Visualization of Evolution of Component-Based Software Architectures in Virtual Reality

Elke Franziska Heidmann*, Lynn von Kurnatowski†, Annika Meinecke†, and Andreas Schreiber†
German Aerospace Center (DLR)
Institute for Software Technology
*heidmann.elke@t-online.de, †forename.surname@dlr.de

***Abstract*—The visualization of software architectures by the use of effective and feasible visual metaphors provides an intuitive approach to comprehend the implemented architecture of a software project. In this regard not only the visualization of the latest status of the implemented architecture is important, but also the visualization of the history of software architectures. Such visualizations show dependencies and contexts in which design decisions were made. Resulting information supports programmers to understand systems and to recognize disadvantageous design decisions. A software which is particularly suited for the visualization of component-based software is IslandViz. This software visualizes OSGi-based software architectures in Virtual Reality with an island metaphor, but at this point the history of an architecture is not taken into account. In this paper we present how IslandViz can be extended to visualize the history of software architectures of OSGi-based software projects. For this purpose we use algorithms for dynamic graphs to realize a dynamic positioning of the islands and an adaptable layout of the regions on the islands. The aim is to ensure that the user's orientation in the virtual environment is preserved even if elements of the visualization must adapt due to changes in the software history.**

***Index Terms*—software visualization, software architecture, software evolution, history of software, virtual reality**

I. Introduction

Software systems are becoming increasingly complex. For this reason, software development also requires a form of construction plan like in engineering disciplines. The software architecture is often considered as such a construction plan for software projects that displays the essential system properties and acts as a description to understand the structure and behavior of a system. Especially for large and complex software projects the software architecture is an important communication vehicle, a basis for a mutual understanding between all involved persons, for decision making and for communication about the system. However, over time, the original construction plan can no longer be used for communication and understanding of the developing system. This is because software products must be continuously modified over the course of their life cycle, since the environment in which they are used changes over time [1]. Moreover the scope of the software increases because of change requests from the user and expected functionality extensions. ISO/IEC-14764 [2] states further reasons for changes on software projects like the correction of errors and the improvement of the software, e.g. in terms of performance and maintainability. The term "software history" describes these changes and enhancements that are made during the development and maintenance phase of a software project. The modifications increase the complexity of the software because additional unstructured dependencies are added gradually. At the same time, the challenge increases that all those involved in the software development can keep an overview of the system despite continuous changes [1].

As an approach to software projects and their architecture, graphical representations are usually used, in which the entities of the software are represented by simple geometric shapes. The UML diagram is the most popular example of this. However, the simple, geometric representation of large software projects can become too complex using UML diagrams. Other alternatives are the use of complex graphical elements, three-dimensional visualizations, metaphors or representations in Virtual Reality (VR). The use of the third dimension in visualizations offers a higher information level. Using metaphors to represent software systems give the viewer a more intuitive way to access information. This happens through objects known from everyday life, for example the city metaphor [3], [4] or the solar system [5].

However, existing software visualizations mainly focus on a specific point in time to represent the software architecture without considering its history. Rebuilding large software architectures step-by-step can give important insights. It helps in understanding all dependencies and gives context in which changes occurred. This makes it possible to recognize disadvantageous design decisions and to rectify them. Consequently, it is helpful to visualize the time aspect of a software project because relevant information can be found in the software history. The software IslandViz [6], [7], which is particularly suited for the representation of modular software doesn't support the representation of software history until now. IslandViz visualizes the software architecture of a software system based on OSGi (Open Service Gateway Initiative)[1] in VR, using the real-world metaphor of islands (Figure 1). The whole architecture is represented as an island landscape on the sea. The single bundles are represented by islands. Their area is divided into regions corresponding to the packages contained in the bundle. The individual compilation units (classes and interfaces) are buildings, which are located in the corresponding regions.

To include the history of a software architecture, it is

[1] https://www.osgi.org/

978-1-7281-9914-6/20/$31.00 ©2020 IEEE
DOI 10.1109/VISSOFT51673.2020.00006

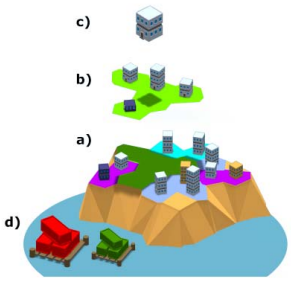


Fig. 1: IslandViz - visualizes the software architecture using the island metaphor

necessary to consider how new, deleted, and changing bundles, packages and compilation units affect the visualization. For IslandViz we identified three main aspects that need to be considered. First, the data must be made available. A database that contains not only the status at a certain point in time, but the entire history of a project. In a second step, the visualization must be able to adapt to constantly changing island layouts. In this context the preservation of the mental map [8] is a key requirement [9], [10]. Finally, the new extension needs to be evaluated using a user study.

In the remaining paper, we present our contributions on visualizing the evolution of OSGi-based software systems as follows:

1) We begin with our approach for the exploration of the evolution of software projects in VR (Sect. II).
2) Then, we present some details on the particular implementation (Sect. III).
3) We give a short overview of a focus group interview we conducted to identify requirements of software developers for a software visualization (Sect. IV).
4) Followed by the description of our pilot user study including the experimental design, the measurements, the procedure used and the results (Sect. V).
5) After that, we interpret the results (Sect. VI).
6) We conclude the paper with the major findings and indicate future work directions (Sect. VIII).

