finally, our SAR procedure labels components on all hierarchy levels with the most frequently occurring words in the names of their contained classifiers and members. This process starts with bottom-level components, iteratively working its way up. It extracts words based on typical naming conventions, e.g., “exampleName” results in labels “example” and “name”.

B.3.2 Immersive Virtual Reality Visualization

We present a 3D software visualization method that builds upon the software structure analysis presented in Section B.3.1 to visualize a subject system’s architecture and design on multiple levels of abstraction. It guides users along a system’s architectural structure via a semantic zoom, while fostering detail inspection via interactive visualizations of relationships among classifiers and components. This step is summarized in box ② in Figure B.1. In the following, we elaborate on these concepts, backed up by examples from our prototype implementation Immersive Software Archaeology (ISA).

Architectural Overview

To guide users’ exploration of a subject system along its architecture, we establish a real-world metaphor that provides users with an overview of the system’s architecture across multiple levels of abstraction. To achieve this, we visually represent the structures recovered by our software structure analysis in form of a solar system with planets, continents, cities, and buildings.

Figure B.3 conceptually depicts an example instance of our solar system metaphor along with its software structure model. Top-level components are represented as planets, bottom-level components are represented as cities with classifiers as buildings, where each city receives a piece of land to be

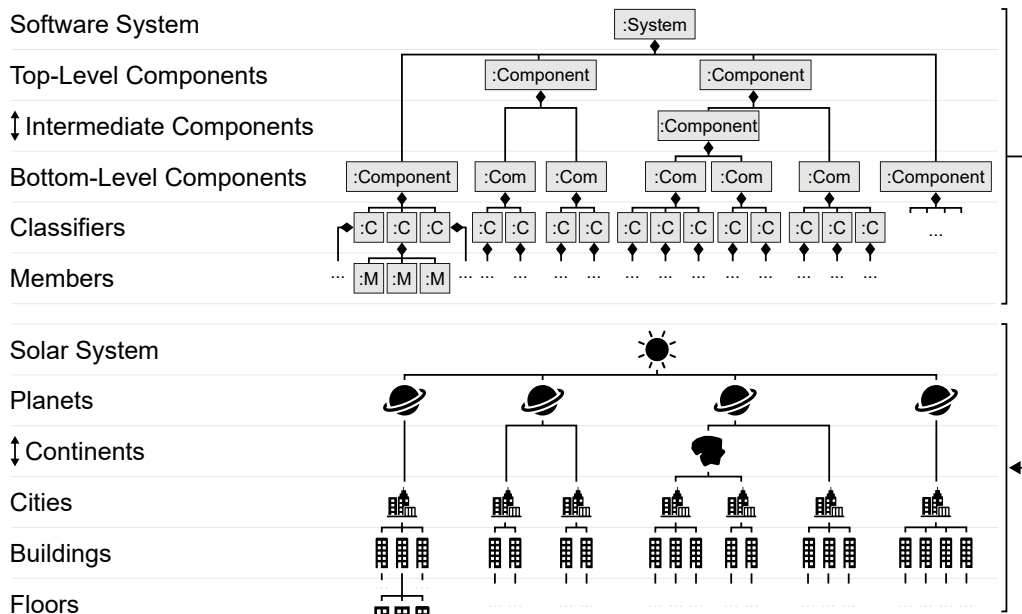


Figure B.3: Transformation of a software structure model to our solar system metaphor. The illustration continues the example given in Figure B.2.

located on. Intermediate components determine how cities are grouped together on a planet so that they form larger land masses, resulting in continents that are separated by water. Internally, these form hierarchies similar to real-world continents and their sub compositions in countries, regions, and so on, which allows representing even deep component nesting.

The three screenshots in the upper area of Figure B.4 depict our VR implementation of this architectural overview in the tool ISA. The left-hand side of the figure maps the semantic zoom levels to the primarily visualized constituents of our metaphor. Screenshots **a** and **b** show the overview on system level. Screenshot **c** shows a close-up view of the surface of a planet, where cities form continents according to the represented component hierarchy.

Semantic Zoom

When entering our visualization, a user is initially presented with the architectural overview of a subject system (cf. upper screenshots in Figure B.4). They can navigate through the architectural overview along its different levels of abstraction by freely inspecting elements. For instance, a user might inspect a system on planet level, find interest in a planet, and inspect its cities, similar to how Screenshot **c** depicts it.

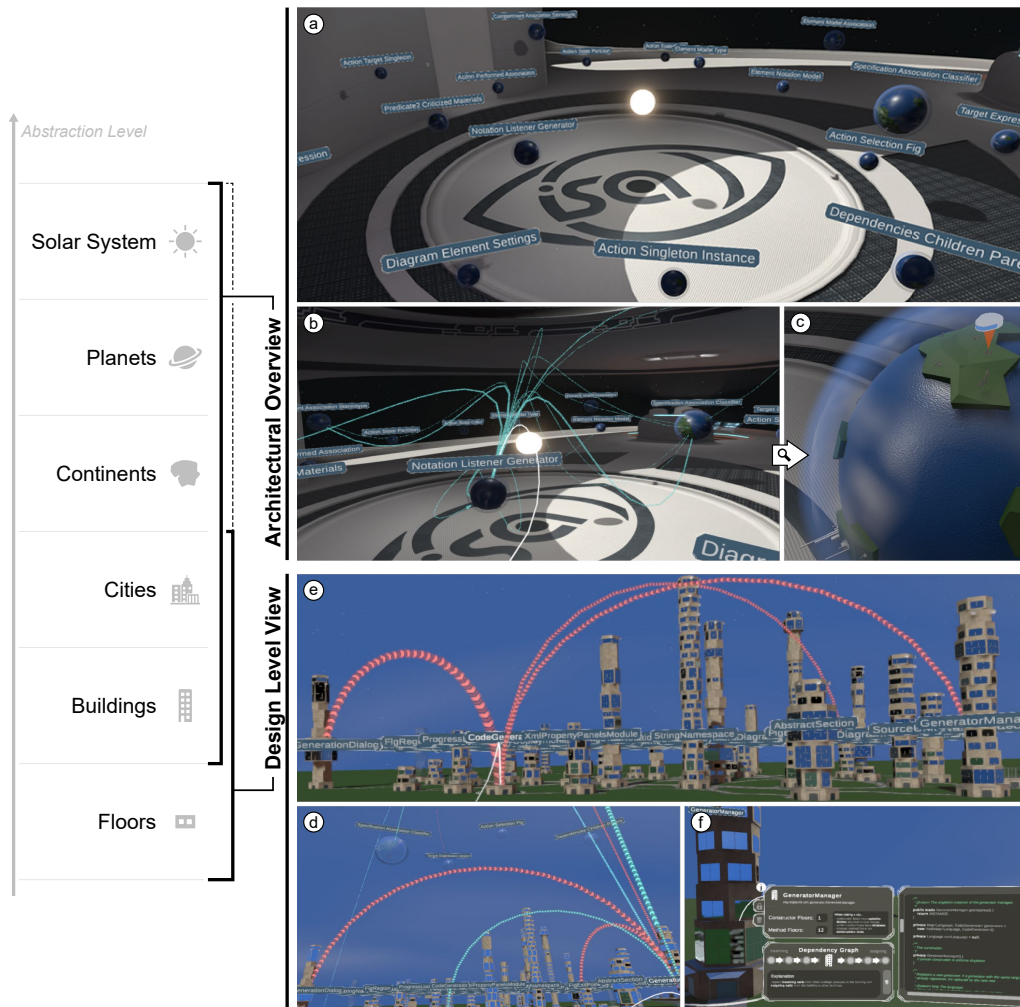


Figure B.4: Screenshots of our prototype implementation ISA showing an example system (~1.800 classifiers) on its architectural level (upper area) and semantically zoomed in on a bottom-level component (lower area). The left hand side maps the semantic zoom levels to the primarily visualized elements.

To inspect a city and its buildings in-depth, our method incorporates a semantic zoom that provides a semantically enriched view of a selected city with more details regarding design-level elements such as methods, cf. lower area of Figure B.4. While the architectural overview puts a user in the role of an overseeing observer, using the semantic zoom locates the user within a city on the surface of a planet where they can explore design-level structure from a first-person perspective.

In the design-level view, buildings are semantically enriched with further structural information to allow users to visually scan classifiers with regard to member-level metrics and, thereby, quickly spot phenomena such as partic-

ularly complex or large-scale methods. Therefore, buildings are composed of visually distinguishable floors with varying heights and diameters, similar to how their software counterparts have varying lengths and complexity. The constructors and methods of a classifier are represented as the floors of a building, where metrics drive the floor's shape. For the height of a floor, we use the number of expressions of the respective method or constructor. For the diameter of a floor, we use its cognitive complexity [27]. Abstract methods are visualized as construction sites, giving the impression of a raw and incomplete structure. Users can interact with buildings and thereby browse through the source code of resp. classifiers (Screenshot ⑥ in Figure B.4).

Maintaining Orientation A shortcoming of existing semantic zooms in 3D software visualizations is that once having zoomed in, users lack an overview of the overall system structure (Section B.2). To address this challenge, we devise concepts that put design-level information in context with the overall system structure. For one, regardless of where a user is located in our visualization, they can always interact with the architectural overview, as depicted in the upper area of Figure B.4. While the user is located in the design-level view, the architectural overview additionally highlights the currently visited city. In our prototype implementation, this is achieved via an orange arrow as depicted in Screenshot ③ in Figure B.4. For another, while users are located in the design-level view, our method additionally projects a subject system's planet structure in the sky above the visited city (Screenshot ④). This strengthens users' immersion into the metaphor while subtly providing contextual information on other parts of a system, for example via connecting lines according to element interrelations.

Relationships

Our method visualizes relationships among elements on different levels of abstraction, embedded into its semantic zoom. We distinguish between vertical relations that express containment, horizontal relations which are either based on references in code or on membership in a common parent structure and cross-hierarchical relations which are horizontal relations across components.

Vertical Relations On design level, vertical relations are explicitly available in the source code of elements, for example by the member declarations of a classifier. On architecture level, vertical relations (i.e., what classifiers belong

to a component, in what higher-level component is a component contained) need to be estimated based on the explicitly available information on horizontal and vertical relations on design level. Our method explicitly encodes vertical relations on all abstraction levels via the hierarchical organization of its visual elements. These follow a vertical path through the hierarchy of the solar system, e.g., as a building in a city on a continent. Thereby, our method encodes vertical relationships directly into its visual structure, allowing users to retrace these via the semantic zoom.

Horizontal Relations Our method explicitly incorporates two kinds of relations among elements on the same abstraction level: sibling relations and reference relations. Sibling relations are relations among elements based on their common containment in a structure, for example, two members of the same classifier as two floors in the same building. Reference relations are based on references in the source code, for example, the call of a method. We distinguish between incoming and outgoing reference relations to or from elements.

In the design-level view, our method visualizes reference relations on design level, for example among buildings. Visualized relations can be calls among the members of represented classifiers or dependencies among classifiers. These can be adopted as-is from the ground-truth design level of the visualized software structure model. Besides being useful with regard to sibling relations, the grouping of heavily interrelated buildings in the same city helps with navigating along reference relations on city level because information is more quickly accessible. Screenshot © in Figure B.4 shows an example of horizontal city-level relations in our prototype implementation, where the interrelated buildings are part of the same city.

On architecture level, our method visualizes reference relations on a more abstract level. Therefore, it determines the reference relations between components by agglomerating the dependencies or calls between their contained classifiers and sub-components accordingly. These are represented as lines between the respective planets, continents, and cities in the architectural overview. The user can choose the granularity level on which architecture-level relations are visualized. Screenshot ⑥ in Figure B.4 shows our prototype implementation of this on the example of outgoing dependencies as blue lines from a selected city, agglomerated to city level.

Cross-Hierarchical Relations Horizontal relations among elements across different components are a ubiquitous part of every software system. They

constitute to the relationship between their parent components. We refer to them as cross-hierarchical relations. In our visualization, cross-hierarchical relations manifest in form of relations among buildings across different cities and among cities across different continents and planets. Cross-hierarchical relations follow a path through the hierarchical structure of a system's architecture along its levels of abstraction. Therefore, they represent not only horizontal relations among elements, e.g., two buildings, but also diagonal relations, e.g., between a building and the city.

In the architectural overview, cross-hierarchical relations are visualized as lines that connect related elements along a path through the visualization's visual hierarchy, such as the blue lines shown in Screenshot (b). In the design-level view, cross-hierarchical relations from or to buildings in the visited city are visualized as lines originating from the respective building, pointing to a location in the planets projected into the sky as shown in Screenshot (d). Thereby, our method embeds the visualization of reference relations into the different levels of architectural abstraction and across the semantic zoom.

VR Interaction

VR as a medium for 3D software visualization can foster a more engaging exploration and easier interaction as compared to a standard screen [6, 7, 8, 30]. However, VR visualizations need to provide users with means for orientation and navigation purposes in their virtual world. To achieve that, our method incorporates VR interaction concepts that we elaborate on in the following.

Interactable Elements The architectural overview of our method displays a solar system in a room-scale size (cf. Figure B.4). Users can move back and forth between the planets and interact with them in various ways. They can place individual planets in their hands to intuitively change the point of view from which a planet is regarded. That allows to optically zoom in on structure (as done in Screenshot (c)), allowing for alternative viewing angles while improving users' ability to inspect coarse-grained visual structure closer.

The organization of our visualization's coarse-grain structure in floating planets and their continents enables our method to draw connecting lines among them with more degrees of freedom as compared to a layout that follows a flat 2-dimensional surface. Connecting lines can make use of all three available dimensions, which provides flexibility while reducing occlusion with other elements. We strengthen this effect further by making the visualization interactable, by enabling users to place planets in their hand, moving and rotating them freely, and thereby influencing the 3D paths of connections.

Information Canvases To access detail information and further interaction possibilities on demand, our method enables users to open information canvases when interacting with planets, continents, cities, and buildings (see Figure B.4 ①) in the virtual world, both in the architectural overview and the design-level view. Because diegetic user interfaces have a positive effect on the immersion and usability of VR tools [31], we design all information canvases as diegetic interfaces which we embed into our visual metaphor. When opening an information canvas, it is attached to the user’s arm where they can carry it around or detach and fixate it in space.

B.4 Evaluation

We conduct a controlled experiment with 54 participants in which we compare our approach with existing tools used for software exploration, to evaluate in what sense our approach fosters users’ ability to access and relate information on an unfamiliar large-scale software system on and across design and architecture level. Specifically, our experiment investigates three research questions, corresponding to key activities that contribute to the exploration of an unfamiliar software system.

- RQ₁** In what sense do the different tools facilitate *accessing information* on software elements such as methods, classes, or components?
- RQ₂** In what sense do the different tools facilitate *establishing horizontal relations* between software elements on the same abstraction level?
- RQ₃** In what sense do the different tools facilitate *establishing vertical relations* between software elements across different abstraction levels?

B.4.1 Subject System

We chose the large-scale open source legacy Java system ArgoUML (~1.800 classes) as subject for our experiment. ArgoUML is a graphical editor for creating, editing, and exporting diagrams of the Unified Modeling Language (UML). ArgoUML was used in prior software visualization evaluations, e.g., for the evaluation of CityVR [6] or Softwareonaut [32].

Table B.1: State-of-the-art VR software visualization tools for Java systems and their fulfillment of our inclusion criteria.

Tool	C_{Java}	C_{open}	C_{code}
IslandViz [33, 34]	✗	✓	✓
VR FlyThruCode [35]	✓	✗	✓
VR City [23]	✓	✗	✓
SEE / EvoStreets [36]	✓	✗	✓
ExplorViz [37]	✓	✓	✗
CityVR [6]	✓	✓	✓

B.4.2 Software Exploration Tools

We implement our method in a VR visualization tool called Immersive Software Archaeology² (ISA). ISA consists of an extensible analysis back-end integrated into the Eclipse IDE and a stand-alone VR visualization front-end.

As comparison for our method, we choose representatives from two kinds of software comprehension tools: As an IDE is common to explore software, we include Eclipse³ as a widely used representative. Features in Eclipse relevant for our experiment are a GOTO navigation (jump to declarations when clicking), a text search (find occurrences of text in files), a package explorer (show a system’s organization in packages), and a call hierarchy (show incoming calls to elements).

For a comparison with the state of the art, we include a VR software visualization that satisfies the following criteria:

C_{Java} is able to visualize plain Java systems, i.e., does not require a specific architecture or underlying framework (such as OSGi).

C_{open} is openly available (for replicability of the experiment), i.e., is accessible for download and free of charge.

C_{code} provides access to the source code of a subject system.

Table B.1 gives an overview of existing VR software visualization tools and their respective fulfillment of our inclusion criteria. We pre-filtered existing

²<https://gitlab.com/immersive-software-archaeology>

³<https://www.eclipse.org/>

tools that do not visualize Java code or that do not come with native VR support. As CityVR by Merino et al. [6] is the only tool that fulfills all our criteria, we include it as a representative for state-of-the-art VR software visualizations. CityVR immerses a user into a room-scale VR representation of a software system via the information city metaphor, where classes are represented as buildings and packages draw the city layout by forming slightly elevated, hierarchically nested districts. CityVR allows users to scroll through the source code of a class or interface by interacting with the respective building.

B.4.3 Experiment Procedure

We divided the experiment into three phases as depicted in Figure B.5 where participants receive tool-specific training and tool-unspecific tasks. Each participant is assigned to one of the three tools randomly. To make the experiment consistent across participants, we created documents and videos for all instructions and tasks, which can be accessed via our online appendix¹. Each experiment run takes ca. 35 to 45 minutes.

Entry Survey ① Each participant starts the experiment with an entry survey¹ with questions on experience with VR, programming proficiency (both in general and with Java), and prior contact with the subject system ArgoUML.

Tool-Specific Training ② We present each participant with a training video on how to operate their tool, e.g., navigating through implementation artifacts and accessing source code. Subsequently we provide participants access to their tool and briefly let them familiarize themselves with the tool.

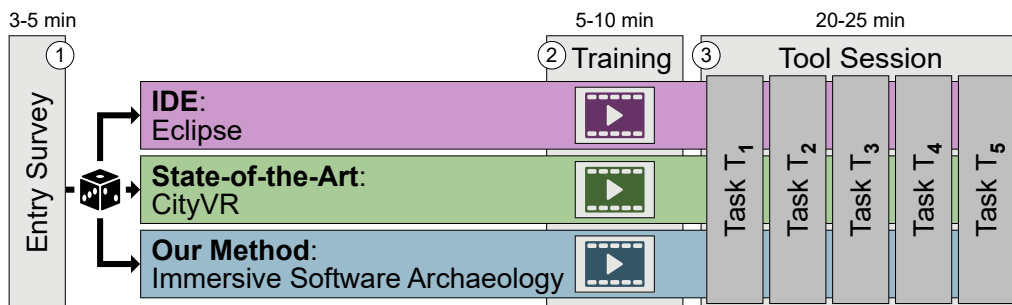


Figure B.5: Overview of the experiment procedure.

Tool Session ③ We provide each participant with five consecutive tasks to complete, which are identical across all participants and tools. The tool session is structured as a dialog between the experiment instructor (i.e., the main author of this paper) and one participant at a time. The experiment instructor reads out one task after another, in between which the participant solves them. At the same time, the experiment instructor provides guidance where necessary, following a pre-defined tool-specific catalogue¹. We record audio of the conversation between the experiment instructor and participants as well as video of their interaction with the provided tool via a screen-recording (of either the IDE window or VR viewpoint). Participants are asked to think aloud during the entire session.

B.4.4 Tasks

To allow for comparison, we designed tasks for the investigated tools so that they each emulate a focused examination of the same part of the subject system, i.e., ArgoUML's code generation feature. Table B.2 lists a shortened version of the tasks investigated throughout the tool session, along with the research questions and exploration activity they address.

Task T₁ Participants access design level information by seeking a specific Java class, given a description of its functionality (RQ₁).

Task T₂ Participants access design level information by seeking a specific Java interface, given its (non-qualified) name, before relating it horizontally with the class found in T₁ (RQ₁ & RQ₂). While solving T₁, participants encounter the interface they will search in T₂ (without knowing it) via references in code. We keep track of whether they notice this in Task T₂. As the solutions for both tasks T₁ and T₂ are prerequisites for their subsequent tasks, we provide users with help in case they cannot solve the tasks independently. While doing so, we measure the amount of guidance needed for each participant according to a scheme, i.e., [none] no help needed, [minor] the participant required a reiterated explanation from the training video, and [major] the participant cannot solve the task in time and receives the solution. Furthermore, in T₂, we measure the accuracy of participants' understanding of the relation between the two elements via a three-point grading scheme that awards one point for each of the following insights:

- There exists a relation between the elements

Table B.2: Shortened version of the tool session tasks of our experiment. The complete task sheet is accessible via our online appendix¹.

ID	Task Instructions (shortened)	Investigated Research Questions	Simplified Depiction
T ₁	There is one class in ArgoUML that is responsible for managing ArgoUML's code generators. Find that class, read its source code, and briefly describe how it works. (Solution: GeneratorManager)	RQ ₁ : Accessing information on design level.	
T ₂	Investigate whether a statement from an outdated version of ArgoUML's documentation is still valid. Search for an interface called CodeGenerator. How are the CodeGenerator and GeneratorManager (Task T ₁) related?	RQ ₁ & RQ ₂ : Accessing and horizontally relating information on design level (in depth).	
T ₃	Investigate which other parts of ArgoUML use the code generation functionality. Identify and list all classes that access functionality of the GeneratorManager and CodeGenerator (i.e., call methods, access fields, etc.)	RQ ₂ : Relating information on design level horizontally (broadly).	
T ₄	Starting with the GeneratorManager and CodeGenerator, identify and list all classes & interfaces that you think belong to ArgoUML's code generation component.	RQ ₃ : Relating information vertically, i.e., between design and architecture level.	
T ₅	Make an estimation on how much effort it would require to remove the code generation component (as you defined it in T ₄) from ArgoUML altogether (Likert-scale). Explain your estimation briefly (short text).	RQ ₂ : Relating information horizontally on architecture level.	

- The class (T_1) maintains instances of the interface (T_2)
- ... and maps these to meta information

Task T_3 Participants horizontally relate design-level information broadly, i.e., they investigate all incoming calls to two elements (RQ_2). We identify patterns in participants' answers to T_3 in terms of three categories:

- Participant finds no related classes
- Participant finds only a limited set of classes, e.g., only classes in the same package, city district, or planet
- Participant finds classes all across the system

Task T_4 Participants vertically relate design level information to architecture level information by defining a functional component based on the insight gained via the three prior tasks (RQ_3). The depictions in Table B.2 visually sketch the different tasks in a simplified way. Lastly, in Task T_5 , participants are asked to establish a horizontal relation between architecture level information (RQ_2). This aspect is particularly difficult to compare between the different tools as, of the three tested approaches, only ours explicitly works on architecture level. As a compromise, we therefore ask participants to delimit the component asked for in T_4 with the rest of the system, i.e., how heavily is it related with other parts of the system horizontally.

B.4.5 Participants

We recruited 56 participants but excluded two: one did not finish the exit survey, another had knowledge on ArgoUML's inner workings (on code level). All remaining participants are students and staff from the IT University of Copenhagen: 22 are Bachelor's students, 20 are Master's students, 9 are PhD students, and 3 are postdoctoral researchers. We distributed participants evenly across the three evaluated tools, i.e., 18 participants for each tool. Among the resulting groups, participants' prior experience levels with VR, general programming, and Java are balanced, each ranging from novices to experts.

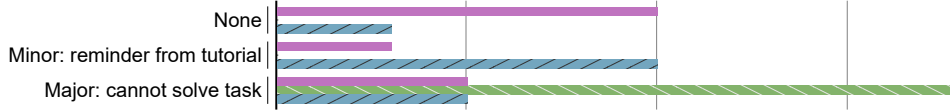
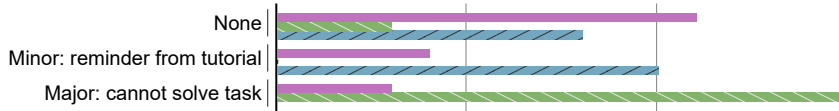
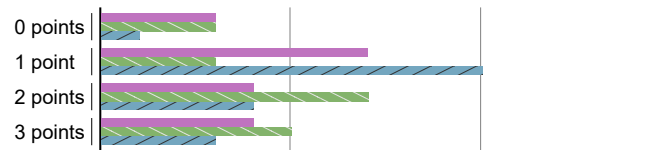
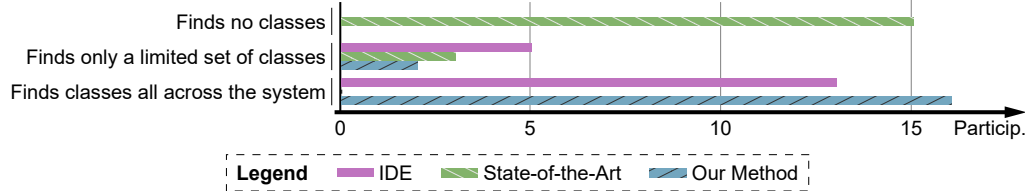
Task T₁: Guidance required to access the GENERATORMANAGER class**Task T₂: Guidance required to access the CODEGENERATOR interface****Task T₂: Extensiveness of participants' understanding of the relation****Task T₃: Extensiveness of participants' list of related classifiers**

Figure B.6: Quantitative evaluation results from the different experiment phases.

B.4.6 Findings

In the following, we both present results and discuss their implications separately for each of the posed research questions. Furthermore, summarized results are depicted in Figure B.6 and detailed results are available via our online appendix¹. To shorten explanations, we refer to participant groups for individual tools via abbreviations: $group_{IDE}$ (Eclipse), $group_{SOTA}$ (state-of-the-art visualization CityVR), $group_{ISA}$ (Immersive Software Archaeology; our implementation).

RQ₁: Accessing Information (T₁, T₂)

To solve T₁, $group_{IDE}$ employed a mixture of Eclipse's text search (13 of 18) and an exploration via the system's package structure (12 of 18), where 7 participants use both. We observed that several participants could not match the classes and interfaces they inspected with their respective location in the system's package hierarchy. For instance, although most participants (12 of 18)

remembered to have encountered the interface searched in T_2 while solving T_1 , 11 participants could not locate it in the package hierarchy and, instead, used the text search to solve T_2 .

Solving tasks T_1 and T_2 each required $\text{group}_{\text{SOTA}}$ to find one specific building in the visualized city. During their search, all 18 participants were drawn to particularly large buildings. While this let them explore various classes throughout the entire system, the building searched in T_1 was not among these for any participant. To solve T_2 , most participants (14) made use of the city layout to narrow down their search, i.e., they assumed the searched building was in proximity to the building found in T_1 . While only few (3 of 18) participants could solve T_2 , we observe that, in contrast to the IDE, $\text{group}_{\text{SOTA}}$ developed an overview of where in the visualization elements are located.

