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

Adrian Hoff
IT University of Copenhagen
Copenhagen, Denmark
adho@itu.dk

Lea Gerling
Universität Hildesheim
Hildesheim, Germany
gerling@sse.uni-hildesheim.de

Christoph Seidl
IT University of Copenhagen
Copenhagen, Denmark
chse@itu.dk

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

***Index Terms*—Software Visualization, Virtual Reality, Software Architecture Recovery, Empirical Software Engineering**

I. 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 [7, 30]. The exploration of a software system ideally starts with the system's architecture [15] 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 [25, 30], 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 [6].

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 [22, 27, 24]. 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 [21], 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 II, we discuss the state of the art in software visualization regarding (its lack of) incorporating software architecture knowledge. In Section III, we describe our method for software architecture recovery (SAR) and how we incorporate its results into an immersive 3D virtual reality representation. In Section IV, 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 V, we close with a conclusion and an outlook on future work.

II. State of the Art

A software visualization provides a visual overview of a subject system [20]. Depending on the purpose of the visualization, it may encompass a system's structure, behavior, evolution, or quality [11, 33]. 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 [33, 1]. 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 [9, 10, 16, 20, 1, 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 [27, 37, 19, 35, 12, 32, 5, 22, 34], 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. [4, 2, 3] 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.

III. 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 1 shows an overview of that. As

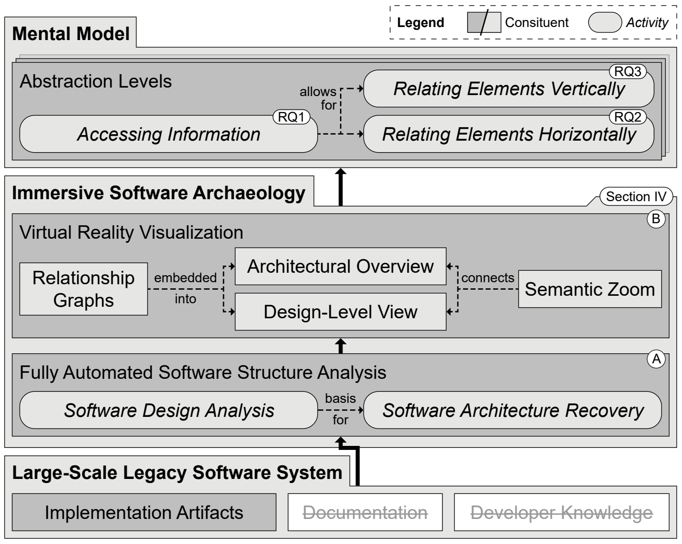


Fig. 1. Overview of our method for utilizing software architecture recovery to provide a visualization with semantic zoom along multiple abstraction levels.

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 Ⓐ in Figure 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 Ⓑ in Figure 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.

A. Automated Software Structure Analysis

Our method encompasses an automated analysis of a system's static structure (cf. Ⓐ in Figure 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 III-A1). The second step is a software architecture recovery procedure based on the results of the recovered design of the system (Section III-A2).

*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 appendix[1]. For members with statement bodies, the software design analysis gathers metrics such as their respective number of expressions and cognitive complexity [8]. 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.

*2) **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 [18]. 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 [25], 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 no 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.