II. Visualization Approach

In the following section we describe our concept to visualize the evolution of software architecture using the island metaphor.

A. Island Positions

In the original IslandViz, island positions and distances between different islands implicitly visualize dependencies between bundles. In the evolution of a software project, bundles are created and deleted and dependencies change. As a result our history extension needs to handle changing dependencies and recalculate island positions accordingly. The difficulties lie in retaining a mental map of the visualization while islands change their positions. The visualization of dynamic graphs $G = G_0, G_1, ..., G_n$ [11], which are defined as a sequence of graphs, where each graph is a representation of the dynamic graph at one given time, faces the same problems. We looked at two approaches from graph theory for our concept.

1) Aggregated Graph Approach: Based on the work of [12] we first decided on creating an aggregated graph which takes all nodes and edges of all commits into account. The weight of nodes and edges is defined by how often they appear in our dynamic graph. Each node was assigned a position in the aggregated graph using a force-based approach. The idea behind this approach is that nodes repel each other while edges prevent them from drifting too far away. The formulas we used based on the work of [12] were:

$$\vec{F_a} = \frac{w_e * d^2}{l^2} * \vec{d} \quad (1)$$

$$l = \frac{\sqrt{w_u * w_v}}{w_e} \quad (2)$$

$$\vec{F_r} = \frac{\sqrt{w_u * w_v}}{d^2} * \vec{d} \quad (3)$$

w_e defines the weight of an edge between nodes u and v with their respective weights. l defines the optimal length of edge e.

To further improve the layout of a single graph in the graph sequence, we used a refinement based on the works of [13]. They defined a mental distance $\Delta(l1, l2)$ between two layouts l_1, l_2 of two graphs g_1, g_2 as the sum of distance between the nodes of both graphs.

$$\Delta(l_1, l_2) = \sum_{v \in V_1 \cap V_2} dist(l_1(v), l_2(v)) \quad (4)$$

After initialising the aggregated graph we applied a force-based layout algorithm on it. The new layout will be accepted if the mental distance between this layout and the other layouts of the sequence is smaller than a predefined tolerance value δ.

2) History-force Approach: Our second approach, which we ultimately decided to explore further, was to include an additional force which will be called history-force further on. Based on [14], we included a force connecting the same instance of a node in neighbouring graphs of the sequence. As a result, nodes will be attracted by their predecessors and descendants. For our implementation we used the following formula to calculate the history-force:

$$\vec{F_h} = \sum_{i=1}^{w} c_5 * c_1 * \frac{1}{2^{i-1}} * \vec{d_n} * \frac{d^2}{c_4} \quad (5)$$

$$\vec{d} = v_{t-i} - v_t \quad (6)$$

The distance vector $\vec{d}$ defines the distance between the currently looked at position of a node v_t and its $i-th$ predecessor. c_5 describes the relationship between the original force and the newly introduced history-force. Based on the recommendation of [14] to not only look at direct neighbors, to improve the positioning of nodes, w defines how many neighboring graphs in the sequence will be looked at. To reduce computation time, we decided to look at 4 predecessors but only one descendent,

since including multiple descendants would require multiple iterations to calculate the layout.

B. Island Layouts

1) Enhanced Hexagon Tiling Algorithm: The form of islands in IslandViz is defined by the *Enhanced Hexagon Tiling Algorithm* (EHTA) [15]. The EHTA assigns to each region a number of cells in a hexagonal grid. The number of cells which are assigned to one region is proportional to the number of compilation units in each package.

Starting at a random cell in a grid, the EHTA distributes each new cell to a region as follows:

- The next cell assigned to a region is a free cell, that is neighboring at least one already assigned cell.
- The probability of a free cell being selected next depends on the number of already assigned neighbors. The probability $P(n)$ that a cell with n assigned neighboring cells is selected is proportional to a score s_n of a cell.

$$P(n) \sim s_n = b^n \tag{7}$$

  The compactness of the resulting selection is defined by b.

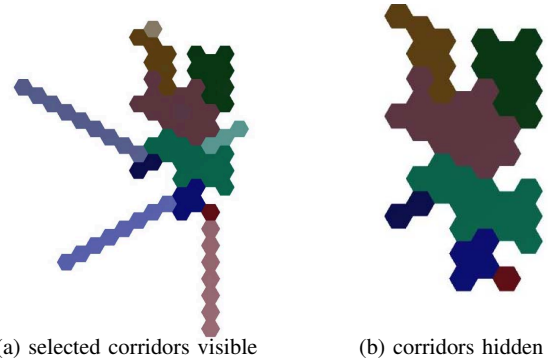
After the required number of cells of a region has been reached, the next region is added at the border of the already existing one. The algorithm iteratively determines the required cells of all regions.

Buildings representing the compilation units of the package have to be placed on the regions accordingly to the island metaphor. This means that the area of the region must be proportional to the number of buildings to provide sufficient space for them. In our approach the assignment is performed in a way that exactly one building can be placed on each cell of the grid which itself is based on the layout.