Similar to $\text{group}_{\text{SOTA}}$, solving tasks T_1 and T_2 each required $\text{group}_{\text{ISA}}$ to find one specific building in the solar system. To solve T_1 , all 18 participants explored the planet structure of the architectural overview. We notice that, similar to $\text{group}_{\text{SOTA}}$, participants in $\text{group}_{\text{ISA}}$ were drawn to large structures, i.e., large planets, continents, and large buildings on planet surfaces. However, to solve T_1 , almost all participants (17 of 18) utilized the text search to locate occurrences of elements with promising names. We observe that the word occurrence tags helped participants with prioritizing their exploration and search results. Notably, $\text{group}_{\text{ISA}}$ approached T_2 vastly different than they approached T_1 . That is, to solve T_2 , only 5 participants used the text search, while all others started exploring the city they have previously entered to solve T_1 by inspecting buildings and utilizing the relationship graphs.

When comparing the IDE with our method in terms of participants' ability to solve T_1 and T_2 (i.e., required no help or only a reminder on tool functionality), we observe only minor differences, despite participants' familiarity with IDEs and 2D interfaces. That is an identical performance in T_1 and a better performance of our method in T_2 where 3 participants in $\text{group}_{\text{IDE}}$ did not solve the task whereas it is 0 for our method.

Discussion An IDE locates users on a low abstraction level where they are able to access and manipulate design-level information. Our observations and results support that this can cause a lack of overview. The goal of the state-of-the-art city metaphor visualization is to provide its users with an extensive view on design-level information, where city layout and shapes of elements are driven by metrics. In our experiment, we find that this impedes participants' access to information other than finding outliers according to the represented metrics (cf. required guidance for tasks T_1 and T_2 in Figure B.6).

We conclude that, by grouping interrelated elements into the same structures (e.g., buildings in the same city, cities on the same continent), our method allows for easier access to information on similar functionality as compared to grouping elements based purely on a package hierarchy (as done in the IDE and state-of-the-art visualization). This shows especially in participants' approach to solving task T₂, where our method relieved participants of finding relevant software elements for the most part.

RQ₂: Relating Information Horizontally (T₂, T₃, T₅)

We investigate RQ₂ via Tasks T₂ and T₃ on design level and via T₅ on architecture level.

Design Level To solve T₂, participants in group_{IDE} generally related elements via reading code. Only 1 participant used Eclipse's call hierarchy feature. In contrast, to solve T₃, 15 participants used the call hierarchy feature whereas 3 instead used the text search. As a result, 13 participants were able to find all requested horizontal relations on design level, 5 participants found only relations to elements within the same package as the investigated class and interface (see Figure B.6). We observe that participants in group_{IDE} were generally not satisfied with the tool support they received for relating elements horizontally, e.g., they needed to query the call hierarchy for each member individually. As a consequence, one participant approached T₃ by deleting elements, recompiling the system, and inspecting all files with compilation errors.

Participants in group_{SOTA} relied on reading code because the respective tool does not visualize relations specifically. While 15 participants did not solve T₃, they generally solved T₂ in more detail than participants using other tools (see Figure B.6).

To solve T₂, 11 participants in group_{ISA} based their answer on the relationship graphs. The remaining 7 participants solved the task by reading through the class' source code. Overall, a majority of participants in group_{ISA} answered T₂ not in much detail, i.e., 10 participants score only 1 point. At the same time, we observe that only 1 participant using our tool is not able to establish a relation at all, while it were 3 participants of each of the other tools. To solve T₃, all participants in group_{ISA} (no exception) made use of the relationship graphs. As a result, 16 of 18 participants are able to find all relationships, while 2 participants miss classes located on other planets.

Architecture Level In their answers to Task T₅, we generally notice that group_{IDE} based their explanations on code constructs (mainly classes) while group_{SOTA} based their answers on visual elements (buildings and city districts). On the other hand, while most participants in group_{ISA} use code constructs (mainly classes), some mix them with visual elements (continents and planets) when describing architectural structure.

Similar to group_{IDE}, multiple participants in group_{SOTA} answer T₅ based on direct observations in the visualization (e.g., assuming few interrelations of a classifier with other parts of the system because its city district was small).

While multiple participants in group_{ISA} equally used visual elements in their explanations, those with decisive answers on architectural concerns had a clear tendency towards using the relationship graph to argue on various abstraction levels, i.e., intra-component connections of classes (buildings in same city) as well as inter-component connections (buildings in different cities). In contrast to the other tools, some participants have very concrete ideas regarding the size of the code generation component and how it is related with the rest of the system.

Discussion While the call hierarchy and GOTO navigation allow quick traversal of the system's call graph, several participants in group_{IDE} mentioned that these features were not ideal for a broad investigation of relations on classifier level such as in T₃. We conclude that an IDE operates on a lower level of abstraction than ideal for horizontally relating elements on a higher level than members, even when inspecting only one specific feature as emulated by T₃. On architecture level (T₅), this shortcoming manifests in vague or incomplete answers.

The state-of-the-art visualization used in our experiment does not include an explicit visualization of relationships. Thus, we cannot discuss its suitability for fostering the exploration of such. However, it allows for conclusions towards the benefits and drawbacks of an explicit visualization of relationships as encompassed in our method. Relying on establishing a relationship purely based on code resulted in more accurate description of group_{SOTA} as compared to group_{ISA} in T₂ where the elements to relate were known and available, but significantly worse results in T₃ where the elements to relate were unknown (cf. Figure B.6). This translates to architectural level, i.e., because they could not solve T₃, group_{SOTA} reported a lack of overview when solving T₅.

With the relationship graphs provided by our method, participants in group_{ISA} across all programming experience levels were able to relate ele-

ments across abstraction levels. On design level, this shows in T_2 where only 1 participant was not able to establish a relation and in T_3 where only 2 participants missed relations to the investigated elements (see. Figure B.6). In extension to our answer to RQ_1 , we conclude that by easing the access to information via the grouping of interrelated elements, our method also fosters establishing horizontal relations. Participants in $group_{ISA}$ generally provided better arguments (see above) for their answers to T_5 as compared to $group_{IDE}$ and $group_{SOTA}$, indicating that they developed a better overview of the system's architecture as compared to participants in other groups.

RQ₃: Relating Information Vertically (T_4)

To solve Task T_4 , participants in $group_{IDE}$ used intersecting combinations of exploring the system's package hierarchy (8), reading through code (9), the text search (5), the call hierarchy (9), and deleting classifiers to see which other elements break (2). While 5 participants answered T_4 very broadly, i.e., 3 pointed to an entire package containing hundreds of classes while 2 based their answer entirely on a search term with hundreds of matching classes, 4 participants were very restrictive, i.e., included only 2 or 3 classes. One participant stated to miss detail knowledge to formulate a sensible answer and did not answer the task. In contrast, 4 participants in $group_{IDE}$ solved T_4 thoroughly by reading through several classifiers in a bottom-up approach.

Participants in $group_{SOTA}$ answered T_4 superficially by either pointing to city districts (Java packages) or including only the two core classifiers (provided in the task description of T_4). One participant did not know how to solve the task at all.

Similar to T_3 , participants in $group_{ISA}$ made use of the dependency graph to solve T_4 . Generally, they expanded upon their answers to T_3 by retracing additional references. That is, 17 out of 18 participants in $group_{ISA}$ vertically related classifiers to the asked component based on relationships with the provided core classifiers. However, similar to $group_{IDE}$, we observe disparate inclusion criteria, i.e., some participants included all classifiers that have any form of relation to the core classifiers, others were more selective and additionally took class names and source code into account. Only 4 participants read through source code as a part of that process.

Discussion Participants in $group_{IDE}$ were generally undecided how to approach establishing a vertical relationship between the asked component and its containment. This mirrors in the variety of different IDE features used (11

participants used 2 or more different features) and the varying degree of detail in participants' answers, ranging from a handful of classes to packages with hundreds of classes. While we, the authors of this paper, are not aware of the subject system's ground-truth architecture, it is safe to assume that the asked component's actual size is not in the range of hundreds of classes. Building up on our previous conclusions, we attribute these estimations to a missing overview on the system's architecture.

Although the used state-of-the-art tool is slim in its feature set (does not visualize relations, allows access to only one class at a time), it provided equally many participants with good enough of an architectural overview to give an answer to T_4 as the IDE. Also, while participants in $\text{group}_{\text{SOTA}}$ all provide superficial answers based on city districts (packages), their answers generally encompassed more sensible amounts of classifiers than the answers of a majority of $\text{group}_{\text{IDE}}$.

The differences in the vertical relation approach of participants in $\text{group}_{\text{ISA}}$ were considerably less far apart than those in the other groups, especially than those in $\text{group}_{\text{IDE}}$. Because 17 out of 18 participants in $\text{group}_{\text{ISA}}$ vertically related classifiers to a component based on their relationship with the core classifiers of the component, they formed more cohesive and sensible components than the participants in the other groups.

B.4.7 Threats to Validity

Construct Validity is concerned with the extent to which an experiment setup actually investigates the subject of the experiment. Our experiment subject was to assess the suitability of different tools for the exploration of an unfamiliar large-scale software system. We formulated three research questions to investigate that and constructed experiment tasks accordingly, on the basis of a real-world software system. While RQ_1 and RQ_2 are addressed by two and three tasks respectively, RQ_3 is addressed by only one task, because we investigate it in depth on architecture level (what classes make up a component) rather than on design level (what members belong to a class).

Internal Validity is concerned with uncontrolled influences that falsely indicate a causal relationship. We minimized this risk by assuring similar conditions for each experiment run with the used tool and its medium as dependent variables. To achieve that, we randomly grouped participants to the three tools while providing the same tasks across all groups. The resulting groups

were equally divided regarding experiences in relevant aspects, i.e., prior experience with VR and programming¹. We designed the tasks of our experiment to not favor textual or visual representations. Furthermore, we provided each participant with a short training video for their assigned tool¹.

External Validity is concerned with the degree to which experimental results can be generalized to settings other than in the experiment. Despite multiple international students from Asia, the vast majority of participants in our experiment were of Northern European origin. With 18 participants per group (54 in total), our results would be more conclusive with a larger sample size. Furthermore, because we recruited mostly students (no practitioners with professional experience), our results hold for a rather inexperienced audience. However, a large majority of our participants declared to program regularly (80.4% program at least once a week) and to be experienced with Java (82.1% self-assessed to at least medium experience on a 5-step Likert scale)¹. We argue that this is indeed an interesting target audience for our method, as it resembles young professionals – a group of people that will have to work with the legacy code produced by current working professionals.

B.5 Conclusion and Future Work

We presented an approach for analyzing and visualizing large-scale software systems for the purpose of their comprehension. In a controlled experiment with 54 participants, we compare its ability to support users with key aspects of software comprehension with an IDE and a state-of-the-art VR software visualization. Our results show that our approach provides engineers with easier access to information, including a better overview of a system's architecture and relationships among elements on all encompassed abstraction levels.

In the future, we will enable engineers to refine the recovered architecture according to their mental model by reorganizing the architectural structure within our visualization, e.g., to adjust component boundaries. Furthermore, we plan to foster engineers' exploration of a system's structure via additional software characteristics, such as a system's behavior or quality.

Bibliography

- [1] Robert Behling, Chris Behling, and Kenneth Sousa. Software reengineering: concepts and methodology. *Industrial Management & Data Systems*, 96(6), 1996.
- [2] Harry Sneed and Chris Verhoef. Re-implementing a legacy system. *Journal of Systems and Software*, 155, 2019.
- [3] Wilhelm Hasselbring. Software Architecture: Past, Present, Future. In *The Essence of Software Engineering*. 2018.
- [4] Michael L Nelson. A survey of reverse engineering and program comprehension. *arXiv preprint cs/0503068*, 2005.
- [5] Laure Bedu, Olivier Tinh, and Fabio Petrillo. A tertiary systematic literature review on software visualization. In *Working Conference on Software Visualization (VISSOFT)*, 2019.
- [6] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.
- [7] Simone Romano, Nicola Capece, Ugo Erra, Giuseppe Scanniello, and Michele Lanza. On the use of virtual reality in software visualization: The case of the city metaphor. *Information and Software Technology*, 114, 2019.
- [8] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesús M González-Barahona, and Michele Lanza. Codecity: On-screen or in virtual reality? In *Working Conference on Software Visualization (VISSOFT)*, 2021.
- [9] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *STRAW*, volume 3, 2003.
- [10] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väättäjä. Software visualization today: systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, 2016.

- [11] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4), 2020.
- [12] Alfredo R Teyseyre and Marcelo R Campo. An overview of 3d software visualization. *IEEE transactions on visualization and computer graphics*, 15(1), 2008.
- [13] Vladimir Averbukh, Natalya Averbukh, Pavel Vasev, Ilya Gvozdarev, Georgy Levchuk, Leonid Melkozerov, and Igor Mikhaylov. Metaphors for software visualization systems based on virtual reality. In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*, 2019.
- [14] Sheelagh Cpendale and Yaser Ghanam. A survey paper on software architecture visualization. Technical report, University of Calgary, 2008.
- [15] P. Caserta and O Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 2011.
- [16] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and evolution of software architectures. In *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011*, 2012.
- [17] Peter Young and Malcolm Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*, 1998.
- [18] Jonathan I Maletic, Jason Leigh, Andrian Marcus, and Greg Dunlap. Visualizing object-oriented software in virtual reality. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, 2001.
- [19] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, 2011.
- [20] Philippe Dugerdil and Sazzadul Alam. Execution trace visualization in a 3D space. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, 2008.

- [21] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2), 2013.
- [22] David Baum, Jan Schilbach, Pascal Kovacs, Ulrich Eisenecker, and Richard Müller. Getaviz: generating structural, behavioral, and evolutionary views of software systems for empirical evaluation. In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2017.
- [23] Juraj Vincur, Pavol Navrat, and Ivan Polasek. VR City: Software Analysis in Virtual Reality Environment. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017.
- [24] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *IEEE TCVG*, 2004.
- [25] Michael Balzer and Oliver Deussen. Hierarchy based 3d visualization of large software structures. In *IEEE Visualization 2004*, 2004.
- [26] Michael Balzer and Oliver Deussen. Level-of-detail visualization of clustered graph layouts. In *6th International Asia-Pacific Symposium on Visualization*, 2007.
- [27] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, 2018.
- [28] Thibaud Lutellier, Devin Chollak, Joshua Garcia, Lin Tan, Derek Rayside, Nenad Medvidovic, and Robert Kroeger. Comparing Software Architecture Recovery Techniques Using Accurate Dependencies. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.
- [29] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, 1996.
- [30] Enes Yigitbas, Simon Gorissen, Nils Weidmann, and Gregor Engels. Collaborative Software Modeling in Virtual Reality. In *ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2021.
- [31] Paola Salomoni, Catia Prandi, Marco Rocchetti, Lorenzo Casanova, and Luca Marchetti. Assessing the efficacy of a diegetic game interface with Oculus Rift. In *13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2016.

-
- [32] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with SoftwareNaut. *Science of Computer Programming*, 79, 2014.
 - [33] Martin Misiak, Andreas Schreiber, Arnulph Fuhrmann, Sascha Zur, Doreen Seider, and Lisa Nafeie. IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality. In *IEEE Working Conference on Software Visualization (VISSOFT)*, 2018.
 - [34] Andreas Schreiber, Lisa Nafeie, Artur Baranowski, Peter Seipel, and Martin Misiak. Visualization of Software Architectures in Virtual Reality and Augmented Reality. In *2019 IEEE Aerospace Conference*, 2019.
 - [35] Roy Oberhauser and Carsten Lecon. Virtual Reality Flythrough of Program Code Structures. In *Proceedings of the Virtual Reality International Conference - Laval Virtual 2017 on - VRIC '17*, 2017.
 - [36] Marcel Steinbeck, Rainer Koschke, and Marc O Rüdell. How evostreets are observed in three-dimensional and virtual reality environments. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.
 - [37] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013.

Appendix

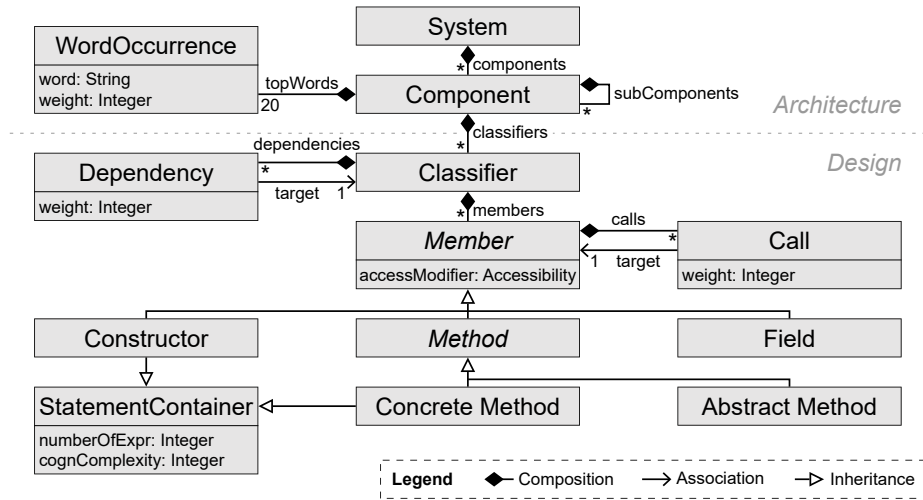


Figure B.7: Data structure meta model employed in our method as provided in the online appendix¹.

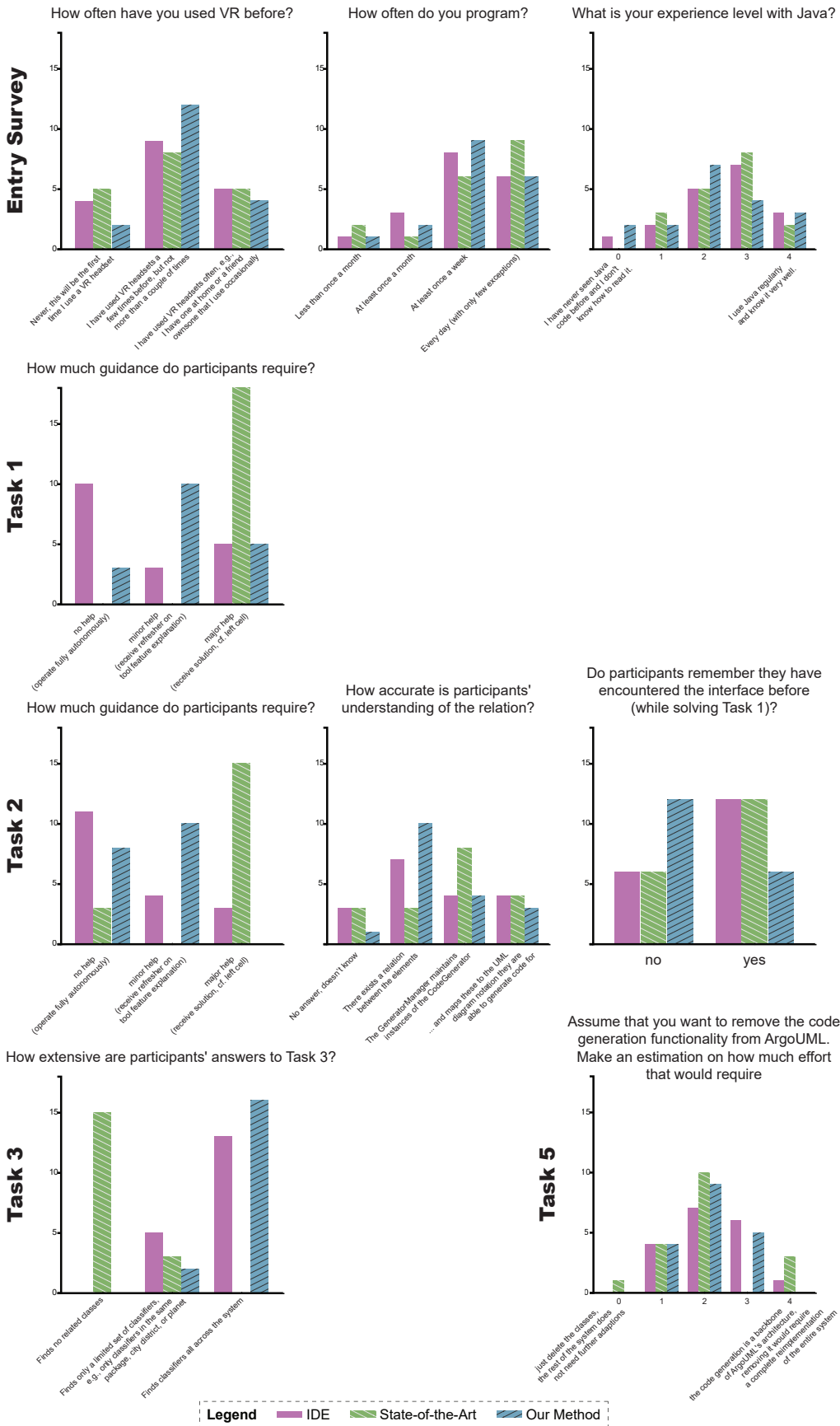
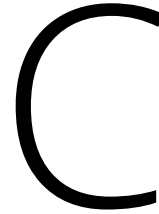


Figure B.8: Additional bar charts from the experiment as provided in the online appendix¹.



Immersive Software Archaeology: Exploring Software Architecture and Design in Virtual Reality

Originally published in: Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2024), March 12–15, Rovaniemi, Finland

Joint work with: Christoph Seidl and Michele Lanza

Abstract

Comprehending large-scale software systems is a challenging and daunting task, particularly when only source code is available. While software visualization attempts to aid that process, existing tools primarily visualize a system's structure in terms of files, folders, packages, or namespaces, neglecting its logical decomposition into cohesive architectural components.

We present the tool Immersive Software Archaeology (ISA) which (i) estimates a view of a system's architecture by utilizing concepts from software architecture recovery and (ii) visualizes the results in virtual reality (VR) so that users can explore a subject system interactively, making the process more engaging. In VR, a semantic zoom lets users gradually transition between architectural components of different granularity and class-level elements while relationship graphs let users navigate along connections across classes and architectural components.

We present results from a controlled experiment with 54 participants to investigate the usefulness of ISA for assisting engineers with exploring an unfamiliar large-scale system compared to another state-of-the-art VR approach and an IDE.

Video Demonstration – https://youtu.be/Fl_SsT13l4k

C.1 Introduction

Software archaeology is the process of recovering knowledge on an unfamiliar (legacy) software system using the artifacts available, which commonly constitute only source code [1, 2]. A relevant activity is recovering a subject system’s architecture which - when conducted based on source code alone - is tedious, challenging, and error-prone [3, 2]. By explicitly encoding information on a system’s structure, behavior, and/or evolution via a visual metaphor, software visualization supports various tasks such as estimating or relating architectural components.

The extent of existing tool support is limited: current 3D/VR software visualizations (i) rigidly locate users on an abstraction level only slightly above source code while (ii) visualizing a system’s architecture in terms of its organization in folders, namespaces, or other “physical” structures.

We present the tool Immersive Software Archaeology¹ (ISA), which supports software archaeologists with recovering knowledge on an unfamiliar large-scale system by visualizing its architecture and design in immersive virtual reality (VR) based on its source code alone. Throughout the paper, we use the term “design” to refer to elements on class level and below, which are represented explicitly in source code, and the term “architecture” to refer to logical elements more abstract than class level, which are not represented explicitly in source code.

We present four core features of ISA (cf. Figure C.1):

- ① ISA employs a fully-automated relationship-based clustering method to analyze a system’s architecture in terms of cohesive architectural components.
- ② ISA presents the results of its analysis via an immersive, interactive metaphor in virtual reality.
- ③ ISA implements a semantic zoom that lets users navigate seamlessly along abstraction levels from architecture down to design level (and vice versa).
- ④ ISA visualizes relationships between elements (i.e., classes and architectural components) via interactive relationship graphs, among and across abstraction levels.

¹<https://gitlab.com/immersive-software-archaeology>

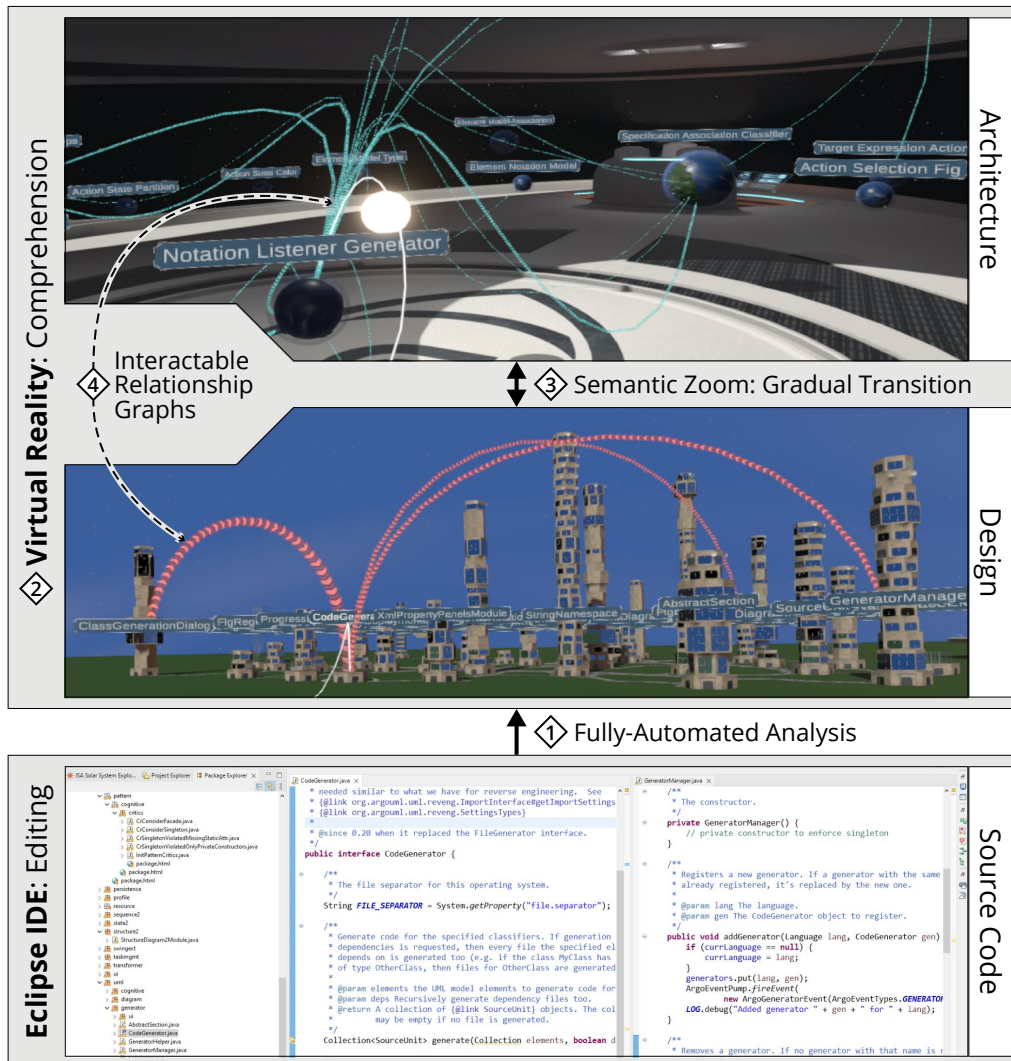


Figure C.1: Overview of ISA from a user’s perspective: After analyzing a subject system in the IDE, users explore its architecture and design in VR. They can transition between abstraction levels via a semantic zoom and navigate along relationships via an interactive graph.