[1]https://gitlab.com/immersive-software-archaeology/publication-vissoft22

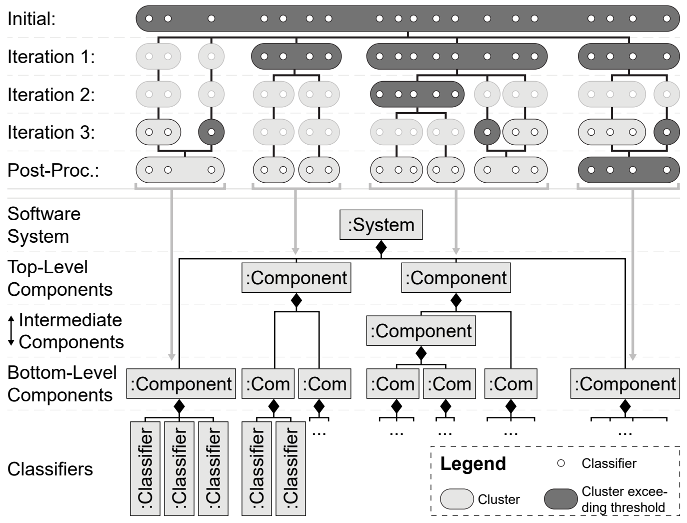


Fig. 2. Upper area: Simplified representation of our exemplary software clustering procedure. Similarities between classifiers are represented via spatial distances. Lower area: Representation of the respective clustering results when populated in a software structure model (member-level structure omitted). 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).

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 2 depicts an example application of that technique. Initially, it groups all classifiers of a system in one root cluster (cf. first row in the upper area of Figure 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 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" [13]), 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

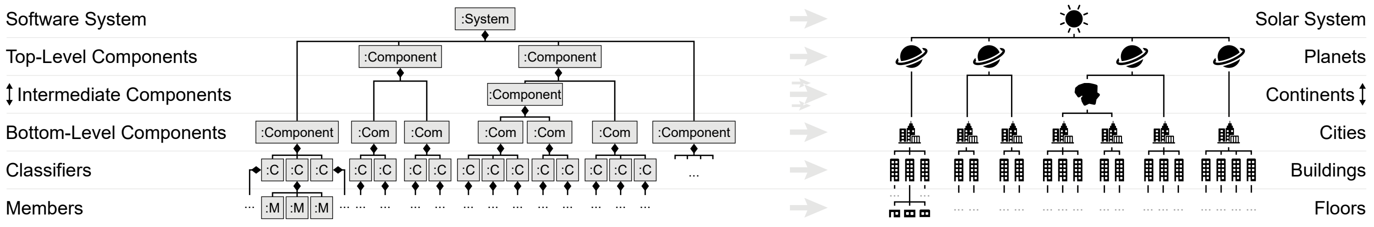


Fig. 3. Transformation of a software structure model to our solar system metaphor. The illustration continues the example given in Figure 2.

clusters (cf. row 3 in the upper area of Figure 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 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 2.

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. It extracts words based on typical naming conventions, e.g., "exampleName" results in labels "example" and "name".

B. Immersive Virtual Reality Visualization

We present a 3D software visualization method that builds upon the software structure analysis presented in Section III-A 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 Ⓑ in Figure 1. In the following, we elaborate on these concepts, backed up by examples from our prototype implementation Immersive Software Archaeology (ISA).

1) ***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 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 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 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 ⓐ and ⓑ show the overview on system level. Screenshot ⓒ shows a close-up view of the surface of a planet, where cities form continents according to the represented component hierarchy.

2) ***Semantic Zoom:*** When entering our visualization, a user is initially presented with the architectural overview of a subject system (cf. upper screenshots in Figure 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 ⓒ depicts it.

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 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 particularly 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 [8]. 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 ⓕ in Figure 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 II). 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 4. While the user is located in the design-level view, the architectural overview additionally highlights the currently visited city. In