Since the number of compilation units within a package can change over the life cycle of the software, the number of buildings to be displayed within a region also changes, and so does the required space. Therefore, the concept for the island layout must ensure that there is sufficient space available within the region for the increasing number of displayed buildings. However, it is not possible to assign a sufficiently large area to each region from the beginning because the necessary information about how many cells are required is not available yet.

In order to allow each region of an island to expand as needed, the island layout is designed in a way that each region always has unoccupied border cells that can be occupied for expansion. This ensures that each region is provided with coastal access to grow.

For this purpose, we extend the EHTA by adding conditions and parameters for the random selection of new cells. The basic idea is to define absolute growth corridors for each newly created region. Cells that are located on a growth corridor can only be assigned to the respective region. For each newly created region, we define a growth corridor that does not overlap with another region or its growth corridor. This ensures an unlimited growth of the individual regions (Figure 2).



(a) selected corridors visible (b) corridors hidden

Fig. 2: Island built using growth corridors

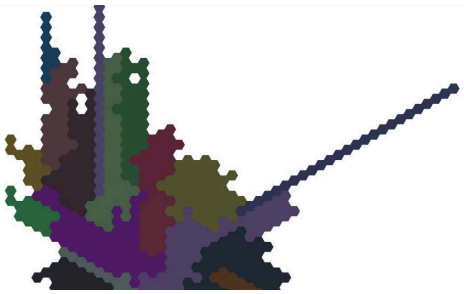


Fig. 3: Example for regions with a width of only one cell

By adding a new region, the starting cell is selected in a manner that its growth corridor can be created without overlapping with already occupied cells or other growth corridors. The Probability $P_{start-opt}(n)$ that a border cell is checked for being a possible starting cell is:

$$P_{start-opt}(n) = b^6 - b^n \tag{8}$$

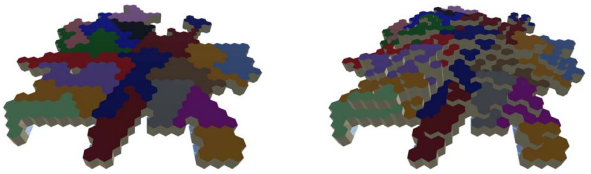
with n being the number of assigned neighbors and b being the same as in equation 7. In case a selected border cell is not suitable as a starting point for a region, additional cells are checked until a start cell is found.

As shown in Figure 3 unrealistic island layouts can arise when regions are created directly between two existing growth corridors. In most cases this results in a growth corridor being only one cell wide. Our solution is, to include the distance to the nearest growth corridors in our calculation. Our new formula is:

$$P_{start-opt}(n, step_r, step_l) = \begin{cases} 0.1 & \text{if } step_r = 0 \vee step_l = 0 \\ step_r * step_l * (b^6 - b^n) & \text{else} \end{cases} \tag{9}$$

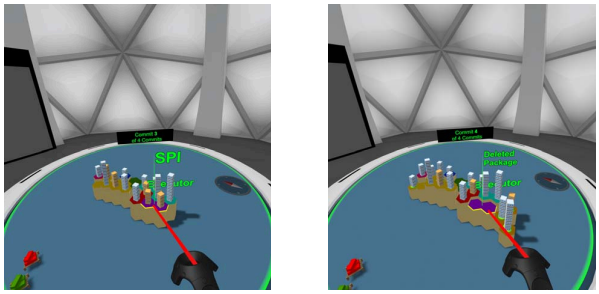
$step_r$ and $step_l$ are the number of cells to the nearest growth corridors to the right and to the left side of a potential starting cell. Is a cell located next to an existing corridor ($step_r = 0 \vee step_l = 0$), the probability will have a low constant value. This ensures that a cell can be selected as starting cell if only free cells are located next to a growth corridor.

2) Height Profile of Islands: In order to extend the visualization of the islands with an additional topology, the height of a single cell can represent the age of the corresponding



(a) same height for all cells (b) time-dependent height of each cell

Fig. 4: Height profile of islands



(a) Commit before the package is deleted (b) Commit where the package was deleted

Fig. 5: Effect on the island layout when deleting a package

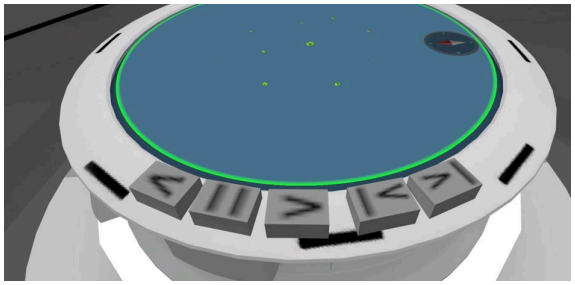


Fig. 6: Navigation interface to either play an animation, visualizing every commit sequentially (buttons on the left) or to navigate from one commit to another (buttons on the right)

Fig. 7: Notation Panel: when changing between commits, the user will receive information on who was the commiter and the beginning of the commit message, if available

compilation unit (Figure 5). For this purpose, the cell of a new class or interface is first displayed at low height above sea level. As time passes, the height of the cell increases.