We evaluated ISA in a controlled experiment with 54 participants [4], comparing it to an existing state-of-the-art VR software visualization and the Eclipse IDE. Our results show that ISA provides better access to information while fostering users’ ability to relate software elements (classes, architectural components, etc.) with one another.

C.2 State of the Art

Many software visualization tools exist that present a system's architecture [5, 6], both in 2D (e.g., Polymetric Views [7], Samoa [8], SoftwareNaut [9], Beck et al. [10]) and 3D (e.g., Balzer and Deussen [11], IslandViz [12], ExplorViz [13], CityVR [14], CodeCity [15]). Experiments showed that using a well-constructed 3D software visualization is more effective than using an IDE in terms of correctness and completion time for software comprehension tasks [15]. In addition, 3D visualizations have advantages over 2D visualizations in terms of spatial memory [16]. 3D metaphors become further effective (task completion time and accuracy) when using the immersive capabilities of VR to allow for a more natural interaction with a 3D scene as compared to a standard-screen representation [17, 18].

Metaphors for 3D software visualization can be divided into abstract metaphors (e.g., 3D graphs [11, 19, 20]) and real-world metaphors (e.g., city metaphor [15, 14, 17], island metaphor [12], or solar metaphors [21, 22, 23, 24]). A metaphor has to be expressive enough to describe different abstraction levels without breaking while letting users transition between these.

We observe two major shortcomings in existing 3D (and thus VR) software visualization tools with regard to presenting a system's architecture. Many tools [14, 13, 12, 15] rigidly lock their users on an abstraction level slightly above the level of (textual) source code, where they focus on code metrics (such as lines of code) instead of providing an overview of a system's architecture. Furthermore, even existing tools focusing on a system's architecture commonly visualize merely folders, packages, or namespaces [5], although these "physical" structures can deviate heavily from a logical arrangement into architectural components (potentially cutting across folders, packages, and namespaces) – especially after evolution cycles in a legacy software system. This leaves open potential for supporting users in relevant software archaeology tasks such as identifying and relating architectural components.

C.3 Immersive Software Archaeology

Our tool Immersive Software Archaeology (ISA) is comprised of two parts, i.e., an extensible platform of Eclipse² plugins for the automated analysis of a subject system's architecture and an immersive VR application developed with the Unity 3D engine³ for the exploration of a system based on a previously

²<https://www.eclipse.org/>

³<https://unity.com>

conducted analysis. Once having analyzed a system in Eclipse, users can enter ISA's VR visualization and inspect the analyzed system via a real-world metaphor (see Section C.3.2) with planets, continents, cities, and buildings, where buildings represent design-level elements (i.e., on the level of classes) and cities, continents, and planets represent higher-level architectural components (i.e., based on a logical structure that is not directly visible in code). From within the VR visualization, users can access a backchannel to the IDE. That is, they can open windows in the IDE from within VR that display the source code of visualized files, e.g., to then directly perform changes in the IDE.

C.3.1 Automated Architecture and Design Analysis in Eclipse

ISA provides a fully-automated analysis of a software system that takes as input the source code of a subject system and generates as output a coherent model of its architecture and design. Along with our tool, we provide an implementation for the analysis of Java software systems. ISA's analysis consists of two subsequent steps.

First, to provide users with design-level information, ISA's analysis extracts design-level information explicit in source code, i.e., the structure of classes, interfaces, enums, etc. with their members and references between them. In addition, ISA calculates metrics for design-level elements such as the cognitive complexity of methods [25]. An output model is populated with the analysis results to capture a view on an entire subject system's design-level structure.

Second, to provide users with an overview of a system's architecture, we provide a software architecture recovery that estimates a decomposition of the previously analyzed design-level elements into a hierarchy of architectural components (i.e., coherent units of functionality). To achieve this, ISA employs a software clustering technique based on the references between classes as described in [4]. The resulting output model encompasses all design-level elements of a system, organized into a hierarchy of cohesive architectural components. When starting ISA's VR application, this model is encoded into the high-level structure of the visualization, determining how planets, their hierarchies of continents, city layouts, as well as the buildings with their individual floors are generated.

On a technical level, ISA's overall analysis process described above is implemented as an extensible platform of Eclipse plugins, each registering with a central analysis core plugin. Developers can implement custom analysis

functionality in their own plugins, e.g., to integrate further target languages (for instance, based on SrcML⁴ [26] or Tree-Sitter⁵), or to employ an alternative clustering strategy. An analysis configuration dialog (① in Figure C.2) lets users choose between the available plugins for each step of the analysis. Furthermore, ISA provides detail settings for its plugin, e.g, to adjust parameters for the software architecture recovery procedure (② in Figure C.2).

We define the above described models for capturing a system’s architecture and design via the Eclipse Modeling Framework⁶ (EMF). By generating source code infrastructure from these models in both Java (for use with Eclipse) and C# (for use with Unity), they serve two purposes. For one, they act as interfaces between the different analysis steps in the IDE, e.g., between design analysis and architecture analysis as described above. For another, they serve as input during the VR visualization’s initialization.

C.3.2 Virtual Reality Exploration of Architecture and Design

ISA’s VR visualization employs an immersive real-world metaphor representation of a subject system, where users travel through and interact with its architecture and design. By explicitly representing architectural components as first-level structural elements, ISA facilitates users’ ability to gain an architectural overview of a system. A semantic zoom lets users gradually transition from this architectural overview to architectural components of lower abstraction levels, down to design-level elements such as classes and their members. Relationship graphs visualize calls and references between classes and architectural components in the context of the user’s current abstraction level.

We implement the ISA VR application with the Unity 3D engine³, building upon the SteamVR⁷ library to support major VR headsets currently available. We illustrate its visualization technique with structures known from object-oriented programming (e.g., classes and interfaces). However, ISA is able to visualize systems implemented in any programming languages with an organization of methods/functions in modules (such as classes) and a hierarchical organization of these.

⁴<https://www.srcml.org>

⁵<https://tree-sitter.github.io/tree-sitter>

⁶<https://www.eclipse.org/modeling/emf>

⁷<https://www.steamvr.com/en>

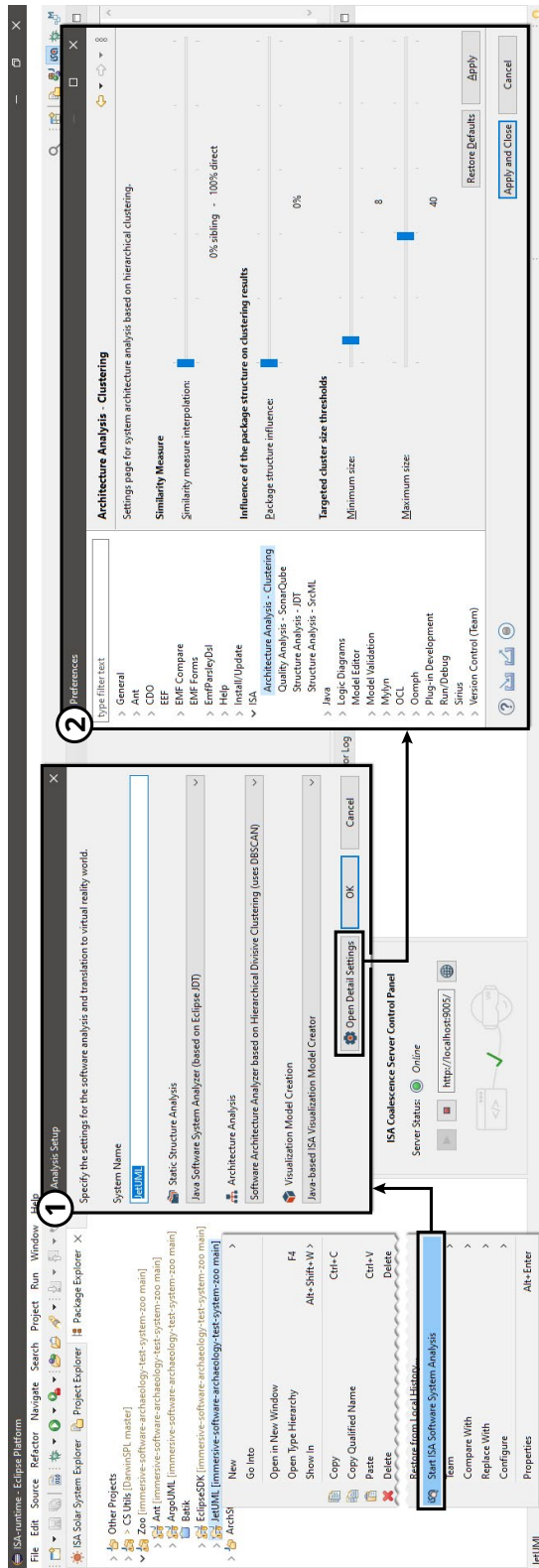


Figure C.2: Screenshots of ISAs extensible Eclipse plugins, where (1) users choose between different analysis plugins and (2) optionally change detail settings for these. Users can implement their own ISA analysis plugins and make them available for selection by registering with a central analysis core plugin.

Architecture-Level Overview

To foster users' ability to explore and comprehend the architecture of an unfamiliar system, ISA visualizes architectural components as first-level structures in its visualization: It employs a solar system metaphor where the hierarchically organized architectural components recovered previously (see Section C.3.1) are represented as planets (Ⓐ-Ⓔ in Figure C.3). Corresponding to the hierarchical organization of the architectural component it represents, each planet contains hierarchically organized pieces of land [4] – similar to how continents, countries, and regions are hierarchically organized in the real world. On the lowest abstraction level, architectural components are represented as cities consisting of buildings which each represent a class-level element (classes, interfaces, enums, etc.) of the subject system. Buildings consist of floors, each representing a method or constructor with the number of expressions determining the floor's height and the cognitive complexity [25] determining the floor's diameter. Further, ISA visualizes member encapsulation by equipping floors with windows (not encapsulated – public keyword in Java) or without windows (restricted access, e.g., private) (Ⓖ). By using these metrics to determine the visual appearance of each building's floors (and thus the building itself), ISA provides users with an impression of the represented class' structure without having to read its source code.

Semantic Zoom

To provide users with access to information on a system while not overwhelming them with details, ISA implements a semantic zoom that displays visual elements (such as buildings) context-sensitively, i.e., with more or less detail depending on the abstraction level of users. When starting the ISA VR visualization, users are initially located in an interactive room-scale overview of a system's architecture (Ⓐ-Ⓔ in Figure C.3), where they can inspect and interact with its architectural components in form of planets, hierarchies of continents, and cities. On this abstraction level, cities on planets consist of simplified versions of their buildings that visually average the metrics for their floors, cf. Ⓓ and Ⓔ in Figure C.3. Users can interact with planets by grabbing and repositioning them (e.g., to locate two planets close to one another to indicate coherence) and opening information canvases to inspect further details (e.g., the number of buildings on a planet).

Users can change their perspective from the architectural overview to a detailed city view (Ⓕ-Ⓖ in Figure C.3), where buildings present each method

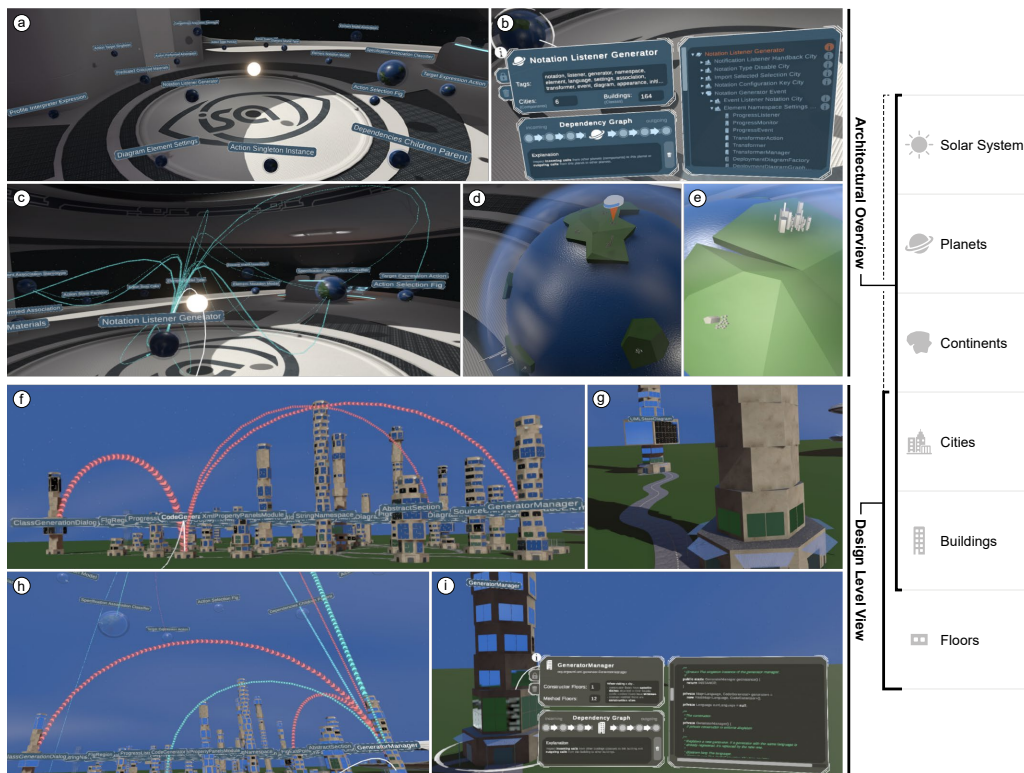


Figure C.3: Screenshots of ISA's VR application: a system's hierarchy of architectural components is visualized as a solar system with planets, hierarchies of continents, and cities, while design-level elements (classes, methods, etc.) are visualized as buildings and floors. A semantic zoom enables users to gradually transition between abstraction levels, while relationship graphs provide navigation along relationships.

and constructor of the represented classes explicitly with metric values visualizing structural properties. Within a city, users can freely inspect design-level structures, open information on classes, and read their source code. This semantic zoom enables users to gradually change their perspective from an overview of a system architecture down to design level and vice versa, following Shneiderman's mantra "Overview first, zoom and filter, then details on-demand" [27].

Interactable Relationship Graphs

To support users in gaining an overview of the relationships among and across architectural components and classes, ISA encompasses interactable relationship graphs. These visualize relationships between software elements (i.e., classes or architectural components) as curved lines between their visualization counterparts, with animated textures indicating the flow direction

of information. To visualize relationships on architecture level (e.g., between two architectural components represented as planets), ISA automatically aggregates and bundles references of architectural components, i.e., all method calls, field accesses, and type references (incl. inheritance). Users can interact with these relationship graphs via a VR user interface (ⓑ and ⓘ in Figure C.3) that enables them to highlight forward references (connecting to all referenced elements) and backward references (connecting all referencing elements), including their transitive closures. Thereby, ISA aids users in tasks such as identifying module boundaries, e.g., by investigating whether a selected class is strongly connected with a selection of other classes.

C.4 Evaluation

We evaluated ISA in a controlled experiment with 54 participants [4], where we compared its ability to support engineers with performing software archaeology tasks with (i) an existing state-of-the-art VR software visualization (CityVR [14]) and (ii) an IDE (Eclipse) as a baseline. We split the participants into three groups, each using a different tool (i.e., ISA, CityVR, or Eclipse), and let them explore a real-world legacy subject system via five software archaeology tasks (identical across participants and tools) on accessing and relating information, e.g., finding a part of the system given a description of its functionality. Thereby, we simulated a step-wise exploration process in which participants established an understanding of the subject system, while measuring and observing their behavior and approaches toward solving each task. Based on that, we concluded in what sense each of the three tools fosters the software archaeology process. Detailed task descriptions, results, and a replication package are available in the study's online appendix⁸.

Our results [4] show that, in comparison with the other tools, ISA's organization of a system's classes in cohesive architectural components provides better access to information because interrelated classes (and components) are likely to be part of the same structure (e.g., buildings in the same city) and can thus be explored side-by-side. In combination with ISA's interactable relationship graphs, participants were able to estimate components more sensibly in terms of size and content as compared to the state-of-the-art VR visualization and the IDE. Conversely, while the state-of-the-art VR visualization did not provide functionality for participants to explore relationships at all,

⁸<https://gitlab.com/immersive-software-archaeology/publication-vissoft22>

ISA's interactable relationship graphs caused a tendency for participants to not investigate relations between elements as thoroughly as in the IDE – that is, beyond seeing a line between the respective visual elements (e.g., buildings in a city). However, ISA's combination of the interactable relationship graphs with its semantic zoom, especially its ability to agglomerate relationships to architecture level, showed to be useful particularly for inexperienced engineers who, when using one of the other tools, faced difficulties with reconstructing relationships even on class level.

C.5 Conclusion and Future Work

We presented the tool Immersive Software Archaeology (ISA) which supports users in recovering knowledge of an unfamiliar software system via an immersive VR visualization of the system based on only its source code. To achieve this, ISA provides (i) a fully-automated configurable and extensible architecture recovery method and, based on that, (ii) an interactive VR visualization that lets users immersively explore and interact with a subject system's architecture and design. A semantic zoom in VR lets users transition between abstraction levels of architecture and design, while interactable relationship graphs provide an overview of interactions between elements, e.g., between architectural components visualized as cities on different planets, between classes visualized as buildings on the same planet, etc.

Bibliography

- [1] Gregorio Robles, Jesus M Gonzalez-Barahona, and Israel Herraiz. An empirical approach to software archaeology. In *ICSM 2005*, 2005.
- [2] Harry Sneed and Chris Verhoef. Re-implementing a legacy system. *Journal of Systems and Software*, 155, 2019.
- [3] Asit Kumar Gahalaut. REVERSE ENGINEERING: AN ESSENCE FOR SOFTWARE RE-ENGINEERING AND PROGRAM ANALYSIS. *International Journal of Engineering Science and Technology*, 2:9, 2010.
- [4] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. In *VISSOFT 2022*, 2022.

- [5] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94, 2014.
- [6] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4), 2020.
- [7] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9), September 2003.
- [8] Roberto Minelli and Michele Lanza. SAMOA A Visual Software Analytics Platform for Mobile Applications. In *2013 IEEE International Conference on Software Maintenance*, September 2013.
- [9] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwarentaut. *Science of Computer Programming*, 79, 2014.
- [10] Martin Beck, Jonas Trümper, and Jürgen Döllner. A visual analysis and design tool for planning software reengineerings. In *VISSOFT 2011*, 2011.
- [11] Michael Balzer and Oliver Deussen. Level-of-detail visualization of clustered graph layouts. In *6th Int. Asia-Pacific Symposium on Visualization*, 2007.
- [12] Elke Franziska Heidmann, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. Visualization of Evolution of Component-Based Software Architectures in Virtual Reality. In *VISSOFT 2020*.
- [13] Wilhelm Hasselbring, Alexander Krause, and Christian Zirkelbach. Explorviz: Research on software visualization, comprehension and collaboration. *Software Impacts*, 6, 2020.
- [14] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In *ICSME 2017*.
- [15] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *ICSE 2011*, 2011.
- [16] Monica Tavanti and Mats Lind. 2d vs 3d, implications on spatial memory. In *INFOVIS 2001*, 2001.

- [17] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M Gonzalez-Barahona, and Michele Lanza. Codecity: A comparison of on-screen and virtual reality. *Information and Software Technology*, 153, 2023.
- [18] Juraj Vincur, Pavol Návrat, and Ivan Polasek. Vr city: Software analysis in virtual reality environment. In *QRS-C 2017*, 2017.
- [19] Orla Greevy, Michele Lanza, and Christoph Wyseier. Visualizing live software systems in 3d. In *Proceedings of the 2006 ACM symposium on Software visualization*, 2006.
- [20] Adrian Hoff, Christoph Seidl, Mircea Lungu, and Michele Lanza. Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2023.
- [21] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics. 2004.
- [22] Adrian Hoff, Michael Nieke, and Christoph 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.
- [23] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 119–130, Limassol, Cyprus, October 2022. IEEE.
- [24] Roy Oberhauser and Carsten Lecon. Virtual Reality Flythrough of Program Code Structures. In *Proceedings of the Virtual Reality Int. Conference - Laval Virtual 2017 on - VRIC '17*, 2017.
- [25] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, 2018.
- [26] Michael L Collard, Michael John Decker, and Jonathan I Maletic. sr-cml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International conference on software maintenance*, pages 516–519. IEEE, 2013.

- [27] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*. 2003.

Uniquifying Architecture Visualization through Variable 3D Model Generation

Originally published in: Proceedings of the 17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2023), January 25–27, 2023, Odense, Denmark

Joint work with: Christoph Seidl and Michele Lanza

Abstract

Software visualization facilitates the interactive exploration of large-scale code bases, e.g., to rediscover the architecture of a legacy system. Visualizations of software structure suffer from repetitive patterns that complicate distinguishing different subsystems and recognizing previously visited parts of an architecture.

We leverage variability-modeling techniques to “uniquify” visualizations of subsystems via custom-tailored 3D models of recognizable landmarks: For each subsystem, we derive a descriptor and translate it to a (random but deterministic) configuration of a feature model of variable 3D geometry to support large numbers of different 3D models while capturing the design language of a particular type of landmark. We devised a hybrid variant derivation mechanism using a slots-and-hooks composition system for 3D geometry as well as adjusting visual characteristics, e.g., material. We demonstrate our method by creating various different trophies as landmarks for the visualization of a software system.

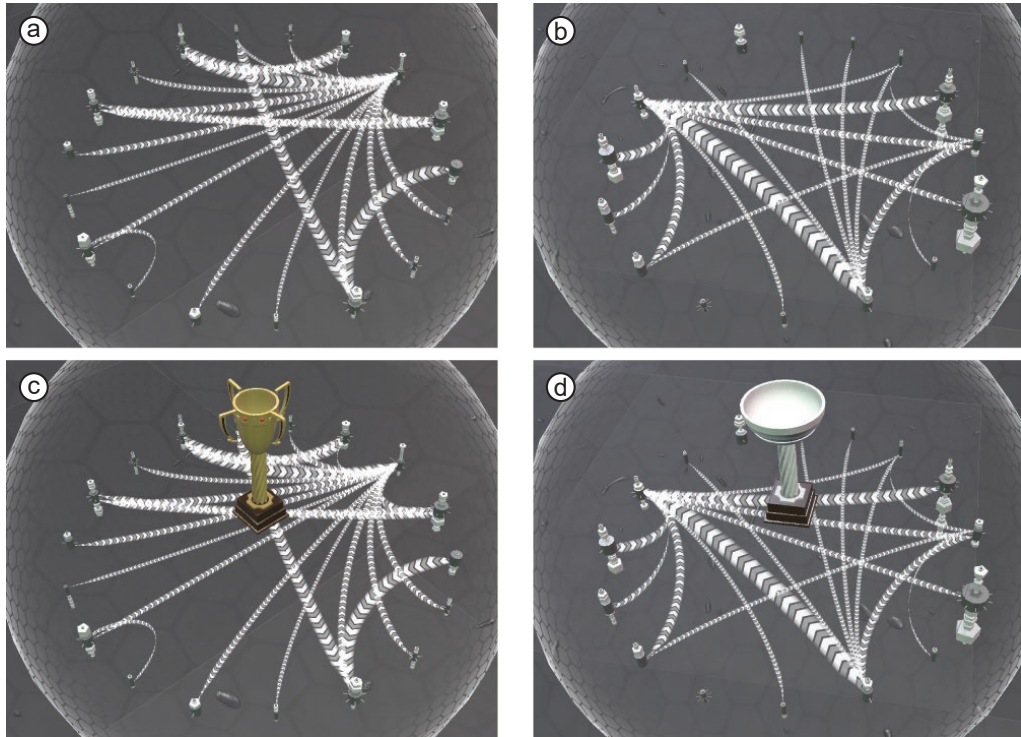


Figure D.1: Example visualization of two architectural elements suffering from hard-to-distinguish patterns (a, b) vs. the same architectural elements “uniquified” through custom-tailored landmarks (c, d).

D.1 Introduction

Software visualization represents the structure, behavior, or evolution of a software system in a visual format to foster comprehension [1]. When exploring large-scale code bases, e.g., to rediscover the design of a legacy system, visualization, esp. of structural system aspects, is essential for gaining an overview of a system’s architecture [2, 3]. However, existing software visualization [4, 5, 6, 7, 8] is hampered by a pivotal problem: The structural visualizations of different parts of a software system are hard to differentiate due to repetitive and complex patterns in the visual representation that cannot easily be distinguished (see Figure D.1).

The lengthy and iterative exploration of large-scale code bases is further complicated when comparing parts of a system or re-encountering an already visited part of a system: previously gained insights and mental models cannot be associated.

We propose a method for 3D software visualization to “uniquify” different parts of a software system by placing custom 3D models of recognizable

landmarks along the structural representation of a particular part of a software system (e.g., packages, components, subsystems). We use trophies (as awarded to winners of sports competitions) as running example for one type of landmark (but one could also imagine statues, towers, bridges etc.).

To aid the structural representation of (potentially very many) parts of a system with custom 3D landmarks, we identified two challenges: First, a large number of sufficiently distinct 3D models is required to ensure that each different part of a system can be represented by a custom landmark. In addition, each type of landmark follows a certain design language [9, 10] that governs potential variations in its appearance, which has to be incorporated in the respective variations of the 3D models.

We leverage techniques from variability engineering to unquify structural visualizations in 3D by generating distinct 3D models for a large number of landmarks following a common design language: We capture the design language of a particular landmark as configuration logic within a feature model where each feature is associated with partial 3D geometry or a visual characteristic of a 3D model, e.g., its material. We analyze the source code associated with the part of a system to be visualized and map its relevant characteristics to a configuration of the feature model. Finally, we create a custom-tailored 3D model representing one landmark via a hybrid variant derivation mechanism combining composition of partial 3D geometry with transformation of visual characteristics.

While this paper describes work in progress, we provide explanations and a prototype implementation¹ of all constituent concepts.

D.2 Background

Our work marries concepts from software visualization with techniques from variability engineering.