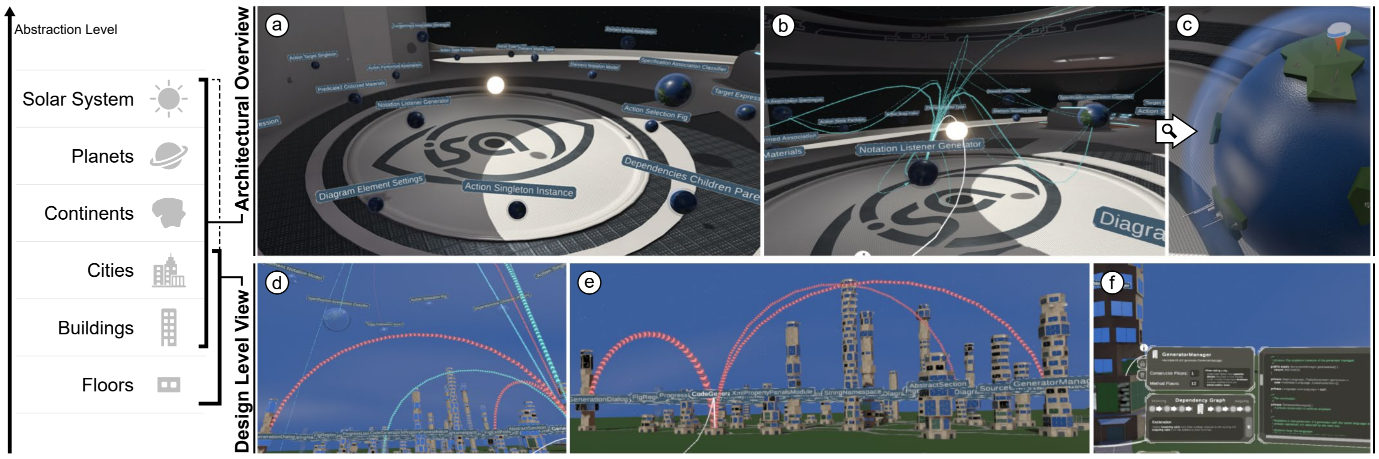


Fig. 4. Screenshots of our prototype implementation ISA showing an example system (∼1.800 classifiers) on its architectural level (upper area) and semantically zoomed in on a bottom-level component (lower area). The left hand side maps the semantic zoom levels to the primarily visualized elements.

our prototype implementation, this is achieved via an orange arrow as depicted in Screenshot ⓒ in Figure 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 ⓓ). 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.

*3) **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 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 ⓑ in Figure 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

TABLE I
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 [23, 29]	✗	✓	✓
VR FlyThruCode [26]	✓	✗	✓
VR City [34]	✓	✗	✓
SEE / EvoStreets [31]	✓	✗	✓
ExplorViz [14]	✓	✓	✗
CityVR [22]	✓	✓	✓

through the visualization's visual hierarchy, such as the blue lines shown in Screenshot ⓑ. 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 ⓓ. Thereby, our method embeds the visualization of reference relations into the different levels of architectural abstraction and across the semantic zoom.

*4) **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 [22, 27, 24, 36]. 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 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 ⓒ), 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 4 ⓕ) 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 [28], 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.

IV. 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.

RQ_1: In what sense do the different tools facilitate *accessing information* on software elements such as methods, classes, or components?

RQ_2: In what sense do the different tools facilitate *establishing horizontal relations* between software elements on the same abstraction level?

RQ_3: In what sense do the different tools facilitate *establishing vertical relations* between software elements across different abstraction levels?

A. *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 [22] or Softwarenaut [17].

B. *Software Exploration Tools*

We implement our method in a VR visualization tool called Immersive Software Archaeology[2] (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 Eclipse[3] 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.
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 I gives an overview of existing VR software visualization tools and their respective fulfillment of our inclusion criteria. We pre-filtered existing tools that do not visualize Java code or that do not come with native VR support. As CityVR by Merino et al. [22] is the only tool that fulfills all our criteria, we include it as a representative for state-of-the-art

[2]https://gitlab.com/immersive-software-archaeology
[3]https://www.eclipse.org/

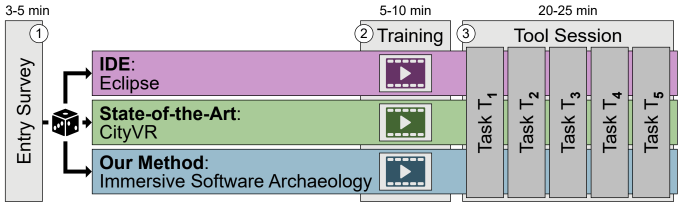


Fig. 5. Overview of the experiment procedure.