3) Deletion of Components: After deleting compilation units within a package, no building will be displayed anymore. The hexagon cell reserved for the compilation unit still remains part of the region. Deleting a package means that all compilation units of the package are deleted. The area of the region is still visible, but the information about the package can no longer be displayed. Instead of the package name, the region is referred as ”Deleted Package” (Figure 5). The approach to not use the once assigned hexagon cells for new buildings after the compilation units have been deleted intends to support the preservation of the mental map. Especially when switching between two non-consecutive commits, it is possible to retrace where elements have been deleted.

C. User Interface

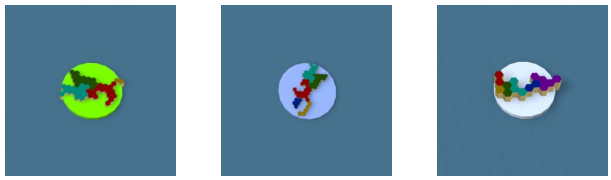
In order to let users navigate through the history of a software project, we enhanced the interface of the original IslandViz. We included a navigation interface (Figure 6), making it possible for users to automatically play all commits sequentially, to gain insight on how the project changed over the entirety of its existence. In addition we added the possibility to go through the commits step by step, giving the users the option to explore each commit before navigating to another.

Since users are free to move around the table which displays IslandViz, the navigation interface moves around the table as well, so it is always at the closest point to the user.

In addition, we added an information panel to support users orientation by displaying which commit is visualized at any given time. When changing between commits, we furthermore added a notification panel (Figure 7) that displays the author of a commit, as well as the beginning of the commit message, if available.

We furthermore added highlighting of changes (Figure 8), to enhance the user experience of our system. If the visualization view of IslandViz is on system level detail, new appearing islands are highlighted by an underlying green disc. If an island already exists but a change occurs on package level (e.g. creation/deletion of packages), resulting in a changed island layout, the island is highlighted by a blue disc. In case only the height of buildings on an island change, due to changes in a compilation unit, the island is highlighted using a white disc.

On bundle level detail, the same color scheme is used to highlight changes. New areas and buildings are highlighted with green discs beneath buildings, changes in a compilation unit are highlighted by blue discs beneath the buildings.



(a) new island (b) layout (c) buildings

Fig. 8: Highlighting of changes when (a) a new island is added (green), (b) the layout of an island is changed (blue) or (c) the height of buildings on an island are changed (white).

III. Implementation

In this section we describe the implementation of the concepts described above, Sect. II. We first describe our data mining approach to extract all relevant information from source code repositories and some details on the particular implementation.

A. Data mining

We analyze Java projects that are based on the OSGi framework (Open Services Gateway Initiative). This framework modularizes and manages software projects and their services. OSGi projects include bundles. Each bundle is a *JAR* archive with a *MANIFEST.MF* file, which describes different information such as dependencies and services. We analyze OSGi-based projects by extracting all relevant information from source code repositories. The information from all *Java* files, *MANIFEST.MF* files, and *XML* files are stored in an intermediate data model. For this purpose we developed a java-based application that provides all data needed for the visualization. The information about the architecture is extracted from a Git repository using an iterative process. In this case Git is the version control system that acts as our data source. A Git repository contains the complete historical information about a software project and consequently all visualization-relevant data.

To store the data we chose a graph database. In the context of this work, the information about relations between the different components is at least as important as the entities themselves, so we selected the Neo4j[2] graph database. Figure 9 shows a selected extract of the graph database. The nodes and relations represent the OSGi specific structure of a software project. In order to map the software history, we added an additional node of the type *CommitImpl*. Each node of this type represents a commit in the version control system. Linked to such a node is the state of the entire architecture at a given point in time. In order to bring all nodes of type *CommitImpl* into a chronological order, an additional relationship called *NEXT* is added between the preceding and subsequent commit (Figure 9). For all other types of nodes a *NEXT*-relationship is only added if the corresponding components stay the same from one commit to another. The result is a timeline for every component that is accessible in the graph database. For the required comparisons in this procedure we entirely use Cypher-queries.

[2]https://neo4j.com/

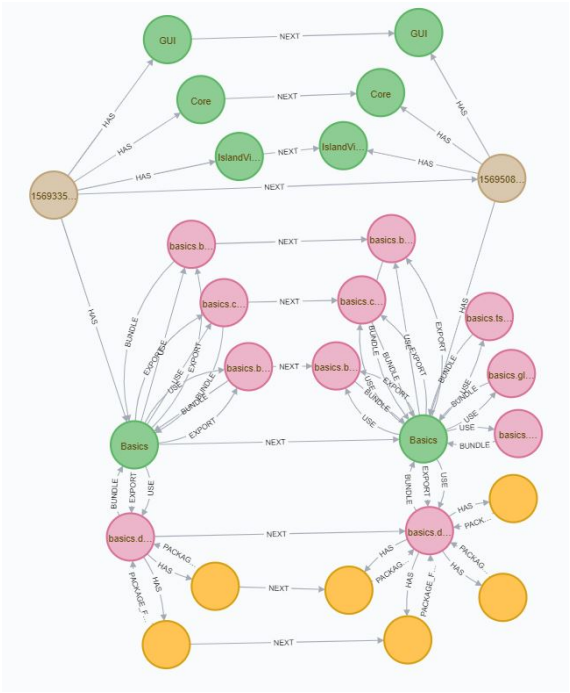


Fig. 9: Visualization of a selected extract of the graph database (brown node: *CommitImpl).*

B. Configuration

We developed our visualization with Unity 2019.3[3] using the SteamVR plugin in the version 1.8.21[4]. The targeted HMD is the HTC Vive Pro. The visualization was developed on a computer with an eight core 2.60 GHz CPU, an NVIDIA GTX 1080 GPU and 64.0 GByte RAM.