D.2.1 Software Visualization

Software visualization presents a software system's structure, behavior, or evolution in a visual format (using 2D and 3D visual metaphors) to foster the comprehension of structural arrangements and relations [11, 1, 12, 13].

2D metaphors are mostly abstract graphs, trees, and diagrams [14, 15, 16]. 3D metaphors borrow analogies from the physical world to exploit viewers'

¹<https://gitlab.com/immersive-software-archaeology/variable-3d-landmarks>

familiarity with real-world constructs, e.g., cities, islands, or planets [4, 5, 6, 11, 17, 7, 8]. 3D software visualizations can be distinguished by the medium they employ, i.e., computer screens, augmented reality, or virtual reality [18, 19].

A fundamental purpose of software visualization is to provide an overview of a visualized subject system, e.g., to foster top-down exploration [6, 20, 21, 22, 2, 23, 3]. This includes the visualization of architecture-level constructs such as packages or subsystems. However, software systems may encompass architecture-level constructs with similar (but not identical) characteristics and size so that the generative processes for creating visual representations yield repetitive visual structures (see Figure D.1). As a consequence, the exploration process is hampered by problems with orientation (inability to distinguish different architectural elements) and relating previously built memory models over the course of exploration (inability to recognize previously inspected parts of a system).

D.2.2 Variability Engineering

A *Software Product Line (SPL)* [24, 25] permits structured reuse within a highly-variable software family by exploiting commonalities and managing variabilities of closely related variations of software artifacts. Within an SPL, the *problem space* captures the configuration logic on conceptual level, whereas the *solution space* contains realization artifacts for all possible variants. Configuration logic is represented by a *variability model*, e.g., a *feature model*, which is a hierarchical tree of (de)selectable (optional/mandatory) features potentially structured into alternative groups permitting selection of at least one/exactly one feature (see Figure D.3). *Cross-tree constraints* (commonly formulated in propositional logic) may further reduce the configuration space. A *configuration* is a valid selection of features obeying the configuration rules imposed by the feature model from the problem space. A *variant* is the realization of a configuration as realization artifacts from the solution space.

Variant derivation translates a configuration to a variant comprising the realization of one specific product. There are three principal kinds of variant derivation [25]: *Annotative* methods prune a representation comprising all possible variations (150% model) to only the realization artifacts required for a configuration (100% model). *Compositional* methods build a variant by combining constituent pieces associated with individual features or combinations thereof. *Transformational* methods modify characteristics of an existing variant to retrieve the desired variant. While realization artifacts of the

solution space commonly consist of source code, other software artifacts can be made subject of variability within an SPL as well, e.g., in our case, 3D models.

D.3 Variable 3D Model Generation

Our method utilizes concepts from variability modeling and 3D model generation to derive "uniquifying" 3D landmarks based on a configurable design language to be placed along otherwise similar visual structures in software architecture visualizations. A prime consideration for our method is to yield 3D landmarks that are distinct, yet following a well-defined design language. In turn, this design language needs to be expressive enough to provide distinct landmarks for all architecture-level constructs of a visualized software system. Figure D.2 depicts a conceptual overview of our method.

We define a configurable design language for 3D landmarks via a feature model and partial 3D geometry ①. We automatically synthesize a software descriptor ① that we use to sample a valid configuration of the feature model ②, and forward it to a hybrid variant derivation mechanism ③ yielding a concrete 3D model for a landmark. In the following, we detail each step.

① *Visual Language Definition* We perceive—and model—the design system [26, 9] associated with a design language in the sense of an SPL: While the individual visual characteristics of a 3D design language manifest in the implementation as 3D models, the (as of yet implicit) rules and variations permissible within a design language constitute a form of configuration logic. In consequence, the solution space (see Section D.2.2) is comprised of individual parts of 3D geometry (partial 3D models). In addition, the problem space consists of a feature model that explicates design rules in the form of permissible configurations described as (de)selectable features. The left part of

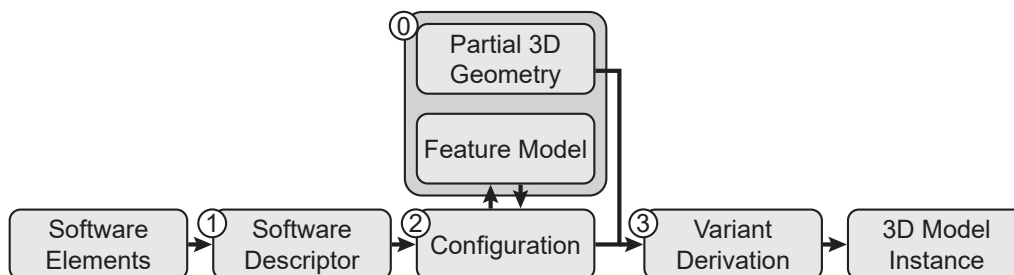


Figure D.2: Overview of our method for generating "uniquifying" landmarks from a software descriptor.

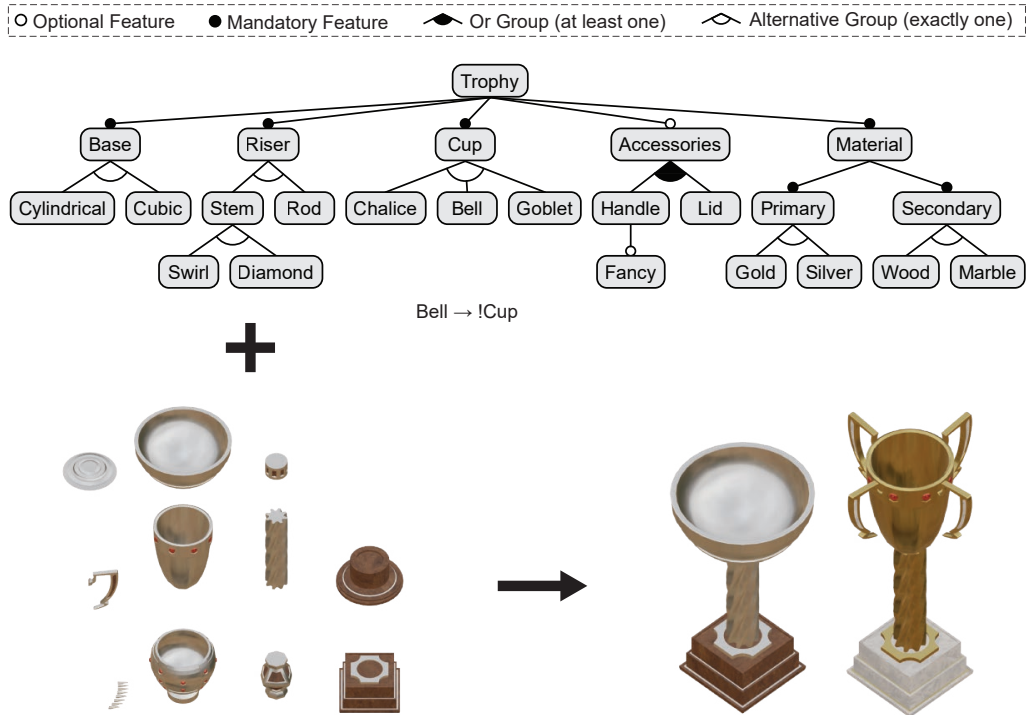


Figure D.3: A feature model capturing the configuration rules of a design language (problem space) is aided with partial 3D geometry (solution space) to retrieve custom-tailored 3D models of landmarks via a hybrid variant derivation procedure composing 3D geometry and transforming visual characteristics, e.g., material.

Figure D.3 depicts these two artifacts using the example of configurable trophies. The feature model defines *which design elements* may be combined and the partial 3D geometry defines *how individual parts of a 3D model* may be combined.

To allow the combination of partial 3D geometry according to imposed rules, we have designed a *slots-and-hooks composition system* [27] for 3D models (see Figure D.4). On a conceptual level, a *slot* serves as an extension point for 3D geometry that defines where and with which rotation/scale an element may be attached. Likewise, a *hook* (defined in another part of 3D geometry) declares the principle option for serving as an extension to a compatible slot. Whether a slot of part of 3D geometry is actually bound (at all) and, if so, with the compatible hook of which specific other part of 3D geometry is determined via a configuration of the feature model during variant derivation (see below). On a practical level, we implement the slots-and-hooks composition system for 3D models by using invisible elements of 3D geometry as markers to define position, rotation, and scale for both slots and hooks. We

established a naming convention that identifies these markers as either slot or hook and determine compatibility via matching names.

① *Software Descriptor Synthesis* The landmarks we generate have to be created deterministically from (the implementation of) an architectural element and, at the same time, must be stable over miniscule changes to the implementation. To achieve these goals and steer the subsequent creation of a configuration (see below), we borrow the concept of a *descriptor* from computer vision: A visual descriptor is an abstract summary of visual characteristics in an image [28] (e.g., edges) that can be *sensitive* to certain characteristics (e.g., color changes) and *robust* against others (e.g., rotation).

We adapt the concept of a visual descriptor to design a *software descriptor* for architectural elements: The software descriptor should be robust against minor changes (e.g., creating a new attribute or method within a class), yet sensitive toward major modifications (e.g., removing entire classes). In addition, in our use case, the software descriptor steers the sampling of a configuration for a 3D model and the respective landmarks should be stable for each architectural element. Hence, creation of the software descriptor must be deterministic, i.e., the same input of software elements must always result in the same descriptor.

While determining the full scope of an expressive software descriptor is part of our ongoing work, we illustrate the principle use of a software descriptor: For each architectural element, we calculate a software descriptor that is sensitive to the number of contained elements (e.g., classes, structs, etc.) as well as their respective qualified names. We devise a canonical form of the descriptor by combining the number of elements with a hash of all fully qualified (i.e., unambiguously identifiable) names in alphabetical order. As an example, consider a Java package `com.application` consisting of 3 classes `Model`, `View`, and `Controller`. Our illustrative software descriptor for this package starts with a 3 (number of contained elements) followed by a hash of their concatenated fully qualified names resulting in “32117573461” as the software descriptor.

Through this procedure, we capture relevant high-level aspects of software constructs contained within an architectural element while abstracting from negligible details within a deterministic and compact representation.

② *Steered Random Configuration Sampling* We derive a valid configuration from the visual design language based on configuration sampling guided by our software descriptor. Similar to the synthesis of software descriptors, the

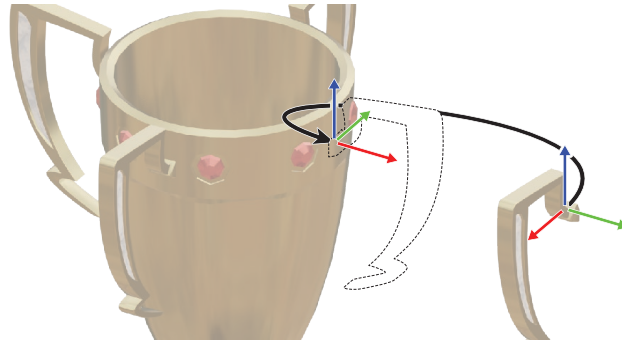


Figure D.4: Example of the compositional aspect of our variant derivation. A slots-and-hooks composition mechanism combines partial geometry based on a naming convention of the respective parts.

sampling of configurations must be deterministic, i.e., the same descriptor should always result in the same configuration. At the same time, different software descriptors should result in different configurations. However, which specific configuration is determined for a particular software descriptor is of no concern (as each represents a viable landmark). Hence, we employ random configuration sampling but steer the selection process by using our software descriptor as seed for the random generator to ensure deterministic results. In our implementation¹, we use the random sampling provided by FeatureIDE².

③ *Hybrid Variant Derivation* To realize a configuration in the form of a variant comprised of a custom-tailored 3D model, we have devised a hybrid variant derivation mechanism that includes aspects of compositional and transformational methods.

The compositional part of variant derivation employs the slots-and-hooks mechanism recursively by stepping through relevant partial 3D geometry and composing parts according to matches in the contained slots and hooks. For instance, for each of the four handles of the trophy variant depicted in Figure D.4, the hook (and, therefore, the associated 3D geometry) is positioned, rotated, and scaled according to the respective slot defined in this specific kind of cup. The order for composing slots and hooks may be arbitrary as each will result in an identical 3D model.

The transformational part of variant derivation mutates the appearance of 3D geometry by revisiting all (previously) partial 3D geometry and exchanging materials. For that purpose, the variant derivation mechanism uses a selection of pre-configured materials by matching their names according to a fea-

²<https://featureide.github.io/>



Figure D.5: Example of multiple visually distinct trophies created through variant derivation for our running example.

ture naming convention. In our running example, we use two different types of materials that are (primarily) visible with metallic parts of a trophy (riser, cup, handles, lid) or with the base of the trophy, where each type of material has two possible variations (see Figure D.1).

The result of variant derivation is a concrete 3D landmark model that developers of 3D software architecture visualizations can utilize to "uniquify" architecture-level constructs similar to the examples depicted in Figure D.5.

D.4 Outstanding Challenges

While we have devised a prototype implementation to demonstrate our concepts, there are outstanding challenges, which we list here to foster academic discourse and invite potential collaborators.

Unique Configuration for each Descriptor Even though our mapping of a software descriptor to a configuration of the feature model is deterministic, there may be cases when two distinct descriptors are mapped to the same configuration and, thus, yield the same 3D model. In part, this is (inadvertently) by design as the number of configurations in the problem space may be lower than the potential number of different architecture descriptors each representing a distinct individual architectural element. To address this challenge, we foresee two promising directions: On conceptual level, we strive to a-priori assess whether the configuration space is sufficiently large to accommodate all possible architectural elements and, if not, to guide expansion of the configuration space (e.g., in the example shown in Figure D.3, that adding a new type of lid would add 144 configurations). On practical level, we will

explore configuration options via attributes and with continuous values (e.g., for model scale or custom colors) to further expand the configuration space with little burden of creating new elements for 3D models.

Continuous Variation Software visualization commonly incorporates relations between architectural elements in the spatial arrangement of the visualization [11], e.g., hierarchies of nested components or the degree of coupling between different subsystems. In consequence, architectural elements that are "more strongly" related to each other are commonly collocated in the visualization. Currently, our method creates different configurations and, thus, visually distinct 3D models for each architectural element. However, we see great potential in exploiting the relation of architectural elements by having landmarks of closely related architectural elements share certain visual characteristics (e.g., an area where all landmarks have a gold material). For this purpose, we will extend the software descriptor to incorporate hierarchy and relation of elements but will also research how to selectively vary a configuration to achieve continuous variation for the generated 3D models.

Robustness toward Evolution While our tentative implementation of an architecture descriptor is robust against changes in implementation details (e.g., methods and attributes), it is sensitive to changes in the set of classes contained in an architectural element and names of the contained classes. However, over the course of software evolution, even these characteristics may change. Currently, this would yield a different configuration and, thus, an entirely different 3D model for a landmark so that the relation to a previously established mental model will be harmed. To cope with evolutionary changes, we strive to either make the software descriptor robust toward certain architectural changes, so that associated landmarks remain unchanged, or determine a method that allows creation of largely similar configurations to that created landmarks contain only minuscule visual differences to the previous edition.

Landmarks with Semantics In our prototype implementation, we sample 3D models based on a random configuration of a feature model using a software descriptor. While this allows us to deterministically generate a large variety of 3D models that follow a configurable design language, the resulting 3D model variants do not convey characteristics of represented software elements such as their size (e.g., by scaling landmarks), quality (e.g., by including visual ef-

fects such as cracks, cf. [29]), or complexity (e.g., by selecting certain features that indicate complexity such as the complex handle in our running example).

D.5 Conclusion & Future Work

In this paper, we exploited variability engineering techniques to devise a method for generating custom-tailored 3D models of landmarks to “uniquify” otherwise visually similar structures in software architecture visualizations. Our method allows to encode the design language of a family of 3D models within a feature model while also permitting to leverage 3D modeling software to prepare visually appealing (partial) 3D models.

The outstanding challenges of our work are rooted within two fundamental areas: descriptor robustness and variant similarity. *Descriptor robustness*: Our joint goals of providing unique 3D models for landmarks “uniquifying” conceptually different architectural elements and ensuring a visual similarity between closely related architectural elements are, at times, diametrically opposed. Accommodating both goals requires determining additional characteristics to incorporate in the descriptor as well as finding a tradeoff between characteristics the descriptor should be sensitive to/robust against. *Variant similarity*: While there are methods and measures for determining the similarity of different configurations (i.e., selections from the feature model), our goal of achieving visual similarity/continuity (for various use cases) requires the ability to prognose similarity of multiple variants (e.g., visual similarity of 3D models), ideally, without having to rely on analyzing resulting products. Our future work is aimed at researching solutions for descriptor robustness and variant similarity to further improve the benefits of our method for demarcating/recognizing architectural elements in the visualization of even larger and evolving software systems.

Bibliography

- [1] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.
- [2] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and evolution of software architectures. In *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning*,

- Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.
- [3] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020.
 - [4] Elke Franziska Heidmann, Lynn von Kurnatowski, Annika Meinecke, and Andreas Schreiber. Visualization of evolution of component-based software architectures in virtual reality. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 12–21. IEEE, 2020.
 - [5] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922, 2008.
 - [6] Adrian Hoff, Michael Nieke, and Christoph 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*, pages 1455–1458, 2021.
 - [7] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. Cityvr: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637. IEEE, 2017.
 - [8] Sazzadul Alam, Sandro Boccuzzo, Richard Wettel, Philippe Dugerdil, Harald Gall, and Michele Lanza. Evospaces-multi-dimensional navigation spaces for software evolution. In *Human Machine Interaction*, pages 167–192. Springer, 2009.
 - [9] Richard Poulin. *The Language of Graphic Design Revised and Updated: An Illustrated Handbook for Understanding Fundamental Design Principles*. Rockport Publishers Inc., 2018.
 - [10] Tim McCreight. *Design Language, Interpretive Edition*. Brynmorgen Press, 2006.
 - [11] Alfredo R Teyseyre and Marcelo R Campo. An overview of 3d software visualization. *IEEE transactions on visualization and computer graphics*, 15(1):87–105, 2008.

- [12] M-AD Storey, F David Fracchia, and Hausi A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.
- [13] Richard Müller and Dirk Zeckzer. Past, present, and future of 3d software visualization—a systematic literature analysis. In *International Conference on Information Visualization Theory and Applications*, volume 2, pages 63–74. SCITEPRESS, 2015.
- [14] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with softwareaut. *Science of Computer Programming*, 79:204–223, 2014.
- [15] Mohammad Alnabhan, Awni Hammouri, Mustafa Hammad, Mohammad Atoum, and Omamah Al-Thnebat. 2d visualization for object-oriented software systems. In *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, pages 1–6. IEEE, 2018.
- [16] Blaine A Price, Ian S Small, and Ronald M Baecker. A taxonomy of software visualization. In *Proc. 25th Hawaii Int. Conf. System Sciences*, 1992.
- [17] Andrian Marcus, Louis Feng, and Jonathan I Maletic. 3d representations for software visualization. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–ff, 2003.
- [18] Leonel Merino, Johannes Fuchs, Michael Blumenschein, Craig Anslow, Mohammad Ghafari, Oscar Nierstrasz, Michael Behrisch, and Daniel A Keim. On the impact of the medium in the effectiveness of 3d software visualizations. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 11–21. IEEE, 2017.
- [19] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M Gonzalez-Barahona, and Michele Lanza. Codecity: A comparison of on-screen and virtual reality. *Information and Software Technology*, 153:107064, 2023.
- [20] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. In *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022.
- [21] Sheelagh Carpendale and Yaser Ghanam. A survey paper on software architecture visualization. Technical report, University of Calgary, 2008.

- [22] Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: A survey. *IEEE transactions on visualization and computer graphics*, 17(7):913–933, 2010.
- [23] Anna-Liisa Mattila, Petri Ihantola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Vääätäjä. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, pages 262–271, 2016.
- [24] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.
- [25] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
- [26] Sarrah Vesselov and Taurie Davis. *Building Design Systems: Unify User Experiences through a Shared Design Language*. Apress, 2019.
- [27] Uwe Aßmann. *Invasive Software Composition*. Springer, 2003.
- [28] Jianxin Wu and Jim M Rehg. Centrist: A visual descriptor for scene categorization. *IEEE transactions on pattern analysis and machine intelligence*, 33(8):1489–1501, 2010.
- [29] Johann Mortara, Philippe Collet, and Anne-Marie Pinna-Dery. Customizable visualization of quality metrics for object-oriented variability implementations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*, pages 43–54, 2022.

Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality

Originally published in: Proceedings of the 39th IEEE Working Conference on International Conference on Software Maintenance and Evolution (ICSME 2023), October 1–6, Bogotá, Colombia

Joint work with: Christoph Seidl, Mircea Lungu, and Michele Lanza

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 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 concept 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 with 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.

Video Demonstration – <https://youtu.be/NKC5YpH3n4Y>

E.1 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, 7, 8, 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 [12, 13, 11]. 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 [16, 10].

We present a method for extending existing VR software visualizations with virtual whiteboards for creating freehand sketches on a system's structure (see Figure E.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.

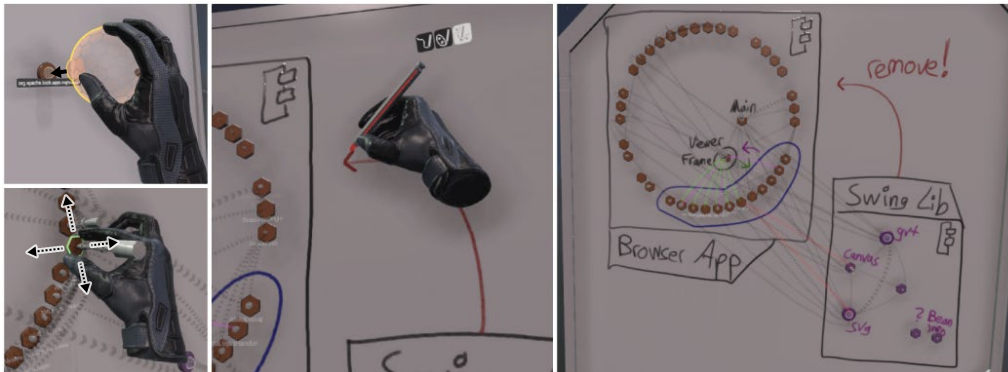


Figure E.1: Freehand Sketching in VR.

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.

E.2 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.

E.2.1 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, 19, 20]), whereas 3D metaphors range from being abstract (e.g., 3D graphs [21, 22, 23]) to real-world inspired with the information city [24, 25, 26, 27, 28, 29, 30, 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.

E.2.2 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 [35, 6]. 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 Murphy'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, 40, 41, 42, 43, 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.

E.2.3 Freehand Sketching on Whiteboards and Paper

Engineers value flexibility when creating diagrams on their software systems [16, 13, 12], 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 [16, 10], especially in early stages of planning, where engineers deliberately improvise rough sketches to ad-hoc capture thoughts [45, 13, 10, 12]. 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, 50, 47]. 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, 52, 53, 54]. Tools exist that interpret sketches to detect elements from certain visual notations (UML) [55, 52, 56, 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.

E.3 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 E.2, which we discuss throughout this section.

Figure E.3 shows an example implementation of our method in an exist-

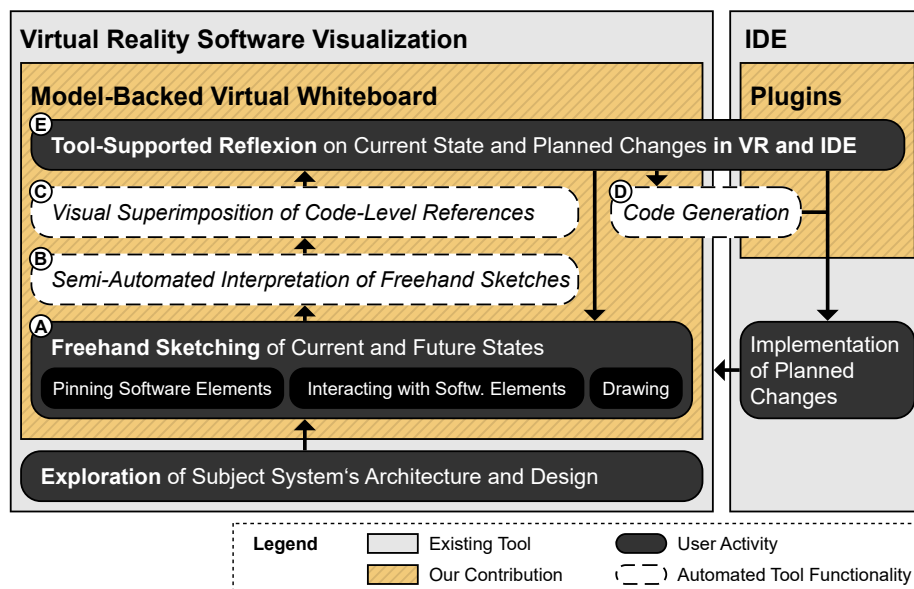


Figure E.2: Overview of our method for preparing software re-engineering via model-backed freehand sketches in a VR visualization.

ing VR software visualization¹. 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 (A). 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 (B) 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 (C), which supports engineers in reflecting on their sketches as well as on the structures they describe (E), 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 (D). We elaborate on each of these steps.