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.

C. *Experiment Procedure*

We divided the experiment into three phases as depicted in Figure 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 appendix[1]. Each experiment run takes ca. 35 to 45 minutes.

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

② *Tool-Specific Training.* 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.

③ *Tool Session.* 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 catalogue[1]. 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.

D. *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 II lists a shortened version of the tasks investigated throughout the tool session, along with the research questions and exploration activity they address.

In Task T_1, participants access design level information by seeking a specific Java class, given a description of its functionality (RQ_1). In Task T_2, 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 T_1 (RQ_1 & RQ_2). While solving T_1, participants encounter the interface they will search in T_2 (without knowing it) via references in code. We keep track of whether they notice this in Task T_2. As the solutions for both tasks T_1 and T_2 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 T_2, 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
- The class (T_1) maintains instances of the interface (T_2)
- ... and maps these to meta information

In 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:

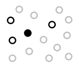
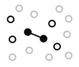
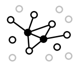
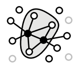
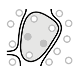
- Participant finds no related classes
- Participant finds only a limited set of classes, e.g., only classes in the same package, city district, or planet
- Participant finds classes all across the system

In 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 II 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.

E. *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.

TABLE II
SHORTENED VERSION OF THE TOOL SESSION TASKS OF OUR EXPERIMENT. THE COMPLETE TASK SHEET IS ACCESSIBLE VIA OUR ONLINE APPENDIX[1].

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 and 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.	

F. 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 6 and detailed results are available via our online appendix[1]. To shorten explanations, we refer to participant groups for individual tools via abbreviations: $\text{group}_{\text{IDE}}$ (Eclipse), $\text{group}_{\text{SOTA}}$ (state-of-the-art visualization CityVR), $\text{group}_{\text{ISA}}$ (Immersive Software Archaeology; our implementation).

***RQ_1**: Accessing Information (T_1 & T_2).* To solve T_1, $\text{group}_{\text{IDE}}$ 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 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 T_2, where our method relieved participants of finding relevant software elements for the most part.

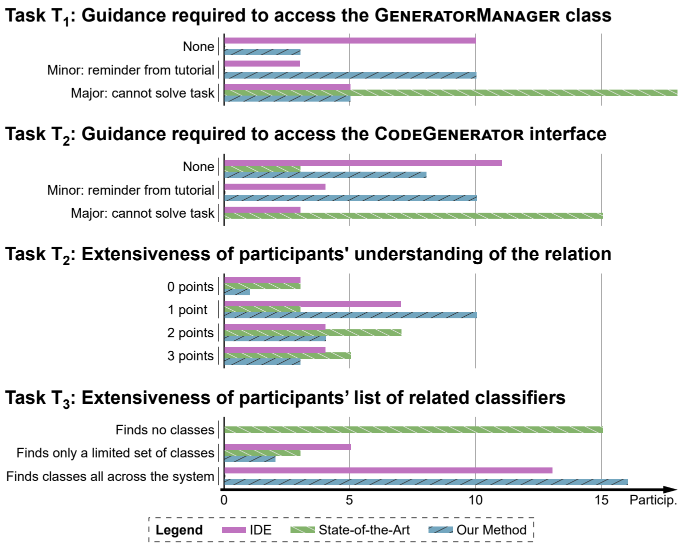


Fig. 6. Quantitative evaluation results from the different experiment phases.

***RQ_2**: Relating Information Horizontally.* We investigate RQ_2 via T_2 and T_3 on design level and via T_5 on architecture level.