At the start of the application, steps are performed as the following:

Preparing the database: We first make sure that all information necessary for the visualization is actually available in the database (Sect. III-A).

Importing the data: All information about the architecture and history of the software project is extracted from the database into an internal data structure.

Island and graph layout: The visualization of the OSGi components is calculated according to the concepts described in (Sect. II). The layouts of the islands are initially calculated by the extended EHTA. Then the positions of the islands within the visualization are computed by the force-based approach with historical force. Because it's

[3]https://unity.com/de/releases/2019-3

[4]https://steamcommunity.com/games/250820/announcements/detail/1621788314781400373

important to maintain the mental map over the span of multiple application usages (that are separated in time), the calculated information for the visualization will be stored in the database. As long as the information for the visualization already exists, no recalculation needs to happen. In this case the layouts are only reconstructed from the data. If however new commits are added since the last usage, the layout is recalculated for these commits.

Creating GameObjects: In the last step of the loading process, all necessary Unity GameObjects are generated to represent the islands and their components.

IV. Focus Group

When we started developing the history extension for IslandViz, we conducted a focus group interview to identify which requirements users had for such a visualization. Since we aim to support software developers in their work on big software projects, these were our target group.

A. Participants

For our focus group we invited six software developers (2 female, 4 male) working at DLR, in the age range of 27 to 42 years. With regard to experience with large software projects, the group was heterogeneous, with some participants only recently starting to work with large software projects, while others had experience in this sector for up to 15 years. Nonetheless, all participants were experienced software developers. All participants were acquainted with IslandViz.

B. Procedure

The focus group had a duration of 2 hours and started with an introduction of the topic, explaining the reasoning why we wanted to conduct the focus group interview, followed by a short introduction of all participants. We split the discussion into three parts. During the first part, we were interested in problems developers face everyday in which information about the evolution of the software project was needed. Furthermore, we wanted to identify where and how the developers currently searched for this kind of information. Lastly we wanted to identify how detailed information needed to be, in order to support developers in their work.

In the second section of our discussion we focused on visualization. Which kind of visualization do developers use and why. We wanted to identify which requirements developers had for a visualization and in which aspects of their work a visualization would be welcome.

In the third section of the focus group we presented our visualization concept (Sect. II), looking for ideas to improve on it. Furthermore, we discussed the user interface and gave participants the possibility to voice their wishes and concerns.

C. Analysis

Analyzing our focus group showed that developers looked at the history of a software to solve errors, comprehend changes during development, and to plan for future changes, as well as for work on the documentation.

Especially when errors occurred in code which was previously working correctly, developers claimed to look at recent changes. They furthermore used the history of a project to get an overview of the current version, but also to find out when and why changes were made in packages they did not work on for a while, and who was working on the code. Moreover, the developers were interested if other developers changed existing code while programming, which changes they made and why. The visualization should therefore include author and commit message. In most cases, developers were only interested in recent changes of up to two weeks, so the visualization should prioritise this time frame and allow fast navigation through recent commits.

Our focus group interview showed that the most important source to look at the history of a software project is the version control system and the created issues. Commits can be assigned to specific issues, making them an important source to get an overview of the project and the work in progress. Next to commits, issues should therefore also be featured in a visualization of software history.

Regarding user interaction, developers wished to filter commits by branches and authors. In addition, the visualization should show new structures and major changes more prominently, simplifying navigation to these commits.

V. User Study

We argue that by creating adaptable island layouts and dynamically changing island positions we support users in retaining a mental map of the software architecture. As a result, our extension should support users in recognizing changes in the software architecture more efficiently and more correctly. We therefore decided to conduct an exploratory user study to confirm our expectations and to identify usability issues. To this purpose we compared our extension of IslandViz with the original implementation using a between subject study design for our pilot study.

A. Conditions

In total we tested two conceptually different visualization systems in our user study – the original IslandViz (Condition A) and our extension (Condition B).

Condition A consisted of the original implementation of IslandViz. Since the original IslandViz does not support jumping between different commits, we included a navigation interface to load a new unity scene showing the previous or following commit from inside IslandViz. However, when changing between different commits in this system the visualization is interrupted during the loading of a new unity scene during which the participants are shown a light grey surface. *Condition B* consisted of the basic implementation of our history extension for IslandViz as described in section II. Island positions and layouts adapt according to the changes from one commit to another. Additional highlighting of changes was left out in this system since we were more interested in finding out how efficient our support of a mental map was realized.