E.3.1 (A) Freehand Sketching of Current and Future States

From a user's perspective in VR, our method consists of a virtual whiteboard (Figure E.3) on which engineers pin software elements and draw sketches on, similar to how they would on a physical whiteboard (A in Figure E.2). 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 E.3 (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

¹<https://gitlab.com/immersive-software-archaeology>

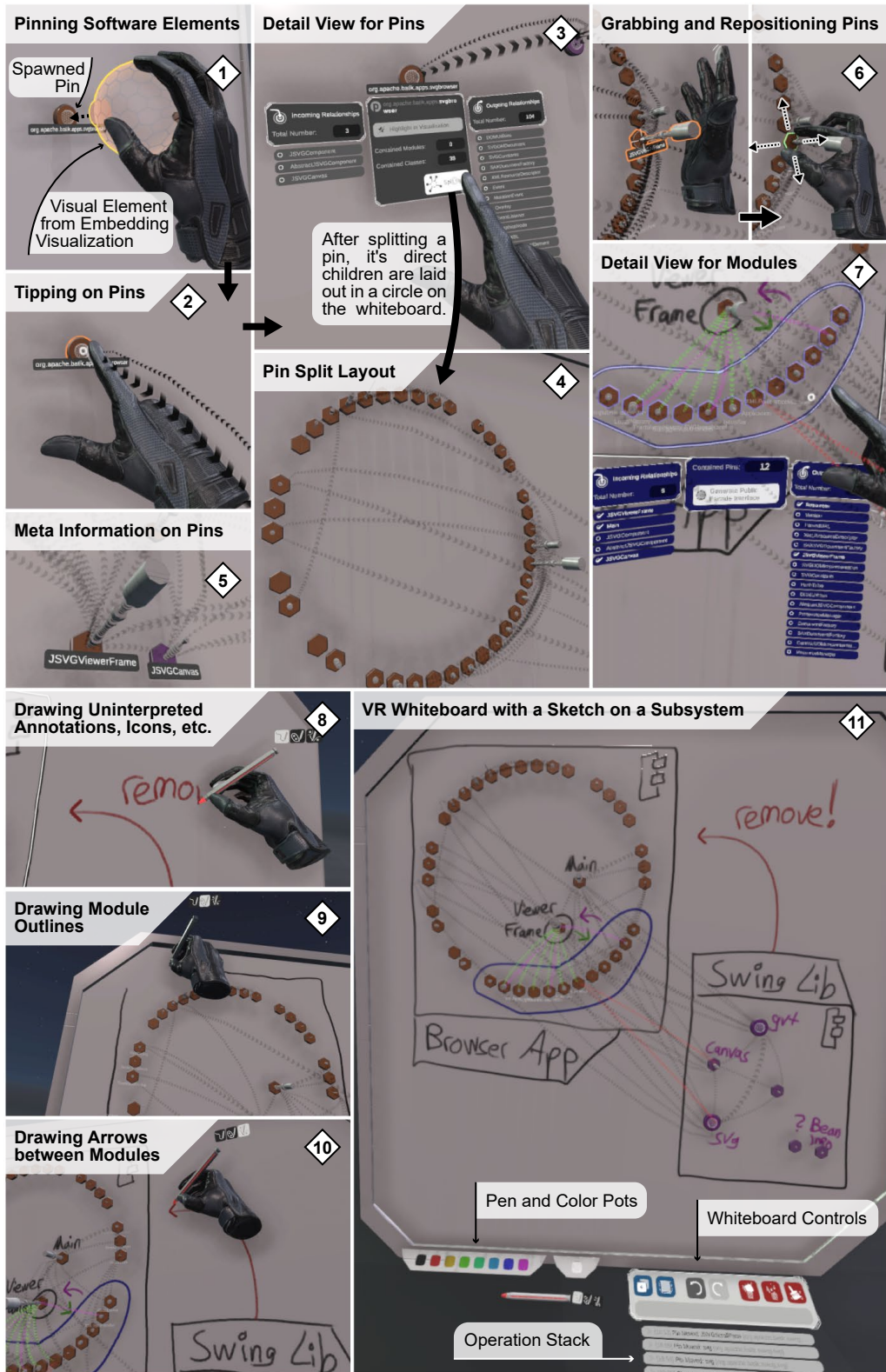







Figure E.3: Screenshots of an implementation of our method in an existing VR software visualization. The embedding visualization is not depicted.


in software visualizations usually encodes relevant metrics, avatars on pins carry this information, too.

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 . 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 ( and ) 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 ( and ) 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 E.3  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 $\diamond 5$. 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 E.3 $\diamond 3$ and $\diamond 7$: 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.

E.3.2 \textcircled{B} Semi-Automated Interpretation of Freehand Sketches

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

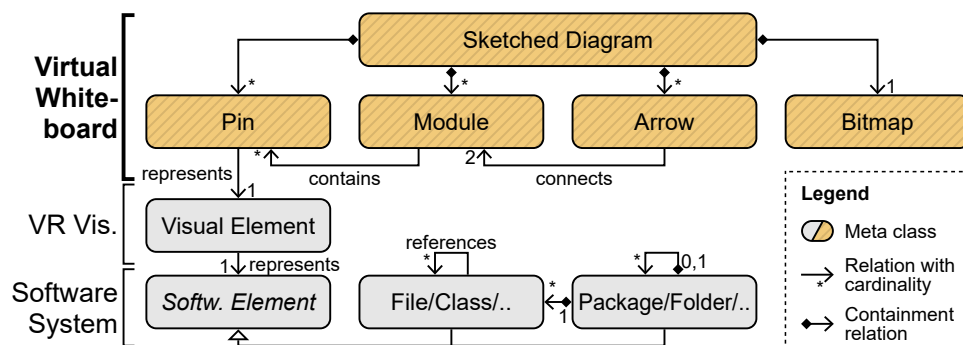


Figure E.4: Meta model for the sketched diagram structure of our method.

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 E.2) 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 ⋈

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 ⋈

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 E.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 E.3, 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 E.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 E.3 ) – 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 E.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 E.3 ). 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.

E.3.3 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 E.2). Figure E.3  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 E.3). 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 E.3.2), 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 E.3 , 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 E.2), 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.

E.3.4 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 E.2). These operations are based on the model structure of a plan (see Figure E.4), allowing for coarse-grained operations, such as the generation of an interface for a freehand drawn module.

E.4 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.

RQ₁ How does VR freehand sketching support engineers in representing architecture-level software structures?

RQ₂ 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 (Switzerland, Denmark, and Germany).

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 E.5 depicts this process.

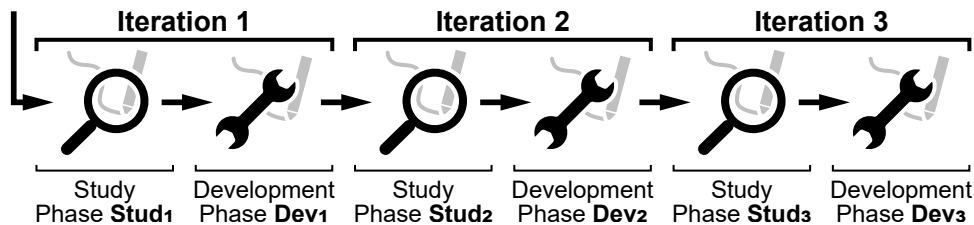


Figure E.5: Structure of our iterative evaluation. We alternated between study phases where we tested our method with participants in a case study and development phases where we improved and extended it.

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

E.4.1 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 E.1 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 appendix².

In Task₁, 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.

In Task₂, we asked participants, first, to investigate the relationship of the example application investigated in Task₁ 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 Task₁ and the package it builds upon.

In Task₃, we asked participants to reflect on the change they had planned in Task₂ via a change impact analysis.

²<https://doi.org/10.6084/m9.figshare.22710490>

Table E.1: 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

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 appendix².

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 appendix². 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 E.6 depicts a graphical representation of our results after Iteration 3.

E.4.2 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 E.4.3. More detailed descriptions of changes with a mapping to verbatim statements of participants can be found in our online appendix².

In the following, we elaborate on our technical implementation¹, which we provided to participants.

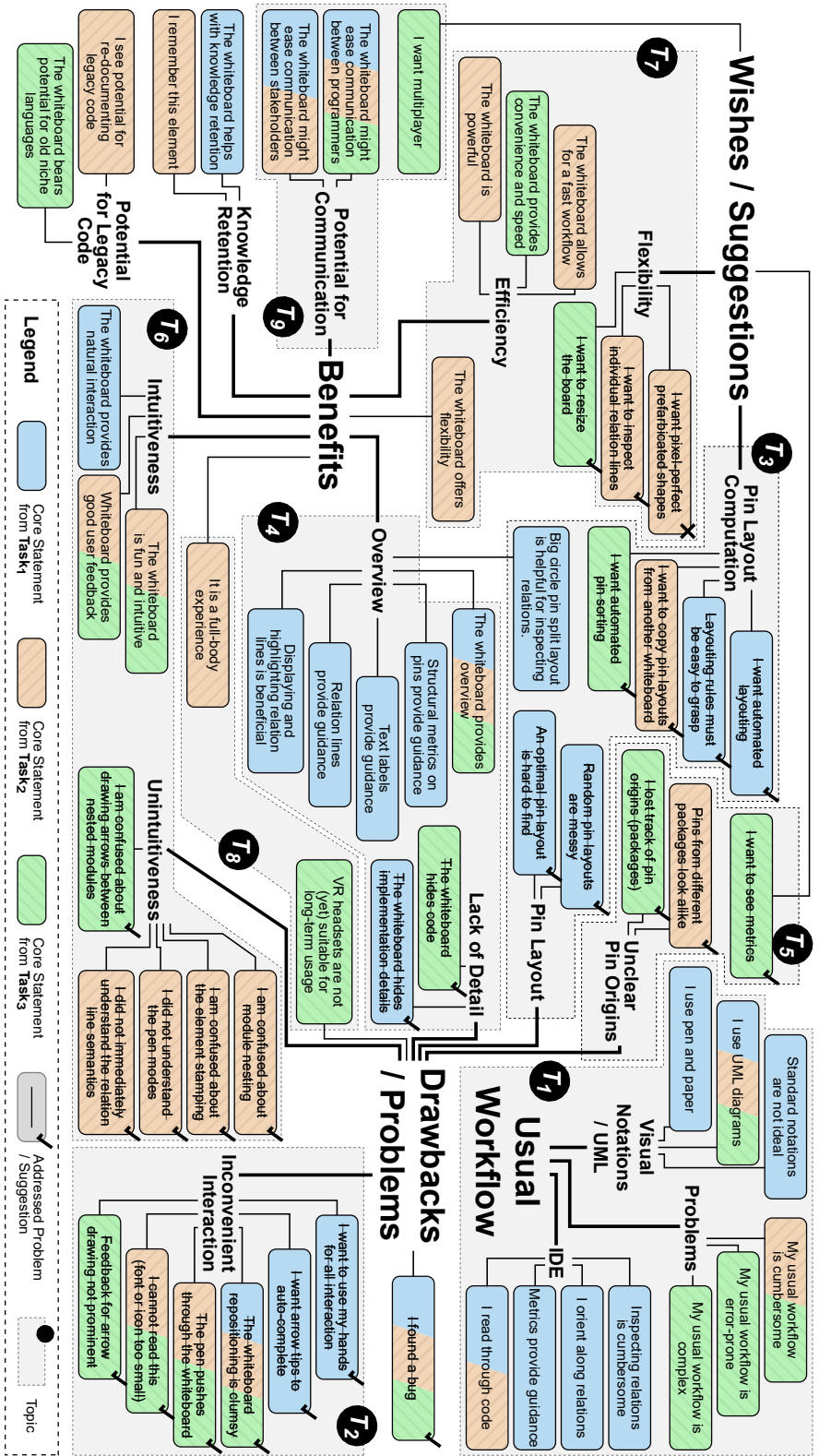


Figure E.6: Overview of participants' core statements during the three study phases organized into a hierarchy of categories and recurring topics. Each core statement summarizes one or more verbatim statements. A complete table with the mapping from statements to topics is available in our online appendix².

Implementation

We integrated the concepts presented in Section E.3 into the open-source VR software visualization tool “Immersive Software Archaeology” (ISA)¹. Figure E.3 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.

E.4.3 Participants’ Answers

During the analysis of our interview transcripts (see Section E.4.1), we extracted recurring topics in participants replies to our questions. In the following, we elaborate on these (T₁-T₉). 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 E.6 provides an overview of the core statements made by the participants, grouped into categories and summarized into topics (T₁-T₉). 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).

T₁) Usual Tools Are Not Ideal

A majority of participants stated that their usual approach for solving Task₁ (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 Task₂ (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 Task₃ (impact analysis for the changes planned in Task₂), 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 Task₃ “*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.*”

T₂) 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 appendix² 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 Dev₂ 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 it.

T₃) 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 E.3). 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 Dev₂ which, in comparison with strategy (a), was again assessed as beneficial in Iteration 3 (Task₁): *“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.’”*

T₄) 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 Task₂ 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 Task₃ (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, Task₁) 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, Task₁)

T₅) 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 Dev₁ by displaying a small avatar of the respective represented visual element on each pin (see Section E.3.1). 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 Task₁, 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 Dev₁ by coloring pins according to the subsystem they stem from (see Section E.3). It was not brought up in subsequent iterations.

T₆) 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 (Task₂ and Task₃) 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 E.3), 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 Task₃. 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).

T₇) 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 Task₂, 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 Task₃: “*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.*”

T₈) 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.”*

T₉) 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.

E.4.4 Answers to Research Questions

We use the results presented in T₁-T₉ above to answer our research questions in the following, highlighting both advantages and disadvantages.

RQ₁: 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 (T₂), requests for improved pin layout algorithms (T₃), suggestions for more meta information on pinned elements (T₅), and more unintuitive functionality (T₆). In

Dev₁ and Dev₂ (Figure E.5), we addressed all points that emerged directly from the technical implementation (e.g., problems with the VR controls, T₂) 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, T₅).

Positive feedback and perceived strengths of our method related to RQ₁ were its intuitiveness (T₆) and its high degree of flexibility (T₇).

The workflow of pinning software elements on a diagram and simply drawing on it was perceived as powerful and efficient (T₇): *“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 Task₂, because participants were able to freely position, outline, highlight, and annotate software elements and their inter-relations. Other key features were those that allowed participants to navigate along the software hierarchies in a sketch, i.e., splitting pins for architectural units (T₃) while maintaining an overview of the pins’ origins (T₅).

RQ₂: 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”, T₆). 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 (T₄) 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 (T₄), programming language independence, and communication (T₉). 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”* (T₄).

The majority of comments relevant for RQ₂ 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.4.5 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 Stud₁, 3 in Stud₂, and 1 in Stud₃. Having only one participant in Stud₃ entails a risk to the evaluation of changes made in Dev₂. However, because we did not conceptually change our method in Dev₂, this risk pertains only to implementation aspects, mostly regarding interaction problems noted in Stud₂ for concepts developed and implemented in Dev₁.

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 guide², 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.

E.5 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 of 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.

Bibliography

- [1] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [2] Martin Beck, Jonas Trumper, and Jurgen Dollner. A visual analysis and design tool for planning software reengineerings. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, September 2011.
- [3] Linda H Rosenberg and Lawrence E Hyatt. Software re-engineering. *Software Assurance Technology Center*, 1996.
- [4] Rainer Koschke and Daniel Simon. *Hierarchical Reflexion Models*. December 2003.
- [5] Welf Lowe, Morgan Ericsson, Jonas Lundberg, and Thomas Panas. Software Comprehension Integrating Program Analysis and Software Visualization.
- [6] Jim Buckley, Nour Ali, Michael English, Jacek Rosik, and Sebastian Herold. Real-time reflexion modelling in architecture reconciliation: A multi case study. *Information and Software Technology*, 61, 2015.
- [7] Jens Knodel and Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, January 2007.
- [8] Gail C Murphy, David Notkin, and Kevin 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)*, pages 277–280. IEEE CS Press, 2015.

-
- [10] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy 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*, April 2007.
 - [11] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, May 2006.
 - [12] Nicolas Mangano, Thomas D LaToza, Marian Petre, and André van der Hoek. How software designers interact with sketches at the whiteboard. *IEEE Transactions on Software Engineering*, 41(2), 2014.
 - [13] Uri Dekel and James D Herbsleb. Notation and Representation in Collaborative Object-Oriented Design: An Observational Study. 2007.
 - [14] Barbara Tversky. What do Sketches say about Thinking? 2002.
 - [15] Yin Yin 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] Sebastian Baltes and Stephan Diehl. Sketches and Diagrams in Practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2014.
 - [17] Richard Müller and Dirk 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*, 29(9), September 2003.
 - [19] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwrenaut. *Science of Computer Programming*, 79, January 2014.
 - [20] Roberto Minelli and Michele Lanza. SAMOA A Visual Software Analytics Platform for Mobile Applications. In *2013 IEEE International Conference on Software Maintenance*, September 2013.

- [21] M. Balzer and O. Deussen. Hierarchy Based 3D Visualization of Large Software Structures. In *IEEE Visualization*, 2004.
- [22] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *6th International Asia-Pacific Symposium on Visualization*, 2007.
- [23] Jonathan Maletic. Visualizing Object-Oriented Software in Virtual Reality. 2001.
- [24] Rodrigo Brito, Aline Brito, Gleison Brito, and Marco Tulio Valente. GoCity: Code City for Go. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, February 2019.
- [25] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, September 2013.
- [26] Michele Lanza, Harald Gall, and Philippe Dugerdil. EvoSpaces: Multi-dimensional Navigation Spaces for Software Evolution. In *2009 13th European Conference on Software Maintenance and Reengineering*, 2009.
- [27] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. CityVR: Gameful Software Visualization. 2017.
- [28] Juraj Vincur, Pavol Navrat, and Ivan Polasek. VR City: Software Analysis in Virtual Reality Environment. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017.
- [29] Richard Wettel and Michele Lanza. Visualizing Software Systems as Cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, June 2007.
- [30] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, 2008.
- [31] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, 2011.

- [32] Adrian Hoff, Michael Nieke, and Christoph 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] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94, August 2014.
- [34] G.C. Murphy, D. Notkin, and K.J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4), April 2001.
- [35] G.C. Murphy and D. Notkin. Reengineering with reflexion models: a case study. *Computer*, 30(8), August 1997.
- [36] Christopher Ackermann, Mikael Lindvall, and Rance Cleaveland. Towards Behavioral Reflexion Models. In *2009 20th International Symposium on Software Reliability Engineering*, November 2009.
- [37] Sebastian Herold, Steve Counsell, Michael English, Mel Ó Cinnéide, and Jim Buckley. *Detection of Violation Causes in Reflexion Models*. February 2015.
- [38] Marcello Romanelli, Andrea Mocci, and Michele Lanza. Towards Visual Reflexion Models. In *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015.
- [39] E.L.A. Baniassad and G.C. Murphy. Conceptual module querying for software reengineering. In *Proceedings of the 20th International Conference on Software Engineering*, April 1998.
- [40] Roberto Almeida Bittencourt, Gustavo Jansen de_Souza Santos, Dalton Dario Serey Guerrero, and Gail C. Murphy. Improving Automated Mapping in Reflexion Models Using Information Retrieval Techniques. In *2010 17th Working Conference on Reverse Engineering*, October 2010.
- [41] A. Le Gear, J. Buckley, J.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.*, November 2005.

- [42] Tobias Olsson, Morgan Ericsson, and Anna Wingkvist. To automatically map source code entities to architectural modules with Naive Bayes. *Journal of Systems and Software*, 183, 2022.
- [43] Zipani Tom Sinkala and Sebastian Herold. InMap: automated interactive code-to-architecture mapping. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, March 2021.
- [44] Zipani Tom Sinkala and Sebastian Herold. Towards Hierarchical Code-to-Architecture Mapping Using Information Retrieval. 2021.
- [45] Christopher D. Hundhausen. Using end-user visualization environments to mediate conversations: a Communicative Dimensions framework. *Journal of Visual Languages & Computing*, 16(3), June 2005.
- [46] James D Herbsleb, Bell Laboratories, and Shuman Boulevard. Metaphorical representation in collaborative software engineering. 1999.
- [47] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. Supporting informal design with interactive whiteboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, April 2014.
- [48] Masaki Suwa and Barbara Tversky. External Representations Contribute to the Dynamic Construction of Ideas. In G. Goos, J. Hartmanis, J. van Leeuwen, Mary Hegarty, Bernd Meyer, and N. Hari Narayanan, editors, *Diagrammatic Representation and Inference*, volume 2317. 2002.
- [49] Masaki Suwa, John Gero, and Terry Purcell. Unexpected discoveries and S-invention of design requirements: important vehicles for a design process. *Design Studies*, 21(6), November 2000.
- [50] Donald A Schön. *The reflective practitioner: How professionals think in action*. 2017.
- [51] Wendy Ju, Brian A. Lee, and Scott R. Klemmer. Range: exploring implicit interaction through electronic whiteboard design. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, November 2008.
- [52] Tek-Jin Nam. Collaborative Design Prototyping Tool for Hardware Software Integrated Information Appliances. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C.

- Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Randall Shumaker, editors, *Virtual Reality*, volume 4563. 2007.
- [53] Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, and Anthony 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] Justin Luo. VR-Notes: A Perspective-Based, Multimedia Annotation System in Virtual Reality. 2020.
- [55] Ronald Chung, Petrut Mirica, and Beryl Plimmer. *InkKit: a generic design tool for the tablet PC*. January 2005.
- [56] John Grundy and John 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] Christian Heide Damm, Klaus Marius Hansen, and Michael Thomsen. Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. 2000.
- [58] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022.

Collaborative Software Exploration with Multimedia Note Taking in Virtual Reality

Originally published in: Proceedings of the 46th International Conference on Program Comprehension (ICPC 2024), April 15–16, Lisbon, Portugal

Joint work with: Mircea Lungu, Christoph Seidl, and Michele Lanza

Abstract

Exploring and comprehending a software system, e.g., as preparation for its re-engineering, is a relevant, yet challenging endeavour often conducted by teams of engineers. Collaborative exploration tools aim to ease the process, e.g., via interactive visualizations in virtual reality (VR). However, these neglect to provide engineers with capabilities for persisting their thoughts and findings.

We present an interactive VR visualization method that enables (distributed) teams of engineers to collaboratively (1) explore a subject system, while (2) persisting insights via free-hand diagrams, audio recordings, and in-visualization VR screenshots.

We invited pairs of software engineering practitioners to use our method to collaboratively explore a software system. We observed how they used our method and collected their feedback and impressions before replaying their findings to the original developers of the subject system for assessment.

Video Demonstration – <https://youtu.be/32EIp4V3b4>

F.1 Introduction

Understanding a software system is a crucial task, frequently undertaken by groups of engineers [1]. For instance, it plays a vital role when preparing for the re-engineering of legacy software systems [2, 3], or for integrating a new member into a team. Nevertheless, the sheer size and complexity of a subject system or the engineers' proficiency with a programming language can hinder the software comprehension process. When attempting to explore and comprehend a software system by solely studying its source code, gaining an overview of its structure can be an intricate task.

Software visualization tools are available to assist engineers in gaining a comprehensive overview of a software system. These tools visually portray various facets of a system's structure, behavior, or evolution, offering an abstraction from the actual source code [4, 5, 6]. However, there are limited options for software visualizations that facilitate collaborative exploration. This becomes increasingly relevant in distributed developer teams [7, 8].

VR software visualization is an appropriate domain for collaboration, which is why state-of-the-art tools support it [9, 10]. However, existing approaches are limited, because they do not allow developers to take notes, posing a risk of valuable insights being lost.

We present a method that combines software exploration and note-making within a VR setting (see Figure E1). Engineers are immersed in an interactive 3D visualization of the source code of a subject system, facilitating collaborative engagement and comprehension of its structure. To document observations and insights in real-time, engineers have the capability to create virtual multi-media whiteboards (inspired by the work of Hoff et al. [11]). These serve as a medium for pinning software elements from the visualization, drawing freehand diagrams with tool assistance, jotting down notes, as well as pinning recorded audio notes and captured in-visualization screenshots of the VR surroundings.

Additionally, these virtual whiteboards remain accessible and interactive outside the VR environment through synchronization with an Integrated Development Environment (IDE), enabling further examination and interaction, e.g., direct access to source code.

We used our method in an exploratory case study to investigate and report on how practitioners engage with VR tools for collaborative software exploration and, based on that, provide lessons learned for developers of similar tools. That is, we explored how engineers engage with exploration methods akin to ours.

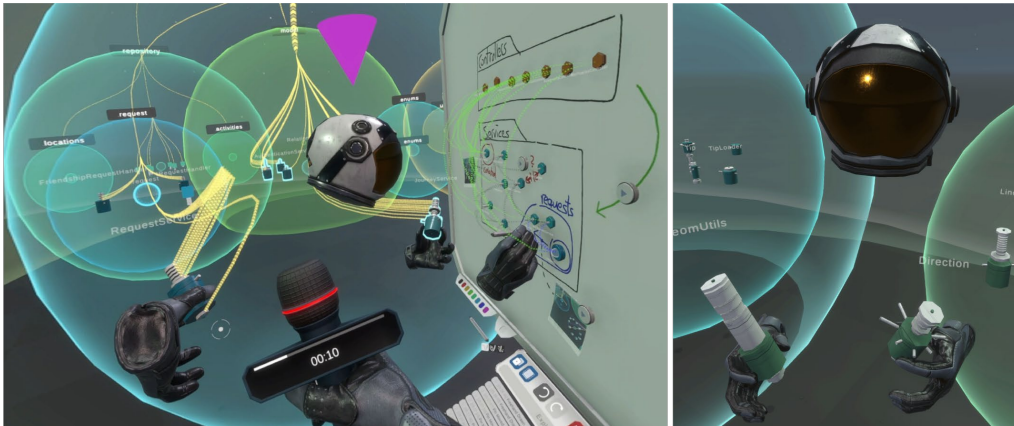


Figure F.1: Screenshot taken from a user's point of view while exploring a system. A collaborator takes notes on a virtual whiteboard.

- RQ₁** How do engineers explore and take notes of a software system in a collaborative VR software exploration and note taking tool?
- RQ₂** What strengths and weaknesses do engineers perceive in using a collaborative VR software exploration and note taking tool?

Further, we studied the results of engineers' explorations to provide first impressions on the suitability of the approach.

- RQ₃** What type of insights do engineers extract from a system when using a collaborative VR software exploration and note taking tool?

To answer these questions, we invited two pairs of developers to use our method for exploring and comprehending a software system they had not seen before. We observed their exploration and note taking strategies (RQ₁), gathered their comments and opinions through questionnaires (RQ₂), and scrutinized the insights they accumulated on virtual whiteboards (RQ₃) by consulting with the original developers of the subject system, in order to verify the accuracy and relevance of the accumulated information.

Our results show that participants engaged vividly in a collaborative software exploration in VR, appreciating especially the architecture-level overview on the system's structure, yielding correct insights.

F.2 Background and Related Work

Our method integrates collaborative software exploration and note-taking in a VR software visualization, positioning it within two main research domains: software visualization and (re-)documentation.

F.2.1 Software Visualization

Software visualization entails the use of visual metaphors to depict abstract and intangible concepts present in source code, with the intention of aiding users in obtaining both a general understanding and detailed insights into a software system's structural, behavioral, or evolutionary facets [4, 5, 6].

The visual metaphors employed range from abstract representations, such as graphs [12, 13, 14] or tree maps [15, 16], to real-world inspired structures like solar systems [17, 18, 19], islands [20], and cities [21, 22, 23, 24, 19]. Additionally, visual metaphors in software visualization can be categorized based on their dimensionality into 2D [25, 26, 27] and 3D variants, with the latter further distinguishable by the medium utilized, spanning standard 2D screens [28, 29, 30], VR [31, 32, 33], and AR [34, 35, 36].

There is a noticeable gap in research focusing on collaborative software exploration and understanding. Anslow et al. introduced a method using an interactive touch-screen table for collaborative exploration, providing different structural views of a subject system [37]. This method is inherently designed for co-located collaboration. For distributed teams, research has explored VR as a medium for collaborative software exploration. Koschke et al. provide a collaborative VR method with diverse views, i.a., for clone detection or identifying architectural drift [9, 8]. Krause-Glau et al. propose a method for behavioral aspects investigation in a collaborative setting across various devices, including VR headsets [38]. While these contributions are valuable for collaborative VR software exploration, they lack the means for engineers to take notes on their insights, potentially leading to loss of valuable information during prolonged sessions – which constitute a relevant use case. We address this gap by proposing a method that integrates collaborative note taking into the exploration process (Section F.3).