Design Level. To solve T_2, participants in group$_{\text{IDE}}$ generally related elements via reading code. Only 1 participant used Eclipse's call hierarchy feature. In contrast, to solve T_3, 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 6). We observe that participants in group$_{\text{IDE}}$ 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 T_3 by deleting elements, recompiling the system, and inspecting all files with compilation errors.

Participants in group$_{\text{SOTA}}$ relied on reading code because the respective tool does not visualize relations specifically. While 15 participants did not solve T_3, they generally solved T_2 in more detail than participants using other tools (see Figure 6).

To solve T_2, 11 participants in group$_{\text{ISA}}$ 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$_{\text{ISA}}$ answered T_2 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 T_3, all participants in group$_{\text{ISA}}$ (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 T_5, we generally notice that group$_{\text{IDE}}$ based their explanations on code constructs (mainly classes) while group$_{\text{SOTA}}$ based their answers on visual elements (buildings and city districts). On the other hand, while most participants in group$_{\text{ISA}}$ use code constructs (mainly classes), some mix them with visual elements (continents and planets) when describing architectural structure.

Similar to group$_{\text{IDE}}$, multiple participants in group$_{\text{SOTA}}$ answer T_5 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$_{\text{ISA}}$ 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$_{\text{IDE}}$ mentioned that these features were not ideal for a broad investigation of relations on classifier level such as in T_3. 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 T_3. On architecture level (T_5), 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$_{\text{SOTA}}$ as compared to group$_{\text{ISA}}$ in T_2 where the elements to relate where known and available, but significantly worse results in T_3 where the elements to relate were unknown (cf. Figure 6). This translates to architectural level, i.e., because they could not solve T_3, group$_{\text{SOTA}}$ reported a lack of overview when solving T_5.

With the relationship graphs provided by our method, participants in group$_{\text{ISA}}$ across all programming experience levels were able to relate elements 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 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$_{\text{ISA}}$ generally provided better arguments (see above) for their answers to T_5 as compared to group$_{\text{IDE}}$ and group$_{\text{SOTA}}$, indicating that they developed a better overview of the system's architecture.

***RQ_3**: Relating Information Vertically.* To solve T_4, participants in group$_{\text{IDE}}$ 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 $\text{group}_{\text{IDE}}$ solved T_4 thoroughly by reading through several classifiers in a bottom-up approach.

Participants in $\text{group}_{\text{SOTA}}$ 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 $\text{group}_{\text{ISA}}$ 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 $\text{group}_{\text{ISA}}$ vertically related classifiers to the asked component based on relationships with the provided core classifiers. However, similar to $\text{group}_{\text{IDE}}$, 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 $\text{group}_{\text{IDE}}$ 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.

G. 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 programming[1]. 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 tool[1].

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.

V. 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.

Acknowledgements

This work is supported by the DFF (Independent Research Fund Denmark) within the project ISA – Immersive Software Archaeology (0136-00070B).

References

[1] Vladimir Averbukh et al. “Metaphors for software visualization systems based on virtual reality”. In: *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*. 2019.

[2] Michael Balzer and Oliver Deussen. “Hierarchy based 3D visualization of large software structures”. In: *IEEE Visualization 2004*. 2004.

[3] Michael Balzer and Oliver Deussen. “Level-of-detail visualization of clustered graph layouts”. In: *6th International Asia-Pacific Symposium on Visualization*. 2007.

[4] Michael Balzer et al. “Software landscapes: Visualizing the structure of large software systems”. In: *IEEE TCVG*. 2004.

[5] David Baum et al. “GETAVIZ: generating structural, behavioral, and evolutionary views of software systems for empirical evaluation”. In: *IEEE Working Conference on Software Visualization (VISSOFT)*. 2017.

[6] Laure Bedu, Olivier Tinh, and Fabio Petrillo. “A tertiary systematic literature review on software visualization”. In: *Working Conference on Software Visualization (VISSOFT)*. 2019.