TABLE I: Task 1

Compare the systems between commit 2 and 3.	
Name the new bundles.	GUI Command, Data Model
Name the deleted bundles.	Scripting

TABLE II: Task 2

Compare the bundle Data Model between commit 3 and 4.	
Which packages were added?	TestUtils
Which packages were deleted?	none
Which files were added in existing packages?	DefaultTypedDatumConverter, DefaultTypedDatumFactory, DefaultTypedDatumSerializer
Which files were deleted in still existing packages?	TypedDatumServiceImpl

B. Procedure

Participants arrived for our study and first were asked to sign an informed consent form. Afterwards they were asked to answer a questionnaire recording their demographic information. Participants were given an introduction text explaining the visualization as well as the tasks they were expected to perform throughout the study. They then put on the head mounted display and were asked to familiarize themselves with the visualization. During this exploration phase, the study conductor again explained the important navigation and interaction techniques of the system. Furthermore, participants were allowed to ask questions. When the participants felt comfortable, the task was presented inside the visualization on the information panel and the study conductor read each task out aloud. While the task was not displayed in the information panel the whole time, participants had the possibility to display it at any time. After completing both tasks, participants answered the System Usability Score (SUS) [16]. Since we were interested in identifying aspects to improve the user experience of our extension, we asked our participants several open questions regarding their experience with the system, as well.

C. Tasks

We created two tasks which were given in both groups. With the first task (Table I) we wanted to find out how participants are able to track changes occurring on system level detail. The second task (Table II) was designed to determine how effective participants can identify changes on bundle level detail of the software system. A task in our study was started by the conductor of the experiment and ended when participants had stated all answers correctly or declared that they were finished. For each task we captured the duration time as well as the completeness of the given answer. With regard to measuring completeness we assigned one point for each expected answer (e.g. in Task 2.3 a total of 3 Points could be reached). In total, participants could achieve 9 points.

D. Dependent Variables and Hypotheses

In our study we measured task completion time and task correctness, as well as the usability of each system using the system usability score. We were interested if our extension resulted in more efficient task completion. Our first hypothesis therefore was:

H_1 The average task completion time using the history extension of IslandViz will be smaller than in the original version.

Furthermore, we expected participants to detect more changes using our extension since a users orientation and mental map is preserved. As a result, our second hypothesis was:

H_2 Using history extension of IslandViz will result in better results regarding task completeness.

In addition, we anticipated overall a better usability score with our history extension based on its support of a mental map of the software architecture as well as the reduced loading times. Hence, our third hypothesis was:

H_3 Based on the SUS, the history extension will have a higher usability rating than the original IslandViz.

E. Participants

12 students and employees of the University of Würzburg participated in our preliminary study (7 female, 5 male). Our participants were between 19 and 52 years old ($Mdn = 22.5$, $IQR = 10.0$). Participants mainly had a background in computer science (4 human computer interaction, 4 other computer science, 4 non computer science). Half of our participants claimed that they had advanced knowledge in programming, while half stated that they had only beginners knowledge. 11 participants stated that they had previous experience with VR, while one participant was new to VR.

F. Results

We used Shapiro-Wilk Tests [17] to analyze our data for normality. For each test we computed the effect size r and applied the thresholds introduced by Cohen [18] with the values 0.1 (small), 0.3 (medium), and 0.5 (large). Comparing the task completion times for both conditions in Figure 10 shows that participants were in general faster to complete both given tasks using the history extension of IslandViz. The mean task completion time for both tasks was normally distributed. We applied an independent samples t-test which showed, the mean task completion time was significantly higher in condition A ($M = 818s$, $\sigma = 152.6s$) than in condition B ($M = 508s$, $\sigma = 163,47s$) with $t = 3.396$, $p = 0.007$, $r = 1.960$(large effect). We therefore accept H_1.

Figure 11 shows, participants in general achieved more correct and complete answers using our history extension. In comparison the variance in group B was much smaller than in group A. The task completeness in task B was not normally distributed, we therefore applied a non-parametric test to analyze our data for significant differences. The overall task completeness showed no significant differences between

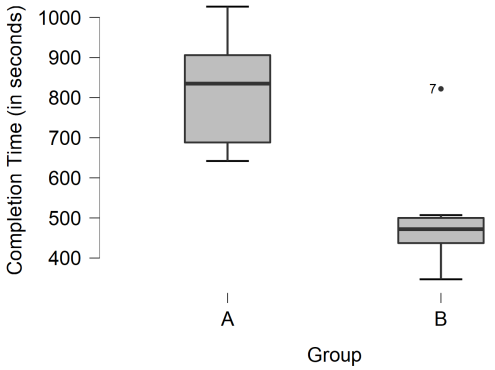


Fig. 10: Completion Times for Tasks in the original IslandViz (A) and the history extension (B)

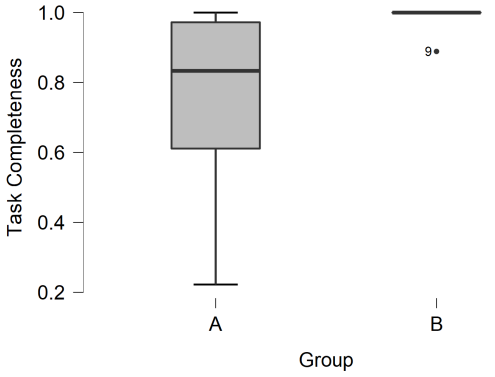


Fig. 11: Task Completeness for Tasks in the original IslandViz (A) and the history extension (B)

condition A ($Mdn = 7.5$ out of 9 points, $IQR = 3.25$) and condition B ($Mdn = 9$ out of 9 points, $IQR = 0.0$), with $p = 0.074$, $r = -0.583$ (strong effect). We therefore cannot accept H_2.