Collaborative environments for software exploration exist mainly for VR. They inadequately support concurrent note-taking, which is essential for preserving insights.

F.2.2 Software Documentation and Note Making

Various techniques and tools exist to aid engineers in generating notes and documentation on a software system's source code.

These range from automated documentation generators like RGB [39], Scribble [40], PAS [41], Re-Doc [42], and others [43, 44], to more informal methods like freehand sketching on paper or whiteboards [45, 46]. The latter is particularly prevalent in collaborative scenarios, providing a flexible medium for capturing ideas and insights. In recent work, Hoff et al. proposed a VR-based sketching method that enables engineers to pin elements from a software visualization and, based on that, sketch diagrams directly on whiteboards [11]. Their method performs conformance checks between the sketched diagrams and the source code, ensuring alignment. However, so far, the method proposed by Hoff et al. was used in only one study and that was not in a collaborative exploration setting. Further, to thoroughly capture complex and extensive notes during the software exploration process, we advocate for additional mechanisms beyond freehand sketching, which are not present in the previous work by Hoff et al.

While freehand sketching is a valuable tool for note-making in software engineering, current VR-based methods need more versatility to adequately capture long and complex notes.

F.3 Collaborative Software Exploration and Note Taking in VR

To support teams of engineers in collaboratively exploring software systems, we propose a VR method that enables multiple engineers to enter a shared synchronized virtual environment (each with their individual head mounted VR device) to collaboratively explore a subject system while capturing thoughts and insights on shared VR whiteboards via freehand sketching, audio recordings, and in-visualization screenshots. Our method can be used in a local network as well as over the internet, allowing distributed teams of engineers to meet up in a virtual space to collaboratively explore a system's architecture and design, discuss ideas, and make notes.

F.3.1 Collaborative Interactive VR Visualization

Our method supports teams of collaborating engineers in freely exploring software systems in a top-down fashion [47], following Schneiderman's

mantra “*overview first, zoom and filter, details on demand*” [48]. It provides engineers with an overview of the folder-level and class-level structure of a subject system, relationships between elements on both these levels, synchronized mechanisms for navigation and zoom, and details on demand on source code as text. Our concepts are designed for subject systems written in object-oriented programming languages. In the following, we elaborate on these using Java as an example.

Architecture Level - Folder Spheres

To provide engineers with an overview of the subject system, our method visualizes its folder structure as hierarchy of nested semi-transparent spheres (see Figure E.3 ①), similar to the software landscapes by Balzer et al. [49, 50]. Each sphere represents one folder with constituent elements (i.e., class-level elements and sub-folders) contained inside it, arranged in a circular layout parallel to the floor of the virtual environment ②. On system root level, all visual elements are contained in a root folder sphere containing the entire subject system ③.

Colors for Orientation To help engineers with distinguishing elements from different parts of the subject system, our method determines colors for folder spheres based on the position of the represented folder in the system’s hierarchy.

Figure E.2 depicts an example that illustrates the color distribution concept. Leaf-level folders are assigned evenly distributed hue values while their nesting level determines the color’s saturation. Hue values for all remaining folders result from the average of their sub-folder colors’ hue.

Interaction via Core Depending on the amount and extents of their contained elements, folder spheres vary in size. With their semi-transparent look, they serve to guide engineers visually, creating an environment engineers can move through and stand in without unintentionally triggering interactions. We consistently manage interaction with folder spheres through a central core, see Figure E.3 ④. These cores are uniform in size (independent of the folder sphere’s radius), ensuring a standard interaction across the system. Engineers have the option to grab cores for discussions with peers or tap on them to access a user interface offering additional information and options ⑤, such as selecting specific entity relationships to display. When an engineer releases the core, it automatically repositions itself back to the folder sphere’s center.

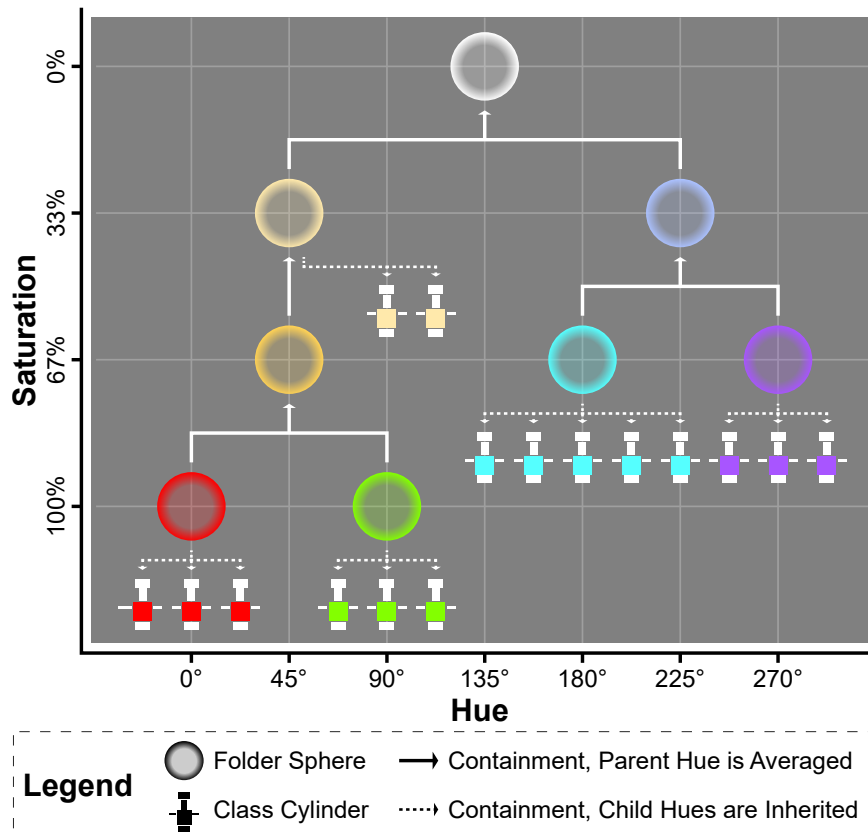


Figure F.2: Example folder sphere hierarchy depicting the color scheme employed by our method.

Opening/Closing Folder Spheres To support engineers with focusing their investigation on relevant parts of a subject system, the visualization allows adjusting the amount of details in a view by opening and closing folder spheres. This is done via the folder sphere's core in a user interface or via hand gestures. When opening a system for the first time, its root folder is opened by default with all first-level folders visible but closed (cf. Figure F.3 (j) and (l)). Thereby, our method limits the amount of information immediately presented to engineers - a feature that is useful for large subject systems. Engineers open the visible folder spheres to look into their directly contained elements. Closing a sphere hides all constituent elements until opened again. The right-hand side of Figure F.3 (b) shows a folder sphere after opening with its sub-folders spheres closed (compare with (j) and (l)). Opening and closing folder spheres is synchronized between all collaborating engineers.

While we considered a feature for opening all folders simultaneously, we decided against it to prevent users from becoming overwhelmed and losing their overview of the system. Our intention is to promote a top-down ap-

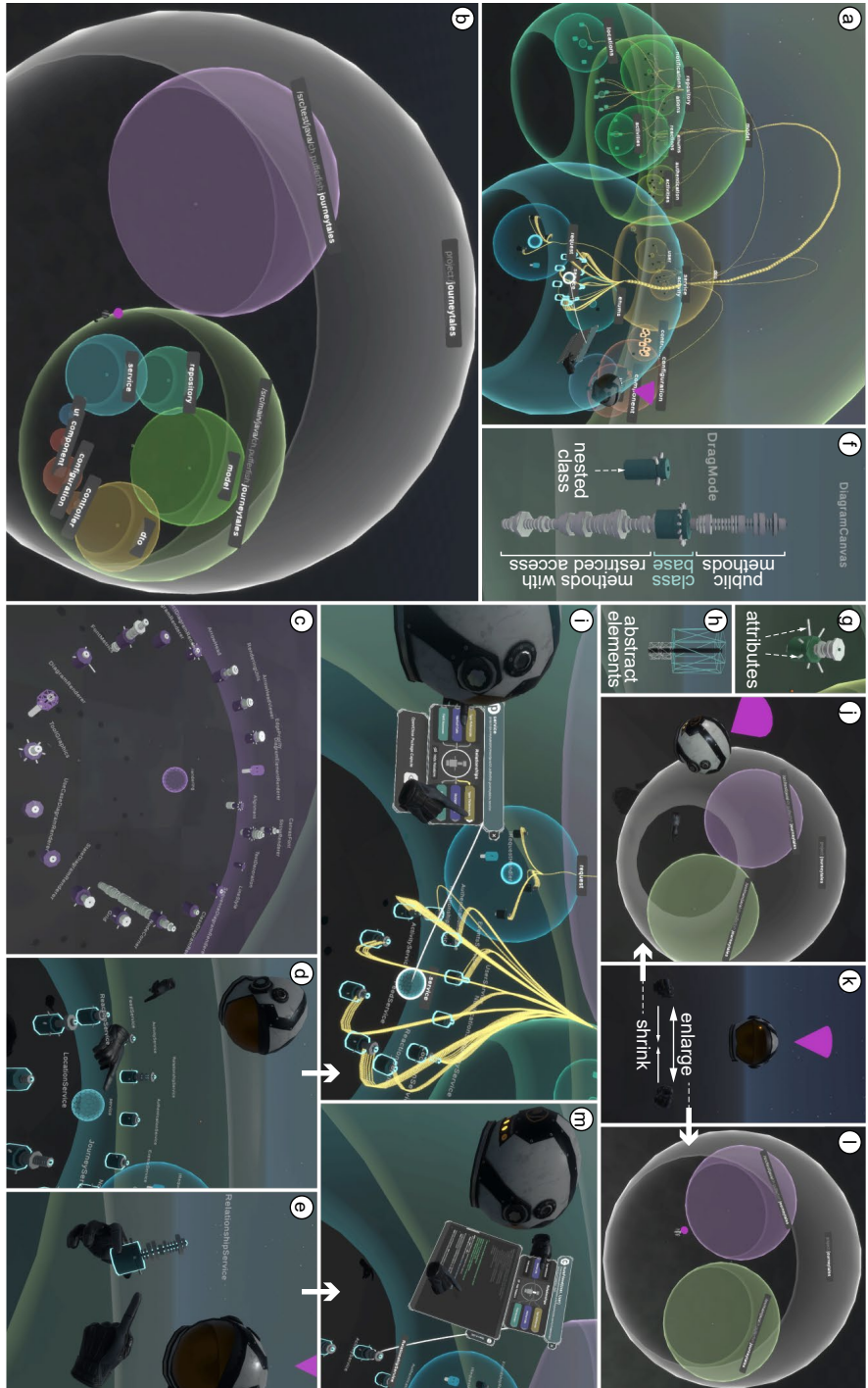


Figure F.3: Screenshots from an implementation of our VR software visualization method. Folders/packages are represented as colored nested spheres (a-c). Classes, interfaces, etc. and their members are represented as stacks of cylinders (class cylinders), encoding meta information and structural metrics via location, shading, and form (f-h). Engineers can interact with elements (d, e) to blend in visual relationships graphs (i, a), read code (m), or scale the visualization up and down (j-l).

proach to exploration. Additionally, while our current implementation favoring a simpler user interaction does not allow engineers to completely hide folders and thereby only display selected non-hidden content, a feature like this could become valuable for navigating very large systems.

Class Level - Class Cylinders

Our method represents class-level elements in a way that provides engineers with a rapid overview of their inner member-level structure by encoding information in several visual properties: base cylinder, method cylinders, attribute spikes, and nested class cylinders discussed below.

Base Cylinder Every class cylinder consists of one base cylinder that is colored according to the color of its containing folder (see Figure F.2 on the coloring concept and Figure F.3 (f) as implemented). Further, our method visually distinguishes between concrete and abstract class-level elements via the surface shading of the base cylinder. For instance, in Java, the base cylinder of a concrete class is displayed with an opaque surface (f) whereas abstract classes and interfaces receive a see-through wireframe surface to convey the look of a less tangible and less concrete structure (h).

Method Cylinders Methods/functions of a class-level element are represented in form of cylinders where structural metrics determine their shape, i.e., the number of expressions in a method determines the cylinder's height while its cognitive complexity [51] (driven primarily by the depth of control flow splits) determines the cylinder's radius.

To rapidly grant engineers an overview of the accessibility of methods, cylinders for methods with unrestricted access (e.g., public modifier in Java) are stacked on top of the base cylinder while cylinders for encapsulated methods (e.g., private, protected, or package visibility in Java) are stacked underneath the base cylinder. Figure F.3 (f) depicts an example of a class with numerous public and private methods with varying size and complexity.

Attribute Spikes Attributes/fields of a class-level element are represented as evenly distributed spikes originating from the colored base of a class cylinder, with unrestricted attributes being notably longer than encapsulated attributes, see Figure F.3 (g).

Nested Class Cylinders Nested class-level elements are represented using the above described mechanisms for regular classes, with two deviations: (1) their base cylinder is notably smaller, i.e., half the size of a root-level class; and (2) to represent the structural connection with their containing class-level element, nested class cylinders are arranged in an evenly spaced circle around the nesting class cylinder. As an example, the left-hand side of Fig. E3 ① depicts a nested class “DragMode” in a regular class “DiagramCanvas”.

Interaction Engineers in VR can grab class cylinders and hold them in their hands ②, e.g., to show them to a collaborator in a conversation. When released, the class cylinder smoothly returns to its original position in the visualization. Further, engineers can tip on the base cylinder, method cylinder, or attribute spikes of a class cylinder with their virtual fingers to open detailed views with additional information (see Sections E3.1 and E3.1).

Source Code Views

On demand, our method provides engineers with a textual view of a visualized element via a synchronized scrollable user interface, see Fig. E3 ①. This interface shows the source code of an element in the context of the file it is stored in. For instance, the source code view for a method displays all code of the containing file where it (1) initially scrolls to the location of the selected method and (2) emphasizes it by graying out all leading and trailing code. In Fig. E3 ① the upper part of the presented code is grayed out to highlight the currently displayed code section.

Relationships

Our method provides engineers with an overview of the relationships between elements via an interactive graph based on statically analyzed references in the subject system’s code. This relationship graph represents references as animated directed lines between respective visual elements, originating from the member that contains the reference and ending in the referenced element, see Figure E3 ③ and ④. Our method distinguishes between incoming and outgoing references to or from a selected element as well as between type references, method calls, and field accesses respectively. Engineers can individually show and hide relationship lines for each of these categories for a selected folder, class-level element, method, or attribute via a synchronized user interface attached to the respective visual element, see Figure E3 ④. To reduce clutter and visual complexity, relationship lines are bundled together as they

cross folder sphere boundaries, similar to the technique proposed in [49, 50], see Figure E3 @. Displaying relationships for a software element containing multiple members (i.e., folders and class-level elements) summarizes all contained incoming/outgoing relationships.

Navigation and Zoom

Engineers can freely navigate the visualization and change their point of view, which is synchronized in real time with all collaborators. They can change their position by (1) teleporting through the virtual space and (2) rotating around their virtual axis (a VR mechanism commonly referred to as “snap turning” or “snap rotation”). To facilitate engineers’ potential experience with other VR tools or games, these resemble standard VR navigation mechanisms present in a majority of current VR applications. Further, they can use hand gestures to (a) move/offset the entire visualization (i.e., the hierarchy of folder spheres with all constituent elements) and (b) zoom in and out to change scale of the root folder sphere relative to the engineer’s hand position, see Figure E3 j) to ①. To ensure a consistent point of view on the system, these mechanisms are synchronized between all collaborators. They offer fine-grain control over the point of view on a subject system while mitigating entry barriers by being operable when seated and in a small space.

F.3.2 Collaborative Note-Taking on VR Multi-Media Whiteboards

Our method encompasses mechanisms for collaborative note taking that enable engineers to persist insights and synchronize their understanding of a subject system while exploring it in VR: building on work from Hoff et al. [11], engineers work with virtual whiteboards to (a) pin elements from a VR software visualization (such as the folder spheres or class cylinders presented in Section F.3.1) and (b) pick up a virtual pen and sketch freely. Figure F4 n), v) and u) illustrate these interactions. Software elements pinned to a whiteboard are represented as pins displaying references between the pinned software elements via curved relation lines. Further, the method presented in [11] automatically interprets freehand drawn shapes as outlines around pins (called modules) and relation-arrows between these. Based on that, it provides engineers with conformance checks between their sketches and the represented software structures by coloring relation lines between pins. Our method extends Hoff et al.’s work [11] with support for taking multi-media notes during an ongoing collaborative exploration via: 1) synchronized whiteboard inter-

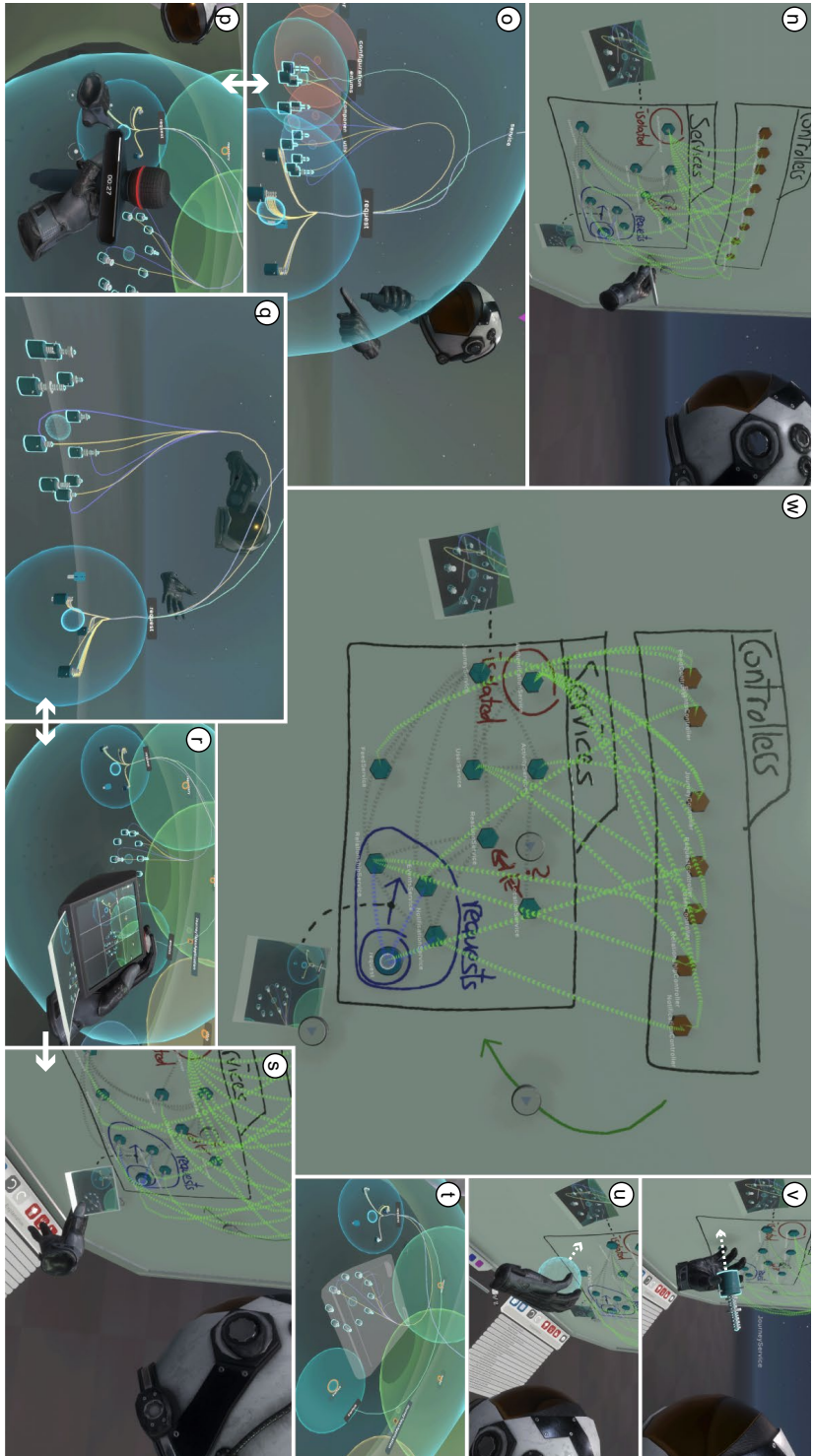


Figure F.4: Screenshots from an implementation of our note making concepts in VR. While exploring a subject system, engineers can pick up a pen and freely sketch on virtual whiteboards (n). They can pin software elements from the visualization (y, u) and draw tool supported diagrams (w). To capture longer thoughts, they can create and pin sound recordings (o, p). Lastly, they can create and pin screenshots (q, r, s).

action (F.3.2), 2) in-visualization screenshots (“VR-shots”) (F.3.2), and 3) audio recordings (F.3.2).

Synchronized Whiteboard Interaction

We extend the VR whiteboards presented in [11] by collaborative capabilities. For one, engineers can grab and freely position whiteboards, which is synchronized to their collaborators in real time. For another, we extend the virtual whiteboards with support for concurrent edits, see Figure F.4 (n). Free-hand drawn notes and pins on a whiteboard (including those presented in the remainder of this section) are synchronized in real time so that the content of all whiteboards is consistent across all collaborators.

In-Visualization Screenshots (“VR-Shots”)

Our method enables engineers to quickly capture and pin screenshots to a whiteboard in VR, serving as a visual reference alongside other notes.

Engineers can take a screenshot by grabbing a virtual camera in VR and pressing its trigger, see Figure F.4 (q) and (r). They can then pin the virtual print-out of the screenshot to a shared whiteboard (s). For instance, this can be used to capture a view on relationship graph lines between two class cylinders in the visualization.

By tapping on pinned photos on a whiteboard, engineers revert the visualization to its exact state at the screenshot’s capture, including position, scale, folder spheres’ open/close statuses, and relationship graph state, while simultaneously being teleported to the screenshot’s location. A semi-transparent indicator marks the camera’s position when the photo was taken, aiding in understanding the photo’s perspective, see Figure F.4 (t). This implements a form of save point for engineers to return to at later points in time.

Audio Recordings

Engineers can share longer and more detailed thoughts using audio recordings while exploring a subject system in VR, helping to free up their mental space. They do this by picking up a virtual microphone, speaking their thoughts aloud, and then pinning the recorded audio to a whiteboard as an audio pin, see Figure F.4 (o) and (p). By tapping on audio pins, engineers can play the recording, with playback shared in real-time among all collaborators.

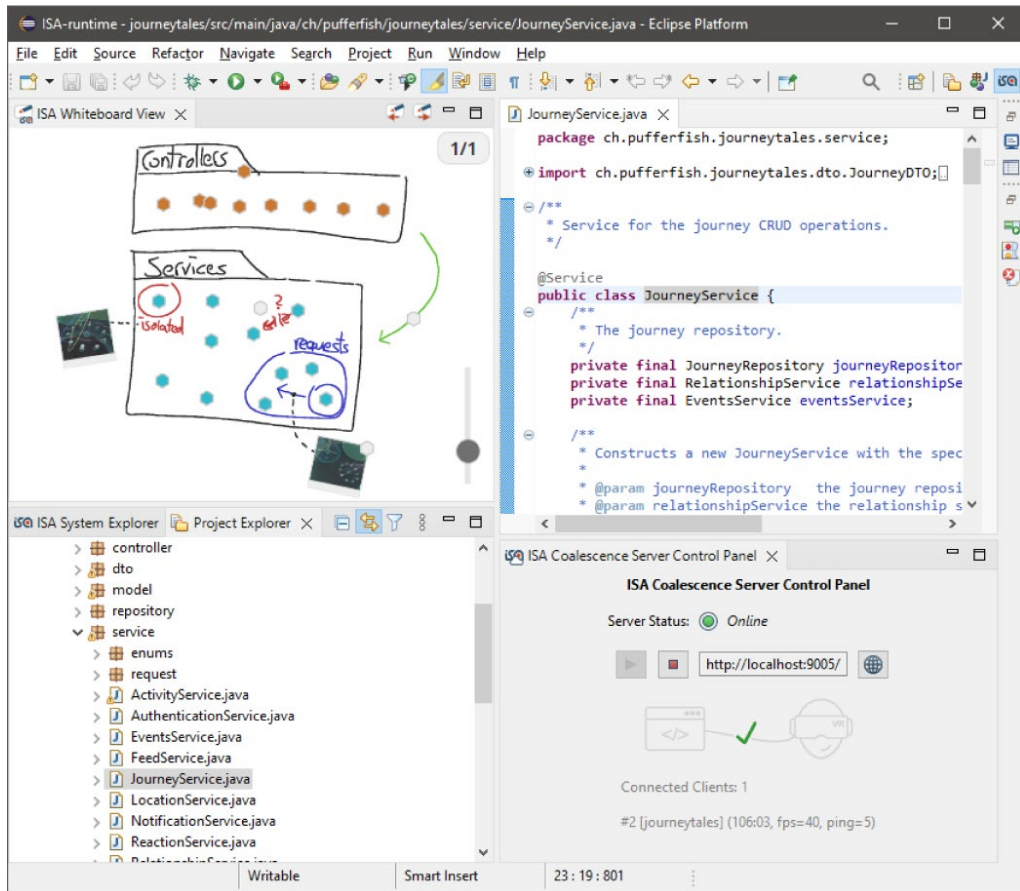


Figure F.5: VR whiteboards are synchronized with an IDE where they can be inspected and interacted with (top-left area) by zooming, panning, playing audio recordings, and opening pinned elements.

Synchronization with IDE

With the above mechanisms, our method aims to support engineers in exploring and comprehending a subject system while persisting thoughts and insights on virtual whiteboards. Further, it synchronizes these whiteboards with an IDE to make engineers' accumulated thoughts and insights accessible outside the VR environment. Figure F.5 depicts an integration into the Eclipse IDE; the whiteboard from Figure F.4 is displayed in the top-left area where engineers can zoom and pan, enlarge screenshots for detail inspection, play audio recordings, and directly open the code of pinned software elements in the IDE code editor. This feature bridges the gap between VR exploration and further use of created notes for subsequent work outside of VR, e.g., to implement planned changes.

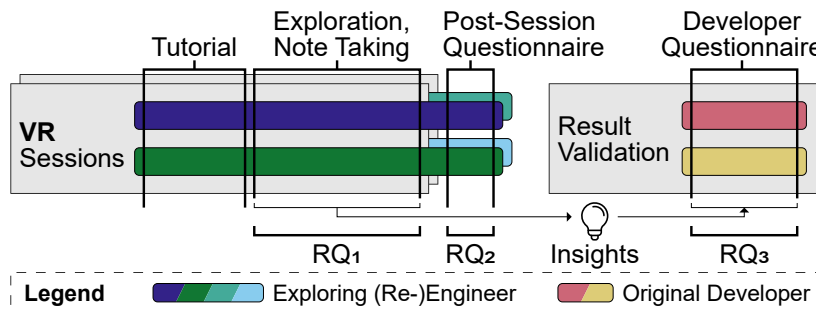


Figure F.6: Case study procedure: Two pairs of developers collaboratively explore a subject system and fill in a questionnaire (left-hand side). We relay the gathered insights to the original developers and assess their correctness and relevance (right-hand side).