[7] Robert Behling, Chris Behling, and Kenneth Sousa. “Software re-engineering: concepts and methodology”. In: *Industrial Management & Data Systems* 96.6 (1996).

[8] G Ann Campbell. “Cognitive complexity: An overview and evaluation”. In: *Proceedings of the 2018 international conference on technical debt*. 2018.

[9] Sheelagh Carpendale and Yaser Ghanam. *A survey paper on software architecture visualization*. Tech. rep. University of Calgary, 2008.

[10] P. Caserta and O Zendra. “Visualization of the Static Aspects of Software: A Survey”. In: *IEEE Transactions on Visualization and Computer Graphics* (2011).

[11] Noptanit Chotisarn et al. “A systematic literature review of modern software visualization”. In: *Journal of Visualization* 23.4 (2020).

[12] Philippe Dugerdil and Sazzadul Alam. “Execution trace visualization in a 3D space”. In: *Fifth International Conference on Information Technology: New Generations (itng 2008)*. 2008.

[13] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise.” In: *kdd*. Vol. 96. 34. 1996.

[14] Florian Fittkau et al. “Live trace visualization for comprehending large software landscapes: The ExplorViz approach”. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013.

[15] Wilhelm Hasselbring. “Software Architecture: Past, Present, Future”. In: *The Essence of Software Engineering*. 2018.

[16] Taimur Khan et al. “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] Mircea Lungu, Michele Lanza, and Oscar Nierstrasz. “Evolutionary and collaborative software architecture recovery with Softwarenaut”. In: *Science of Computer Programming* 79 (2014).

[18] Thibaud Lutellier et al. “Comparing Software Architecture Recovery Techniques Using Accurate Dependencies”. In: *IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015.

[19] Jonathan I Maletic et al. “Visualizing object-oriented software in virtual reality”. In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 2001.

[20] Anna-Liisa Mattila et al. “Software visualization today: systematic literature review”. In: *Proceedings of the 20th International Academic Mindtrek Conference*. 2016.

[21] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. “Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery.” In: *STRAW*. Vol. 3. 2003.

[22] Leonel Merino et al. “CityVR: Gameful software visualization”. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017.

[23] Martin Misiak et al. “IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality”. In: *IEEE Working Conference on Software Visualization (VISSOFT)*. 2018.

[24] David Moreno-Lumbreras et al. “CodeCity: On-screen or in virtual reality?” In: *Working Conference on Software Visualization (VISSOFT)*. 2021.

[25] Michael L Nelson. “A survey of reverse engineering and program comprehension”. In: *arXiv preprint cs/0503068* (2005).

[26] 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.

[27] Simone Romano et al. “On the use of virtual reality in software visualization: The case of the city metaphor”. In: *Information and Software Technology* 114 (2019).

[28] Paola Salomoni et al. “Assessing the efficacy of a diegetic game interface with Oculus Rift”. In: *13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. 2016.

[29] Andreas Schreiber et al. “Visualization of Software Architectures in Virtual Reality and Augmented Reality”. In: *2019 IEEE Aerospace Conference*. 2019.

[30] Harry Sneed and Chris Verhoef. “Re-implementing a legacy system”. In: *Journal of Systems and Software* 155 (2019).

[31] Marcel Steinbeck, Rainer Koschke, and Marc O Rüdel. “How evostreets are observed in three-dimensional and virtual reality environments”. In: *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020.

[32] Frank Steinbrückner and Claus Lewerentz. "Understanding software evolution with software cities". In: *Information Visualization* 12.2 (2013).

[33] Alfredo R Teyseyre and Marcelo R Campo. "An overview of 3D software visualization". In: *IEEE transactions on visualization and computer graphics* 15.1 (2008).

[34] 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.

[35] 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.

[36] Enes Yigitbas et al. "Collaborative Software Modeling in Virtual Reality". In: *ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2021.

[37] Peter Young and Malcolm Munro. "Visualising software in virtual reality". In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. 1998.