The results of the SUS (Figure 12) show that both conditions were rated similarly with condition B ($M = 73.75$, $\sigma = 18.29$) having a slightly higher score than condition A ($M = 92.92$,$\sigma = 10.66$). Both conditions are therefore graded acceptable by users.

In addition to the above analyzed criteria, we asked participants multiple open questions to find out how we can improve our system. The answers show that in general participants liked the idea of visualizing large software projects in VR using an island metaphor. When asked about what they liked about the systems when they were completing the above tasks, participants of group A commented more about the visualization in general. Participants of group B commented about the consistency in the visualization which allowed them to detect changes faster. When asked what they did not like, participants in group A reported difficulties in detecting changes due to the long loading times and the multitude of changes happening at once. Furthermore, they did not like

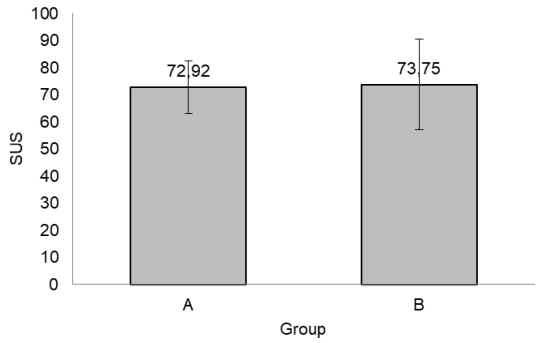


Fig. 12: Results of the SUS in the original IslandViz (A) and the history extension (B)

that islands changed their locations completely. Participants in group B in general did not like that there was no additional highlighting of changes. In addition they complained about the slightly moving islands when switching between commits, especially in the context of task 2 on bundle level detail.

VI. Discussion

With our user study we wanted to determine if our concepts introduced in Section II improved users ability to discover changes in a software project over time. We furthermore wanted to gain information on which features would enhance user experience. The study was only conducted as pilot study, with a small sample size to gain early insights in the results of our concept. In addition, we wanted to generate ideas to improve our interface. The results therefore have only a small validity and reliability. Nonetheless, the results indicate that our extension is advantageous when exploring the history of a software product which we want to further explore in the future.

While users rated both conditions similar using the SUS, the answers to our open questions show that participants of group B had less significant issues using our history extension then participants in group A.

With regard to completion times, completing tasks in our history extension was significantly faster than using the original IslandViz. This might be in part due to the fact that additional loading times prolonged the needed time in condition A; however, in combination with our observations and the answers to our open questions, we are confident that the improved layout consistency of our extension was an important factor as well.

Even though statistical testing did not provide significant results with regard to task completeness, participants in group B were able to achieve 98% of the given points. Participants of group A were in general able to identify the changes as well; nonetheless, the variance was much higher in this group, indicating that the system in condition B provided more support which led to these differences.

VII. Related Work

Lanza [19] presents an approach to visualize software history through changing metrics. For this purpose, he develops an "Evolution Matrix", in which a class is represented as a rectangle at a specific point in time of a software development process. To represent the temporal aspect, the rectangles are arranged in this evolution matrix. Each row of the matrix is assigned a class of the software system while the columns of the matrix represent the state of the class at different points in time. By looking at the matrix row-wise, the evolution of a class within a system can be recognized. By Looking at the matrix column-wise an overview of the system at any given point in time is possible.

Hassan et al. [20] follow a similar approach with their "Spectograph". Here, also artifacts of the software system are arranged line by line and points in time column by column, though in addition to this the considered metric is encoded by the colour of the matrix field. For this purpose, four colours are suggested that each correspond to a quarter of the value range of the metric.

Gall et al. [21] visualize the history of software architecture in three-dimensional space. To achieve this, the software architecture is represented at a specific point in time in a two-dimensional graph. By lining up these graphs in the third dimension, the temporal component is introduced.

Steinbrückner and Lewerentz [22] developed "EvoStreets", a visualization which is based on the city metaphor, but also considers the software history. In this work the package hierarchy of the software system is realized by streets. Buildings represent classes which are arranged along the streets. To change the city according to the software history, Steinbrück and Lew define rules. For example, one of the rules determines that new elements are added to the end of the street and another rule describes that the visualization of deleted elements are marked with a bright color.

VIII. Conclusion and future Work

In this paper we presented our approach to visualize the history of OSGi-based software projects through an island metaphor. The main focus was to preserve the mental map of the user while viewing the software history. In the context of this work, this term describes that the orientation of the user within the visualization should be preserved, although the visualization-components are required to adapt and change with the modifications in the software history. This was accomplished in different ways depending on the level of detail that needs to be displayed in the visualization. In the overview of the system the islands of the archipelago are positioned by an algorithm for dynamic graphs. This algorithm keeps the islands close to their previous positions if the archipelago grows. An adaptive layout makes it possible that a bundle can be displayed as an island and that additional buildings can be added to each region. This is realized by providing secure coastal access and growth corridors of the regions. For the implementation we have extended the Enhanced Hexagon Tiling Algorithm (EHTA).