F.4 Case Study with Practitioners

We evaluated our method in an observational case study. Two pairs of software engineering practitioners participated, using an implementation of our VR method to explore a subject system in an open and uninterrupted setting (Figure F.6, left-hand side). We collected their feedback through a post-session questionnaire. Subsequently, we scrutinized the insights they accumulated on virtual whiteboards and presented it to the original developers for validation of accuracy and relevance (Figure F.6, right-hand side).

F.4.1 Tool Implementation

We implemented the concepts presented in Section E.3 in our tool Immersive Software Archaeology¹ (ISA). ISA employs a client-server architecture, with the server ensuring real-time synchronization and consistency of whiteboards and the visualization across clients. The client is developed in C# with the Unity game engine, relying on the SteamVR platform² to make it compatible with all major VR hardware. The server side is implemented in Java as part of ISA's ecosystem of Eclipse plug-ins where it integrates with an automated software analysis. All code is open-source in ISA's repository¹.

F.4.2 Case Study Procedure

Our study is divided in two phases (cf. Figure F.6), i.e., VR sessions with software engineering practitioners (depicted on the left) and subsequent result

¹<https://gitlab.com/immersive-software-archaeology>

²<https://store.steampowered.com/app/250820/SteamVR/>

validation by the original developers of the subject system (shown on the right).

VR Sessions

In a first step, we addressed RQ₁ and RQ₂ through VR sessions with pairs of software engineering practitioners. Our tool supports both internet-based distributed setups as well as local network connections. However, for our study, we chose to have participants collaborate in the same room to facilitate a smoother introduction and especially to help with the VR hardware. The VR sessions were structured in three main stages.

1. *Tutorial (max. 30 minutes)* Each VR session began with a brief tutorial, explaining the tool's visual metaphor and VR controls, navigating the system and interacting with elements. Participants and the experiment instructor (first author of this paper) went through this process together in VR, following a predefined script. The full tutorial script can be found in our online appendix³.

2. *Exploration and Note Taking (min. 45 minutes)* Following the tutorial, the instructor briefly introduced the subject system (two sentences) and read out the open experiment task: "Please explore the system and make notes of your findings." Subsequently, the instructor did not intervene unless there were technical issues or if participants requested help with controls or interaction. No content-related assistance was given. The entire script is available in our appendix³. The end result of each session was a set of VR whiteboards with notes and audio recordings (see online appendix³). Additionally, for both sessions, we video-recorded participants' VR point of view as well as audio of what they were saying.

3. *Post-Session Questionnaire* After the VR sessions, each participant filled out an anonymous questionnaire on their own, sharing their thoughts on the support provided by our method in collaboratively exploring a software system and taking notes on gained insights. The full questionnaire is available in our online appendix³.

³<https://doi.org/10.6084/m9.figshare.24499726>

Result Validation

In the second phase, we addressed RQ₃ by forwarding participants' insights from the VR sessions to the subject system's original developers using the post-session questionnaire³. For that purpose, we manually analyzed all handwritten notes and audio recordings on the virtual whiteboards created by participants of the previously conducted VR sessions, consulting video recordings of the VR sessions to ensure accurate context interpretation. We combined video recordings of participants' VR POV side-by-side, resulting in one video for each session. We then manually analyzed these videos and extracted all notes written on virtual whiteboards using the context provided by the video, e.g., a conversation between participants. We included all insight explicitly noted and display them in Table F.2. Long audio pins commenting on multiple aspects resulted in multiple separate insights. The subject system's original developers subsequently evaluated each insight for its accuracy and relevance ("How relevant is this insight when planning potential changes to the system?").

F.4.3 Subject System

Our study focuses on a Java Spring Boot web server backend API, a component of a travel journey management system with approximately 10,000 lines of code. The system enables users to monitor their travel activities, whether by plane, train, or visits to specific locations. Key features include persistent access to travel data, a user account system, search functionality, and friend management. The system's source code is publicly accessible⁴.

We chose this system for our study because we could contact the original developers, a crucial aspect for validating our results. Moreover, Spring is among the most popular frameworks for building enterprise applications and thus the system is a good representative for a large class of relevant Java systems. However, note that our visualization is not tailored for Spring applications.

F.4.4 Participants

Our only inclusion criterion for participants of the study was having professional experience with Java development. We did not explicitly seek participants experienced with Spring Boot or VR and asked about contact with these

⁴gitlab.com/usi-si-oss/teaching/projects-showcase/sa4/team-4-pufferfish/backend

technologies in our questionnaire. Four engineers from one company participated in our study, all with limited VR experience prior to the study. One participant reported having never used VR at all. Concerning their familiarity with software systems akin to the subject system, three of the participants confirmed their regular use of the Spring platform, while one reported having previous, yet not extensive, experience with it.

F.4.5 RQ₁: How do engineers explore and take notes of a software system in a collaborative VR software exploration and note taking tool?

Figure E7 depicts a timeline highlighting the activities and communication patterns of each participant in the study. We analyzed video recordings for both sessions, categorizing participant activities into seven distinct types: (1) exploring the system through inspecting elements of the visualization (most notably folder spheres, class cylinders, relationship graphs), (2) exploring relationships between elements on a virtual whiteboard (mostly re-arranging pins and tracing relations via curved relation lines between pins), (3) examining and navigating source code on UI canvases, (4) handwriting notes on a virtual whiteboard, (5) attaching software elements to a whiteboard, e.g., in proximity to a handwritten note, (6) creating and pinning audio recordings to a virtual whiteboard, and (7) capturing and pinning VR screenshots to a virtual whiteboard.

We summarize multiple interactions with the visualization under point (1) since retrospectively determining the participants' focus during their visual exploration of a complex view comprised of nested, semi-transparent folder spheres, numerous class cylinders, and connecting relation lines is challenging to accurately pinpoint.

Varied Architectural Exploration Throughout both sessions, participants oscillated between exploring the overarching architecture and inspecting source code details, mostly engaging in collaborative work with occasional individual exploration. In Session 1, participants adopted a top-down, breadth-first approach for exploring the system's architecture, employing a mix of examining the visual elements of the visualization (i.e., folder spheres and class cylinders) and pinning package folders to virtual whiteboards for continued examination. They inspected source code when class-level elements aroused their interest, though their primary focus remained on comprehending the

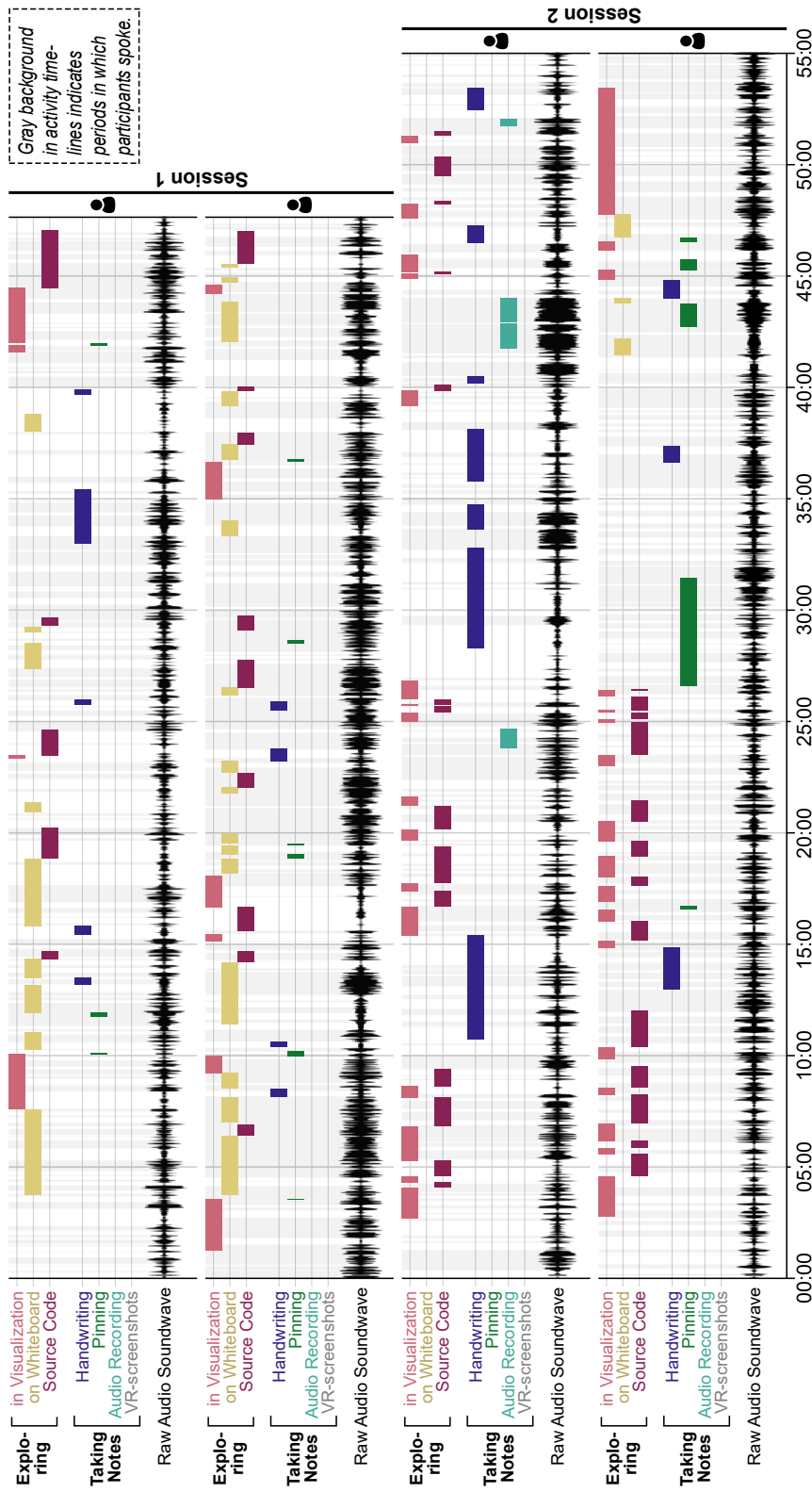


Figure F.7: Timeline depicting participant activity in the two VR sessions. It shows for each participant the intervals in which they were carrying out one of the activities listed on the left-hand side. Note that activity labels are magnified for one of the four participants due to space constraints. Further, participants' speech recordings are visualized, with intervals of actively speaking to one another highlighted.

system's architectural structure. This pattern is visible in the activity timeline for Session 1, as shown in Figure F.7.

Participants in Session 2 adopted a more dynamic approach, swiftly navigating through folder spheres and intermittently analyzing source code, driven by the spontaneous discovery of relevant folders and classes. In Session 2, virtual whiteboards were scarcely used for exploration purposes (cf. Figure F.7).

Handwriting Varied Across Sessions In both sessions, participants made extensive use of handwriting for documenting insights, but varied their strategies. Session 1 participants mainly wrote isolated single words to provide context to clusters of pinned class-level elements on the whiteboard, while participants in Session 2 produced more elaborate handwritten notes and supplemented them with audio recordings. In both cases, whiteboard pins for software elements and audio recordings were strategically located in relation to handwritten notes.

VR Screenshots Not Utilized None of the participants incorporated VR screenshots in their virtual whiteboards. As far as we can tell, participants did not experience scenarios in which they perceived capturing an image of a specific part of the visualization or using it as a save point for future reference as beneficial.

Ample Communication Figure F.7 shows for each participant both their raw audio soundwave on our recordings of the VR session as well as periods of time when they were speaking (as gray background behind the activity timelines). It shows that in both sessions, participants were overwhelmingly active in communication with only occasional short silent phases of individual source code inspection. Furthermore, we observed discussion phases, where exploration and note taking were temporarily halted to deliberate on ideas, most notably in Session 1 ca. from minute 30 to 33.

F.4.6 RQ₂: What strengths and weaknesses do engineers perceive in using a collaborative VR software exploration and note taking tool?

In the following, we provide a summary of the feedback collected from participants of both VR sessions through a post-session questionnaire (cf. Figure F.6). The questionnaire yielded participants' quantitative verdict on the support they received for exploring a subject system and taking notes on the

Table F.1: Stacked bar chart with participant responses from the post VR session questionnaire. Bars are colored and sorted by participant (see mapping of participants to colors in Figure F.6). Bars extending more to the right indicate stronger agreement or perceived usefulness. The answer of Participant 3 to Q₇ was discarded due to a misunderstanding of the question.

No.	Statement / Question	Agreement (Likert-Scale)	Avg.
Q ₁	The VR tool helped me with exploring the subject system's architecture.		4.25
Q ₂	The VR tool helped me with exploring the system's class-level elements.		3.25
Q ₃	The VR tool helped me with taking notes and documenting the subject system.		2.75
Q ₄	How valuable do you assess the audio recording feature?		3.25
Q ₅	How valuable do you assess the camera feature?		3.75
Q ₆	How valuable do you assess the whiteboard as a whole?		4.5
Q ₇	How valuable do you assess the collaborative aspect of the VR tool?		4.33
Q ₈	The VR tool provided a benefit to how I would usually have explored and taken notes on the software system.		3.75

results in VR (Table F.1) as well as qualitative data in terms of free text comments. A full version with all verbatim comments is available in our online appendix³.

Exploration Table F.1 shows that participants assessed the support they received in exploring significantly higher than the support they received in taking notes (Q₁&Q₂ vs. Q₃).

Further, Table F.1 shows that participants valued the exploration capabilities in VR especially for architecture-level aspects (Q₁ vs. Q₂). In their comments, participants reported on a perceived ease in obtaining an overview of the subject system, identifying relevant software elements, and understanding their interconnections. Apart from that, they wished for textual search for software elements and reported on a general unfamiliarity with VR resulting in perceived slow interaction with the tool: *“I’m not used to VR, so I was slow to perform the activities.”*

Note Taking Participants’ overall merit of using VR to take notes was mixed (cf. Table F.1; Q₃). On a positive side, they highlighted the benefits of having

unlimited space on the virtual whiteboards, their integration with the rest of the visualization, and the resulting high-level views on a subject system's architecture. One participant particularly emphasized the voice recordings and the ease of linking audio to visual elements on the whiteboards. Conversely, criticisms centered around the cumbersome nature of handwriting in VR with multiple related comments to an unfamiliarity with VR and the need for more practice.

For audio recording, participants of Session 1 who did not use the feature gave worse feedback (both giving a 2 on the Likert-scale, Q₄) than participants of Session 2 (who extensively used the audio recordings, and who gave a 5 and 4 grade respectively). Session 1 participants both justified their low scores with the absence of an automated speech-to-text transcription feature, e.g., *"I wouldn't use it much because i prefer having written notes. Maybe it would be useful if I could create a transcript from the recording"*.

In accordance with our observations during the VR sessions, participants perceived the virtual camera as underutilized, yet appreciated (cf. Table F.1; Q₅): *"I forgot to use it, but it certainly helps to explore/move faster when switching between whiteboards and the codebase"*.

Overall, despite comments on requiring more practice to feel comfortable with handwriting in VR, participants assessed the virtual whiteboards as very useful (Q₆).

Collaboration Feedback on VR collaboration was mostly positive (cf. Table F.1; Q₇) and in line with our observations and answer to RQ₁. As suggestions for future work, participants emphasized a wish for locking elements in space that are shared between collaborators (especially whiteboards) so that they cannot be moved and repositioned until unlocked again to avoid accidental interactions.

F.4.7 RQ₃: What type of insights do engineers extract from a system when using a collaborative VR software exploration and note taking tool?

Table F.2 lists all insights on the subject system captured by participants in form of handwritten notes and audio recordings on virtual whiteboards. Session 1 yielded Insights I_{1.1} to I_{1.4}, while Session 2 provided Insights I_{2.1} to I_{2.10}. We investigated these insights to discern patterns and verified their accuracy and relevance by seeking feedback from the original developers of the subject system via an online questionnaire (available in our online appendix³).

Table F.2: Stacked bar chart with feedback from the subject system’s original developers on correctness and relevance of the insights collected during both VR sessions. Each bar represents an answer from one of the two original developers. Wider bars with higher values indicate more correctness or perceived relevance.

No.	Participants’ insights into the subject system, noted on virtual whiteboards	Feedback from Developers			
		Correctness		Relevance	
I1.1	Among the system's modules are 'controller' and 'service'.	5	3	5	2
I1.2	The system uses server-side events for social media aspects, e.g., updating the trip feed, forwarding reactions to trips, etc.	5	5	3	3
I1.3	All controllers interact with the authentication service.	4	4	4	5
I1.4	The location service is isolated from the other services (they don't interact).	4	5	3	2
I2.1	The backend system does not have any sub-systems, it is a single big system with multiple responsibilities.	4	4	5	4
I2.2	The system has both unit tests and integration tests.	4	5	5	5
I2.3	There are todos in the test classes that should be looked into.	5	5	2	2
I2.4	Journeys consist of lists of activities with visits to locations.	5	5	5	5
I2.5	Users create and own journeys.	5	5	5	3
I2.6	User profiles can be private or public.	5	5	5	4
I2.7	Users can add one another as friends.	5	5	5	4
I2.8	Friends can have trips together.	1	1	4	1
I2.9	Users can react to trips of friends.	5	5	1	3
I2.10	There is a feed of activities (trips) that is shared with friends.	5	5	4	2

Extracted insights Session 1 participants focused on the system's overarching structure, adopting a strict top-down approach (cf. answer to RQ₁ above). This pattern is evident in their notes, which exclusively covered system-level and architectural aspects without addressing behavioral details.

Conversely, Session 2 participants adopted a use-case centered exploration, focusing on more specific aspects of the system's behavior and inner workings rather than system-wide aspects. They started their exploration with the system's test package, e.g., investigating example usages of different parts via unit tests. Thus, notes of Session 2 participants touched upon testing (Insights I_{2.2} and I_{2.3}) while the remaining notes capture the system's behavior from a user's point of view.

Correctness As per verdict of the original developers, the insights participants noted during their VR sessions were largely correct with an average of 4.43 on a scale from 1 (incorrect) to 5 (correct). Only Insight I_{2.8} was clearly incorrect.

Relevance As per the verdict of the original developers of the subject system, the relevance of participants' insights varied with an average of 3.61 on a scale from 1 (irrelevant) to 5 (relevant).

F.4.8 Discussion of Results and Lessons Learned

In the following, we summarize our results for RQ₁-RQ₃ and highlight lessons learned from our case study for builders of related VR tools. Overall, our study demonstrates that even practitioners with minimal prior experience in VR software visualization can utilize methods like ours for collaborative exploration and comprehension.

Key Suitability at Architecture-Level Through observation and direct participant feedback, we identified that a VR exploration and multimedia note-taking environment primarily enhances work at the architectural level as opposed to finer statement-level details.

Freehand Sketching in VR Requires Practice Based on participant actions and feedback, relying solely on handwriting and sketching in VR for capturing insights can be tedious and time-consuming, a situation that might extend beyond VR environments. However, participants also noted improvements in their VR handwriting skills even within the short duration of the case study

sessions, an observation made frequently in conversations between them, e.g., “I wrote ‘friends’. Ah, the handwriting is getting better!”.

Audio Recordings Require Transcription Audio recordings were more favorably received than handwriting, although there was a clear desire for enhanced tool support in this area, mostly in terms of automatic conversion of audio recordings to text.

VR Screenshots Require Further Investigation While screenshots were deemed valuable, they were not utilized in the VR sessions. Additional research is necessary to explore this phenomenon, e.g., to investigate a correlation with the duration of VR sessions or in usage spanning over multiple VR sessions.

Communication was Ample There were clear signs of successful collaboration among participants. This was evident not only from our observations during the VR sessions but also from the constant communication occurring throughout them, as depicted in the audio waves in Figure F.7. Participants utilized having different perspectives on the same subject system and synchronizing their insights, e.g., one participant reading code, the other finding a mentioned method through the relationship graph and exploring from there.

Accurate Results with Varying Relevance In addressing RQ₃, we determined that participants in our case study produced results that were accurate. The relevance of these results varied, which is understandable considering that we instructed participants to record all noteworthy findings without specifically evaluating their relevance in the constrained timeframe of the VR sessions. Although our findings on the correctness of participants’ insights are encouraging, further studies are needed to investigate their relevance, e.g., to find correlations between insights, their correctness, and their relevance (which we were not able to find in this work).

VR Requires Practice A recurring theme in feedback and observations was the novelty and necessary practice associated with both the method and VR as a technology in general. This applies to both exploring software in VR and taking notes on the findings as highlighted in participant feedback during the post-session questionnaires.

F.4.9 Reflections and Threats to Validity

In the following, we discuss potential risks to our study's findings and our strategies to reduce their impact.

Participant Selection and Number The preliminary case study presented in this work was conducted with only four engineers organized in two pairs that worked together. These were selected by convenience, i.e., we contacted a company and asked for volunteering engineers willing to participate in the study. This indicates a necessity for more extensive investigations to uncover additional patterns in the behavior of a broader spectrum of engineers. Nevertheless, our study granted first insights into how practitioners use a VR software exploration and note taking tool, how they assess different tool features, what kind of information they extract from it, and how accurate and relevant these are.

Subject System The subject system used in our study comprises 10,000 lines of code. Results from our study must thus be interpreted in a context of exploring similarly sized systems.

Further studies must be conducted to evaluate the scalability of VR exploration and note taking tools for systems of significantly larger scale. We opted for the subject system used in this study because we had access to its original developers, a unique opportunity for assessing the results of participants' exploration sessions.

Potential Response Biases Response biases in studies can occur due to question wording or social dynamics among participants or between participants and interviewers [52]. In our study, this threat potentially applies to feedback from VR participants as well as to the result assessments of the subject system's original developers. To counteract response biases, we kept our experiment's purpose confidential until after its completion and minimized our interaction with both participants and the subject system's original developers. With the latter, we had no contact prior to the study other than for forwarding the insights collected by VR participants (initial contact was established by a third party). Moreover, we gathered feedback through anonymous, individually answered questionnaires. VR participants completed these directly following their session.

F.5 Conclusion and Future Work

Our method allows distributed software engineering teams to immerse into an interactive VR space to visually analyze and comprehend a subject software system. In this virtual space, engineers can take multi-media notes on their observations and insights using a variety of tools including freehand sketching, diagramming, audio recordings, and screenshots.

We conducted a preliminary case study to investigate how pairs of software engineering practitioners use our method to explore a subject system. The participants, new to the system they were exploring, provided valuable insights and feedback on different VR features. Further, we assessed the accuracy and relevance of their findings by consulting the system's original developers. All in all, despite requiring more practice to fully utilize our method and VR technology, participants found our approach beneficial for architectural exploration, exhibited vivid communication during the sessions, and produced correct notes.

Future work will entail larger and more refined studies with additional software engineering practitioners, and comparisons with traditional (non-VR) collaborative software exploration methods such as other kinds of software visualization and IDEs. Further, we plan to study engineers' interaction with exported notes in the IDE (such as the one displayed in Figure F.5). We also plan to enhance our method based on participant feedback, especially on integrating audio recordings with speech-to-text transcription and VR-friendly search capabilities (e.g., also via a speech interface).

As a bottom line, it is crucial to interpret our findings within the current technological landscape. Our participants were navigating a novel technology and tool set. As VR technology becomes more mainstream and engineers become more accustomed to similar tools, we anticipate that the results and user experiences reported here will improve.

To conclude, we remain optimistic about the potential of VR methods to enhance collaborative software engineering practices.

Bibliography

- [1] Sandra Yin and Julia McCreary. Myths and realities: Defining re-engineering for a large organization. In *NASA. Goddard Space Flight Center, Proceedings of the Seventeenth Annual Software Engineering Workshop*, 1992.
- [2] Scott Tilley. A Reverse-Engineering Environment Framework: Technical report, Defense Technical Information Center, Fort Belvoir, VA, April 1998.
- [3] Harry Sneed and Chris Verhoef. Re-implementing a legacy system. *Journal of Systems and Software*, 155:162–184, September 2019.
- [4] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.
- [5] Pierre Caserta and O Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, July 2011.
- [6] Denis Graanin, Kreimir Matkovi, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, September 2005.
- [7] Christof Ebert, Marco Kuhrmann, and Rafael Prikladnicki. Global Software Engineering: Evolution and Trends. In *2016 IEEE 11th International Conference on Global Software Engineering (ICGSE)*, pages 144–153, Orange County, CA, USA, August 2016. IEEE.
- [8] Rainer Koschke and Marcel Steinbeck. Modeling, Visualizing, and Checking Software Architectures Collaboratively in Shared Virtual Worlds. 2021.
- [9] Rainer Koschke and Marcel Steinbeck. SEE Your Clones With Your Teammates. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pages 15–21, Luxembourg, October 2021. IEEE.
- [10] Alexander Krause-Glau, Marcel Bader, and Wilhelm Hasselbring. *Collaborative Software Visualization for Program Comprehension*. October 2022. Pages: 86.

-
- [11] Adrian Hoff, Christoph Seidl, Mircea Lungu, and Michele Lanza. Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2023.
 - [12] O. Greevy, M. Lanza, and C. Wyssseier. Visualizing Feature Interaction in 3-D. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, Budapest, Hungary, 2005. IEEE.
 - [13] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003. Conference Name: IEEE Transactions on Software Engineering.
 - [14] M. Balzer and O. Deussen. Hierarchy Based 3D Visualization of Large Software Structures. In *IEEE Visualization 2004*, pages 4p–4p, Austin, TX, USA, 2004. IEEE Comput. Soc.
 - [15] Danny Holten, Roel Vliegen, and Jarke J. Van Wijk. Visual Realism for the Visualization of Software Metrics. In *In Proceedings of Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 1–6. Springer, 2005.
 - [16] Danny Holten, Roel Vliegen, and Jarke van Wijk. Visualization of Software Metrics using Computer Graphics Techniques. January 2006.
 - [17] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics. page 7, 2004.
 - [18] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 119–130, Limassol, Cyprus, October 2022. IEEE.
 - [19] Roy Oberhauser and Carsten Lecon. Virtual Reality Flythrough of Program Code Structures. In *Proceedings of the Virtual Reality International Conference - Laval Virtual 2017 on - VRIC '17*, pages 1–4, Laval, France, 2017. ACM Press.
 - [20] Martin Misiak, Andreas Schreiber, Arnulph Fuhrmann, Sascha Zur, Doreen Seider, and Lisa Nafeie. IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality. In *2018 IEEE Working Conference*

- on Software Visualization (VISSOFT)*, pages 112–116, Madrid, September 2018. IEEE.
- [21] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205, London, UK, 2000. IEEE Comput. Soc.
- [22] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, page 921, Leipzig, Germany, 2008. ACM Press.
- [23] Juraj Vincur, Pavol Navrat, and Ivan Polasek. VR City: Software Analysis in Virtual Reality Environment. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 509–516, Prague, Czech Republic, July 2017. IEEE.
- [24] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. CityVR: Gameful Software Visualization. page 5, 2017.
- [25] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwrenaut. *Science of Computer Programming*, 79:204–223, January 2014.
- [26] Roberto Minelli and Michele Lanza. SAMOA A Visual Software Analytics Platform for Mobile Applications. In *2013 IEEE International Conference on Software Maintenance*, pages 476–479, September 2013. ISSN: 1063-6773.
- [27] Mohammad Alnabhan, Awni Hammouri, Mustafa Hammad, Mohammed Atoum, and Omamah Al-Thnebat. *2D visualization for object-oriented software systems*. April 2018. Pages: 6.
- [28] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, Ischia, Italy, 1998. IEEE Comput. Soc.
- [29] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 551, Waikiki, Honolulu, HI, USA, 2011. ACM Press.

- [30] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, April 2013.
- [31] David Moreno-Lumbreras, Jesus M Gonzalez-Barahona, and Andrea Villaverde. BabiaXR: Virtual Reality software data visualizations for the Web. 2021.
- [32] Adrian Hoff, Michael Nieke, and Christoph 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*, pages 1455–1458, Athens Greece, August 2021. ACM.
- [33] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Exploring software cities in virtual reality. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 130–134, Bremen, Germany, September 2015. IEEE.
- [34] Leonel Merino, Alexandre Bergel, and Oscar Nierstrasz. Overcoming Issues of 3D Software Visualization through Immersive Augmented Reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–64, Madrid, September 2018. IEEE.
- [35] Dussan Freire-Pozo, Kevin Cespedes-Arancibia, Leonel Merino, Alison Fernandez-Blanco, Andres Neyem, and Juan Pablo Sandoval Alcocer. DGT-AVisualizing Code Dependencies in AR. In *2023 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023.
- [36] Andreas Schreiber, Lisa Nafeie, Artur Baranowski, Peter Seipel, and Martin Misiak. Visualization of Software Architectures in Virtual Reality and Augmented Reality. In *2019 IEEE Aerospace Conference*, pages 1–12, Big Sky, MT, USA, March 2019. IEEE.
- [37] Craig Anslow, Stuart Marshall, James Noble, and Robert Biddle. Source-Vis: Collaborative software visualization for co-located environments. In *2013 First IEEE Working Conference on Software Visualization (VIS-SOFT)*, pages 1–10, Eindhoven, Netherlands, September 2013. IEEE.

- [38] Alexander Krause-Glau, Malte Hansen, and Wilhelm Hasselbring. Collaborative program comprehension via software visualization in extended reality. *Information and Software Technology*, 151:107007, November 2022.
- [39] Michael Moser, Josef Pichler, Gunther Fleck, and Michael Witlatschil. RbG: A documentation generator for scientific and engineering software. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 464–468, Montreal, QC, Canada, March 2015. IEEE.
- [40] Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. *Scribble: Closing the Book on Ad Hoc Documentation Tools*. 2009.
- [41] Vaclav Rajlich. Incremental Redocumentation Using the Web. *IEEE Software*, 17(5):102–106, September 2000.
- [42] Nicolas Anquetil, Káthia Oliveira, Anita Paulo, Laesse jr, and Susa Vieira. *A tool to automate re-documentation*. 2005.
- [43] Verena Geist, Michael Moser, Josef Pichler, Stefanie Beyer, and Martin Pinzger. *Leveraging Machine Learning for Software Redocumentation*. February 2020. Pages: 626.
- [44] Sugumaran Nallusamy, Meei Hao Hoo, and Farizuwana Akma Zulkifle. Controlled Experiment for Assessing the Contribution of Ontology Based Software Redocumentation Approach to Support Program Understanding. *Computing and Informatics*, 40(5):1025–1055, 2021.
- [45] Sebastian Baltes and Stephan Diehl. Sketches and Diagrams in Practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 530–541, November 2014. arXiv:1706.09172 [cs].
- [46] Uri Dekel and James D Herbsleb. *Notation and Representation in Collaborative Object-Oriented Design: An Observational Study*. 2007.
- [47] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

-
- [48] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, pages 336–343, Boulder, CO, USA, 1996. IEEE Comput. Soc. Press.
 - [49] Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software landscapes: Visualizing the structure of large software systems. In *IEEE TCVG*, 2004.
 - [50] M. Balzer and O. Deussen. Level-of-detail visualization of clustered graph layouts. In *2007 6th International Asia-Pacific Symposium on Visualization*, pages 133–140, Sydney, NSW, February 2007. IEEE.
 - [51] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.
 - [52] Sebastian Baltes and Paul Ralph. Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, 27(4):94, 2022.

Immersive Software Archaeology: Collaborative Exploration and Note Taking in Virtual Reality

Originally published in: Proceedings of the 46th International Conference on Program Comprehension (ICPC 2024), April 15–16, Lisbon, Portugal

Joint work with: Mircea Lungu, Christoph Seidl, and Michele Lanza

Abstract

Understanding software systems is a vital task, often undertaken by teams of engineers, for the development and maintenance of systems. Collaborative software visualization tools are essential in this context, yet they are limited. Existing tools, particularly in virtual reality, allow exploration but lack the crucial feature of note-taking, which is a significant limitation.

We present Immersive Software Archaeology (ISA), a virtual reality tool that enables engineering teams to collaboratively explore and comprehend software systems. Unique to ISA, it facilitates note-taking during exploration with virtual multimedia whiteboards that support freehand diagramming, audio recordings, and VR screenshots. Notes taken on these whiteboards are synchronized with an Integrated Development Environment (IDE), providing easy access to the results of a VR exploration while performing changes to the system's source code.

Video Demonstration – <https://youtu.be/32EIpf4V3b4>

G.1 Introduction and Related Work

Exploring and comprehending an unfamiliar software system, e.g., for re-engineering a legacy system, is a complex yet crucial task typically performed by teams of engineers [1, 2]. When done by reading through source code, the process is hampered the size and complexity of a system.

Software visualization can be beneficial in this scenario: by using visual metaphors to represent the structure, behavior, or evolution of a system, it provides software engineers with a comprehensive overview [3, 4].

Software visualizations vary in their metaphor (e.g., graphs [5, 6] or information cities [7, 8, 9]), dimensionality (2D [10, 11] or 3D), and – for 3D visualizations – display medium (standard screen [12, 13, 14] or virtual/augmented reality (VR/AR) [15, 16, 17, 18]).

Despite the availability of many software visualization tools, there is a scarcity of collaborative options, especially for remote settings. VR is a preferred medium for such settings [19, 20], but a major drawback of existing VR visualizations is their lack of support for note-taking during exploration, risking the loss of insights.

We introduce Immersive Software Archaeology (ISA), a collaborative VR software visualization tool. Based on automated system analysis, ISA allows software engineering teams to collaboratively explore a system's structure in immersive VR using an interactive visualization that is synchronized over the internet. Engineers can record their thoughts and insights on collaborative multimedia whiteboards during their VR exploration - which is not possible with existing VR tools. Post-exploration, these notes are accessible in the Integrated Development Environment (IDE) Eclipse, aiding in the implementation of changes to the system's source code.

G.2 Collaborative Software Exploration in VR

We present the collaborative virtual reality software visualization tool Immersive Software Archaeology (ISA) in its current version 2.1¹. ISA is comprised of two main components: a model server that integrates with the widely used open-source development environment Eclipse, and a VR visualization client. We discuss how users interact with both components, and follow with an overview of the relevant aspects of its architecture.

¹<https://gitlab.com/immersive-software-archaeology>

G.2.1 Usage from a User's Point of view

Figure G.1 offers an overview of ISA, illustrated through screenshots captured from the perspective of a user. The process of using ISA begins within Eclipse, where users initiate an automated procedure to generate the information necessary to visualize a selected subject system (a and b). Once that is completed, multiple users are able to jointly explore the subject system in a collaborative VR visualization and record multimedia notes about their observations and insights (c to d). Finally, notes taken in VR are accessible in Eclipse, allowing users to review them while making modifications to the source code of the system under study, as indicated in e, f.

Automated Model Generation in Eclipse

Users start ISA's automated model generation process via an entry in the Eclipse project explorer's context menu a. This process involves multiple steps (see Section G.2.2), for which users choose between alternative implementations depending on the ISA Eclipse plugins installed b.

Upon completion of the model generation process, the results are persisted and users make them available for VR visualization clients by launching ISA's model server, either through a confirmation dialog that pops up after the model generation process or by using a control panel view in the Eclipse UI.

Collaborative Exploration in Virtual Reality

Once the ISA Eclipse model server is reachable, multiple users can connect to it via a local network or the internet using the ISA VR client running on a head-mounted VR device. Once connected, they have the option to select and load a system from the range of those analyzed in the connected ISA Eclipse model server.

After the loading phase completes, users enter in a real-time, synchronized visualization environment of the selected system where they can view the virtual representations of fellow collaborators, observing each other's interactions with elements of the visualization as detailed in the subsequent sections.

Folder Spheres ISA visualizes the folder-level structure of a subject system in form of nested semi-transparent folder spheres with different colors. These folder spheres are initially in a closed state, as shown in c. Users can interact

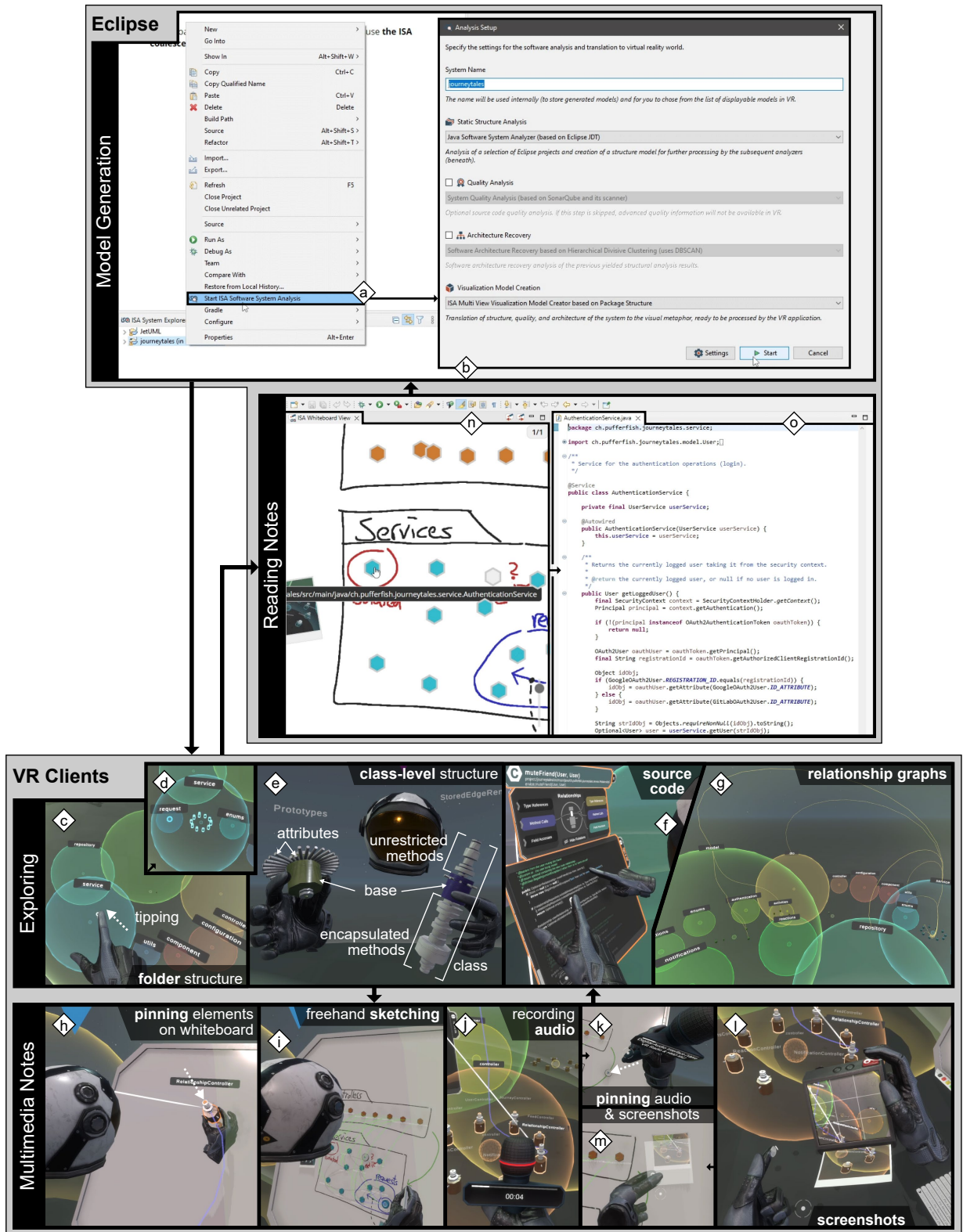






Figure G.1: Overview of ISA's collaborative VR software exploration and note taking approach from a user's point of view.



with them by tapping on them to open and reveal the contents within, compare  and . This interaction promotes a top-down approach to exploring the system's structure [21]. Additionally, ISA enables users to utilize hand gestures to manipulate the scale of the visualization and to move it within the virtual space, providing an additional form of navigation besides the standard click-and-point VR teleportation mechanisms.

Class Cylinders Positioned within folder spheres, ISA visualizes the class-level elements of a subject system as stacks of cylinders  consisting of four parts: (1) a base cylinder colored according to the containing folder sphere's color, (2) cylinders representing methods without access restrictions (e.g., public in Java), (3) cylinders representing encapsulated methods (e.g., private, protected, or package visible in Java), and (4) spikes originating from the base cylinder represent attributes.


The height of method cylinders is proportional to the number of expressions contained in the represented method while their radius is proportional to its cognitive complexity as measured by the metric proposed by Campbell  [22]. Attribute spikes vary in length depending on whether they are encapsulated (short spike) or accessible without restrictions (long spike).

With the above, ISA encodes a summary of structural class metrics into the shape of class cylinders and their constituents. Users can visually comprehend these already before reading code. Due to their cylindrical shape and symmetrical layout, class cylinders' outline is independent of the viewing angle, an aspect particularly relevant in a collaborative setting.

Relationship Edges In ISA, tapping on the visual representations of folders, classes, methods, or attributes enables users to access a user interface that provides detailed information about the tapped element. This user interface is interactive and can be repositioned by the users as needed. All interactions with it are synchronized in real time among all users, enhancing the collaborative experience.


Figure G.1  depicts the user interface for a selected method. Its upper section features controls for a relationship visualization that represents references to or from a software element (e.g., a method) in the subject system's source code. It distinguishes between different types of references into type references, method calls, and field accesses. Users can dynamically display or hide these references for a selected element. For example,  shows type references originating from a selected class. The relationship visualization is

available for all types of elements, including folders, where it aggregates references coming from or going to their constituent elements.


Source Code When opening the user interface for an element containing direct source code (i.e., a class-level element, method, or attribute), it displays a scrollable view of the source code at its lower section, as shown in . In combination, the relationship graph and code view provide a comprehensive and interactive means for users to explore and understand the code details and structural relationships between them in a subject system.


Collaborative Note Taking in Virtual Reality

Users can spawn virtual whiteboards to take notes during VR exploration sessions. These whiteboards allow for pinning elements from the visualization, creating freehand sketched diagrams, and attaching audio recordings as well as screenshots taken in VR.

Pinning Software Elements Users can grab folder spheres and class cylinders from the visualization and pin them on a virtual whiteboard via a respective hand gesture (indicated in ). When an element is pinned to the whiteboard, a corresponding pin appears. Users can manipulate these pins by moving them around or accessing an interface with additional information, such as a list of referenced classes.

This user interface enables to replace pins representing folders by pins for class-level elements and sub-folders they contain, arranged in a circular layout. This feature allows users to navigate the hierarchical structure of the software system.

Relationships between Pinned Elements If a pinned software element references another element also pinned on the same whiteboard, a curved line is drawn between their pins to show this connection . These relation lines vary in thickness depending on how many source code references they represent, e.g., when showing relations between two folder pins.

Drawing Freehand Diagrams with Automated Conformance Checks Users can pick up a virtual pen and draw freely on the whiteboards using a variety of colors . Based on different drawing modes of the virtual pen, our tool distinguishes user's pen strokes into (1) uninterpreted drawing (for icons, text, etc.),

(2) outlines around pins, and (3) arrows between outlines. Based on that, relation lines between pins are colored according to the freehand drawn arrows between outlines, providing users with a check on the conformance between their hand drawn arrows and the ground truth references in the subject system's source code [23].

Recording Audio To capture elaborate thoughts, users can pick up a virtual microphone and create arbitrarily long audio recordings $\langle j \rangle$. Once completed, they pin their audio recording to a whiteboard using the same gesture as for pinning software elements $\langle k \rangle$.

Capturing VR Screenshots To capture a specific view on the visualization, users can pick up a virtual camera and take VR screenshots $\langle l \rangle$. These can then be pinned to a whiteboard as shown in $\langle m \rangle$. When tipping on a VR screenshot pinned to a whiteboard, users can restore the visualization to the state of taking the picture, implementing a form of temporal snapshot in the exploration process.

Reading Notes in Eclipse

To assist users in working in the source code based on the insights gained during VR explorations, they can access their whiteboard notes directly in Eclipse. For that purpose, ISA extends the Eclipse UI with a view enabling users to inspect whiteboards, zoom in and out, play audio recordings, and enlarge screenshots $\langle n \rangle$. Further, users can open files of pinned elements in Eclipse by clicking on pins in the whiteboard view $\langle o \rangle$.

G.2.2 Tool Architecture

Figure G.2 provides a simplified overview of ISA's core components and their interconnections.

Model Server: Platform of Eclipse Plugins

ISA's model server is implemented as an extensible platform of Eclipse bundles. The lower part of Figure G.2 provides an overview of that platform.

Automated Model Generation ISA's overall model generation process is handled by a core plugin which successively executes a pipeline of three steps, each passing its results on to the next:

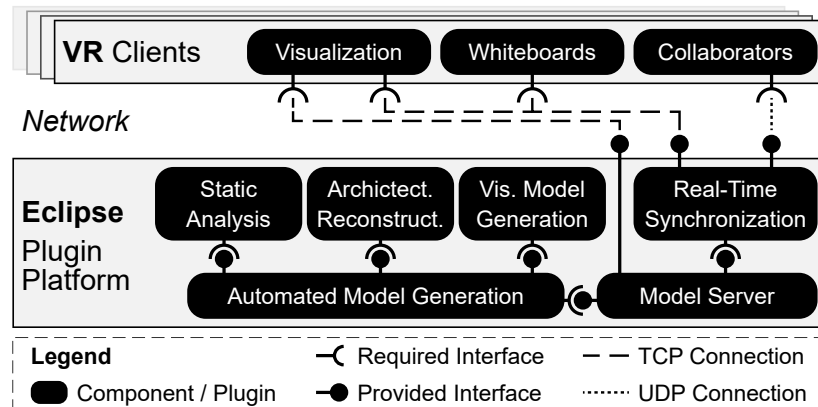


Figure G.2: Architectural overview of ISA. A platform of Eclipse plugins provides an automated system analysis and a model server. VR clients connect to it and display a collaborative visualization.

1. analyzing Eclipse projects selected by a user and storing the results in a model suitable for capturing the structure of an object-oriented software system from folder to member level,
2. (optional step) employing an architecture recovery procedure [24] that replaces the folder-level structure in the results of Step 1,
3. transforming the software model resulting from the previous step into a visualization model as input for VR clients.

A concrete solution for a step, e.g., an analysis for Java source code as Step 1, is implemented in form of an Eclipse bundle registering with the ISA core plugin via an extension point. ISA's model generation platform can be extended with alternative solutions for individual steps, e.g., support for analyzing an additional programming language (Step 1) or an alternative mapping of software elements to visual elements (Step 3), while integrating with pre-existing solutions for other steps without further adaptations.

To further ease the extension of ISA's model generation platform, we define the structure of information passed between its steps via meta models, using the Eclipse Modeling Framework (EMF).

Communication via Network

To exchange information between Eclipse server and VR clients, ISA uses two channels:

1. *UDP Channel* The ISA server establishes a dedicated UDP connection with each VR client for continuous sharing of position and rotation data for users' heads, hands, and interacted objects such as whiteboards, class cylinders, or cameras. This system does not implement additional measures to recover dropped network packets, as any lost data is overridden by the subsequent successful transmission's updated information.

2. *TCP Channel* For information exchange that is less time critical but that requires reliable and ordered message delivery (e.g. showing relations via the relationship graph or closing folder sphere), the ISA model server provides an HTTP-based interface. Upon receiving events from a connected client, the server (1) verifies their consistency with a log of all pre-existing events, (2) persists a new version of the log with the inserted events, and (3) forwards the new events in their respective order to all connected clients.

We use the Eclipse Modeling Framework (EMF) to define meta models for the data structures in our TCP-based message exchange, to automatically generate equivalent code from the meta models in both Java (for use in Eclipse bundles) and C# (for use in the VR visualization client).

VR Visualization Clients

ISA's VR visualization client acts as local realization of the synchronized visualization state maintained by the ISA Eclipse server. That includes interactions carried out directly by the user of an ISA VR client – these first go through the model server and its verification before being sent back and then being applied in the order consistent with potential other events issued by collaborators in the meantime.

ISA's VR visualization client is based on the SteamVR platform² and implemented in C# using the Unity 3D engine³, making ISA's VR client compatible with all VR hardware supported by SteamVR.

G.3 Case Study with Practitioners

We evaluated our tool in an exploratory case study with four software engineering practitioners. Working in pairs, participants used our tool to collaboratively explore an unfamiliar Java subject system. After these sessions, we

²<https://store.steampowered.com/steamvr>

³<https://unity.com/>

used a questionnaire to collect participants' feedback on using VR for exploring an unfamiliar software system and taking notes on findings. Further, we analyzed the whiteboards created by both teams during their session and extracted all statements about the subject system they have noted. We then relayed these statements to the original developers of the subject system to assess their correctness and relevance in a re-engineering context. Below, we discuss general results of the study. More detailed documents and raw data is accessible in an online appendix⁴.

Feedback from the subject system's original developers show that, while the relevance of participants' statements was mixed with some being vital for future work with the system's source code and others not concerning relevant aspects at all, the correctness of participants' statements was very high. Results from the analysis of the VR sessions and post-questionnaire show that ISA provides good support for an exploration on the level of architectural elements (folders and classes). Regarding note taking, participants valued the flexible nature of using freehand scribbling, recording audio, and taking screenshots. At the same time, they pointed out that it requires practice to fully utilize the immersive VR tool, especially for handwriting on the VR whiteboards. Further, they mentioned potential of an automated audio recording transcription feature for the virtual whiteboards.

G.4 Conclusion and Future Work

We presented the collaborative VR software exploration tool Immersive Software Archaeology (ISA). We described its usage from a user's points of view and provided an overview of its architecture. Further, we reported on results from an exploratory case study with four software engineering practitioners.

In future work, we plan to extend ISA with support for automated audio-to-text transcription. Further, we plan to conduct quantitative studies involving larger samples of software engineering practitioners to investigate the tool's effectiveness, e.g., comparing ISA with traditional software exploration environments such as an IDE.

Bibliography

- [1] Sandra Yin and Julia McCreary. Myths and realities: Defining re-engineering for a large organization. In *NASA. Goddard Space Flight Cen-*

⁴<https://doi.org/10.6084/m9.figshare.24499726>

- ter, Proceedings of the Seventeenth Annual Software Engineering Workshop*, 1992.
- [2] Harry Sneed and Chris Verhoef. Re-implementing a legacy system. *Journal of Systems and Software*, 155:162–184, September 2019.
 - [3] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.
 - [4] Denis Graanin, Kreimir Matkovi, and Mohamed Eltoweissy. Software visualization. *Innovations in Systems and Software Engineering*, 1(2):221–230, September 2005.
 - [5] O. Greevy, M. Lanza, and C. Wyseier. Visualizing Feature Interaction in 3-D. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6, Budapest, Hungary, 2005. IEEE.
 - [6] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003. Conference Name: IEEE Transactions on Software Engineering.
 - [7] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205, London, UK, 2000. IEEE Comput. Soc.
 - [8] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. In *Companion of the 13th international conference on Software engineering - ICSE Companion '08*, page 921, Leipzig, Germany, 2008. ACM Press.
 - [9] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. CityVR: Gameful Software Visualization. page 5, 2017.
 - [10] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. Evolutionary and collaborative software architecture recovery with Softwareonaut. *Science of Computer Programming*, 79:204–223, January 2014.
 - [11] Roberto Minelli and Michele Lanza. SAMOA A Visual Software Analytics Platform for Mobile Applications. In *2013 IEEE International Conference on Software Maintenance*, pages 476–479, September 2013. ISSN: 1063-6773.

- [12] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, Ischia, Italy, 1998. IEEE Comput. Soc.
- [13] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: a controlled experiment. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 551, Waikiki, Honolulu, HI, USA, 2011. ACM Press.
- [14] Frank Steinbrückner and Claus Lewerentz. Understanding software evolution with software cities. *Information Visualization*, 12(2):200–216, April 2013.
- [15] Adrian Hoff, Michael Nieke, and Christoph 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*, pages 1455–1458, Athens Greece, August 2021. ACM.
- [16] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. Exploring software cities in virtual reality. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 130–134, Bremen, Germany, September 2015. IEEE.
- [17] David Moreno-Lumbreras, Jesus M Gonzalez-Barahona, and Andrea Villaverde. BabiaXR: Virtual Reality software data visualizations for the Web. 2021.
- [18] Dussan Freire-Pozo, Kevin Cespedes-Arancibia, Leonel Merino, Alison Fernandez-Blanco, Andres Neyem, and Juan Pablo Sandoval Alcocer. DGT-AVisualizing Code Dependencies in AR. In *2023 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2023.
- [19] Rainer Koschke and Marcel Steinbeck. SEE Your Clones With Your Teammates. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*, pages 15–21, Luxembourg, October 2021. IEEE.
- [20] Alexander Krause-Glau, Marcel Bader, and Wilhelm Hasselbring. *Collaborative Software Visualization for Program Comprehension*. October 2022. Pages: 86.

-
- [21] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.
 - [22] G Ann Campbell. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*, pages 57–58, 2018.
 - [23] Adrian Hoff, Christoph Seidl, Mircea Lungu, and Michele Lanza. Preparing Software Re-Engineering via Freehand Sketches in Virtual Reality. In *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2023.
 - [24] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 119–130, Limassol, Cyprus, October 2022. IEEE.