In a pilot user study, we compared the extended island visualization with an application that also displays the software architecture using the island metaphor, but reloads the visualization between each commit. First results indicate that the extended island visualization supports the user in retaining a mental map of the software project. Nonetheless, we plan to confirm our results in a second, extended user study in which we will set the main focus on enhancing the user interface.

Our visualization is specifically designed with the goal to support users in their understanding of complex software architectures. Therefore, future work will foremost focus on enhancing our interface and interaction techniques to make our extension of IslandViz more accessible to users. To this purpose we will also take into consideration the results from our focus group interview and pilot user study. Other future work aims to support other component models than OSGi and other programming languages than Java. Beside visualizing the evolution of software projects, we plan to integrate further software metrics into the visualization.

Acknowledgment

This paper is based on a master thesis that originated from a cooperation with the University of Würzburg. We thank all colleagues who took part in creating this work.

References

[1] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, ser. EWSPT '96. Berlin, Heidelberg: Springer-Verlag, 1996, p. 108–124.

[2] ISO/IEC-14764, *Software Enineering - Software Life Cycle Processes - Maintenance*, ISO ISO/IEC 14 764:2006(E), Dec 2006.

[3] T. Panas, R. Berrigan, and J. Grundy, "A 3D metaphor for software production visualization," in *Proceedings of the Seventh International Conference on Information Visualization*, ser. IV '03. USA: IEEE Computer Society, 2003, p. 314.

[4] R. Wettel and M. Lanza, "Codecity: 3D visualization of large-scale software," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 921–922. [Online]. Available: https://doi.org/10.1145/1370175.1370188

[5] H. Graham, H. Y. Yang, and R. Berrigan, "A solar system metaphor for 3D visualisation of object oriented software metrics," in *Proceedings of the 2004 Australasian Symposium on Information Visualisation - Volume 35*, ser. APVis '04. AUS: Australian Computer Society, Inc., 2004, p. 53–59.

[6] M. Misiak, D. Seider, S. Zur, A. Fuhrmann, and A. Schreiber, "Immersive exploration of OSGi-based software systems in virtual reality," in *2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, March 2018, pp. 1–2.

[7] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, and M. Misiak, "Visualization of software architectures in virtual reality and augmented reality," in *2019 IEEE Aerospace Conference*, March 2019, pp. 1–12.

[8] P. Eades, W. Lai, K. Misue, and K. Sugiyama, "Preserving the mental map of a diagram," in *Proceedings of Compugraphics '91*, 1991, pp. 24–33.

[9] D. Beyer and A. E. Hassan, "Animated visualization of software history using evolution storyboards," in *Proceedings of the 13th Working Conference on Reverse Engineering*, ser. WCRE '06. USA: IEEE Computer Society, 2006, p. 199–210. [Online]. Available: https://doi.org/10.1109/WCRE.2006.14

[10] F. Steinbrückner and C. Lewerentz, "Understanding software evolution with software cities," vol. 12, pp. 200–216, 04 2013.

[11] S. Diehl, C. Görg, and A. Kerren, "Preserving the mental map using foresighted layout," in *Proceedings of the 3rd Joint Eurographics - IEEE TCVG Conference on Visualization*, ser. EGVISSYM'01. Goslar, DEU: Eurographics Association, 2001, p. 175–184.

[12] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis '03. New York, NY, USA: ACM, 2003, pp. 77–ff. [Online]. Available: http://doi.acm.org/10.1145/774833.774844

[13] S. Diehl and C. Görg, "Graphs, they are changing," in *Graph Drawing*, M. T. Goodrich and S. G. Kobourov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 23–31.

[14] A. Chapanond, M. S. Krishnamoorthy, G. M. Prabhu, and J. R. Punin, "Evolving graph representation and visualization," 2010.

[15] M. Yang and R. P. Biuk-Aghai, "Enhanced hexagon-tiling algorithm for map-like information visualisation," in *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, ser. VINCI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 137–142. [Online]. Available: https://doi.org/10.1145/2801040.2801056

[16] J. Brooke, "SUS: A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, 11 1995.

[17] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," vol. 52, no. 3, pp. 591–611, publisher: [Oxford University Press, Biometrika Trust]. [Online]. Available: https://www.jstor.org/stable/2333709

[18] J. Cohen, "A power primer," vol. 112, no. 1, pp. 155–159, place: US Publisher: American Psychological Association.

[19] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," *International Workshop on Principles of Software Evolution (IWPSE)*, 09 2001.

[20] A. E. Hassan, Jingwei Wu, and R. C. Holt, "Visualizing historical data using spectrographs," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, 2005, pp. 10 pp.–31.

[21] H. Gall, M. Jazayeri, and C. Riva, "Visualizing software release histories: the use of color and third dimension," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Aug 1999, pp. 99–108.

[22] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 193–202. [Online]. Available: https://doi.org/10.1145/1879211.1879239