Modeling, Visualizing, and Checking Software Architectures Collaboratively in Shared Virtual Worlds

Rainer Koschke[1], Marcel Steinbeck[1]

[1]University of Bremen, Bibliothekstraße 1, 28359 Bremen, Germany

Abstract

Software visualization is useful to highlight certain aspects of software in a way that is easy to grasp for humans. In this paper, we present our software visualization platform SEE which, among other use cases related to software development, assists developers and architects in identifying inconsistencies between the architecture and the implementation of a software—using the software reflexion model. SEE is based on the software-as-a-city metaphor and presents the generated software cities in virtual worlds that can be entered by multiple users from different locations (i.e., they do not have to be physically in the same place). Within these worlds, users can see each other as avatars and communicate via a built-in voice chat. A special feature of SEE is the ability for users to interact remotely with the cities in real-time and thus creates a basis for collaborative work that goes far beyond the classic means of distributed software development.

Keywords

reflexion analysis, software visualization, virtual and augmented reality, code cities, distributed development

1. Introduction

There has been a sustained trend towards distributed software development long before the current pandemic situation [1]. Distributed development is a consequence of budget and time limitations, lack of developers, need for specialized expertise, lack of space, and other factors. It takes place at large scale in terms of offshoring but also at small scale within an organization whose developers of the same team are not all in the same room. If developers of distributed teams need to work together, spatial gaps need to be bridged. Remote joint development is a particular challenge in situations where tight collaboration requires a high degree of communication. This is, for instance, the case when a team needs to recover and validate a software architecture from an existing system. For large systems, there is rarely a single person who knows all the details. Hence, multiple developers need to work together to reconstruct an accurate architectural description and to decide which implementation dependencies violate the architectural rules and how to handle these violations.

There are a few collaborative UML modeling tools, where different users can work on the same diagrams to model an architecture [2, 3]. Likewise, there are collaborative integrated development tools (IDE) such as Intelli/J IDEA with the feature *Code With Me*, which allow to edit and debug code collaboratively at the source-code level. Even though there are several tools to model and validate an architecture—even in the market place—we are not aware of any truly collaborative architecture modeling and checking tool that enables developers to model and validate architecture together at the same time, yet at different locations. Currently, architects and developers need to use screen sharing and video-conferencing systems to use such tools remotely. These, however, are very generic tools with no relation to the actual task at hand and, hence, are cumbersome to use.

Contributions In this paper, we present our software visualization tool *SEE* (for Software Engineering Experience). SEE is a multi-purpose visualization platform based on the *software-as-a-city* metaphor that allows users (software architects, developers, etc.) at different locations (i.e., they do not have to be physically in the same place) to work collaboratively on software architecture in shared virtual worlds. Within these worlds, all users have a visual representation—an avatar—and can thus *see* each other. In addition, users can talk to each other via an integrated voice chat. The virtual worlds created by SEE are dynamic so that users can highlight and change parts of the visualized software cities, which is visible to the other users in real-time. SEE can be used from different hardware devices: desktop computers, tablets, and virtual reality systems (VR). One of the primary use cases of SEE is the support of the software reflexion model [4], that is, the automatic identification of inconsistencies between a specified software architecture its implementation. This use case, and how it is implemented in SEE, will be the central subject of this paper.

Outline The remainder of this paper is structured as follows. Section 2 presents related research. Section 3 describes SEE and Section 4 how SEE can be used to support remote collaborative reflexion analysis. Section 5 concludes.

ECSA2021 Companion Volume

koschke@uni-bremen.de (R. Koschke);
marcel@informatik.uni-bremen.de (M. Steinbeck)

Figure 1: Virtual room with code cities for different use cases

2. Related Research

Our research focuses on the visualization of software and software architecture using the *software-as-a-city* metaphor with a special emphasis on collaborative visualizations where users can face each other in shared virtual worlds from different hardware devices. In this section, we will present related research of software visualization with regard to the *software-as-a-city* metaphor, software visualization in virtual and augmented reality environments, and collaborative software visualization.

2.1. Software as a City

In addition to the quantitative properties of software (e.g., lines of code), the hierarchy (e.g., namespaces) is often another aspect that needs to be visualized. An early approach that covers both aspects at the same time is the so called *Tree-map* [5] visualization. In Tree-maps the hierarchy of a software system is depicted with recursively nested rectangles where the area of the innermost rectangles is proportional to a certain metric. Initially, Tree-maps were designed as a two-dimensional visualization. However, quickly the idea came up to map an additional metric to the height of the rectangles, leading to three-dimensional blocks. Such three-dimensional Tree-maps create the impression of a typical North American city with buildings arranged in a grid. Due to this pictorial representation, three-dimensional Tree-maps are also known as *Code-Cities* [6]. Code-Cities were quickly adopted by the research community and are still very popular today [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26]. Besides the original Tree-map/Code-City algorithm there are also other layout methods grounded in the software-as-a-city metaphor, e.g., the *EvoStreets* visualization [27]. Using additional visual attributes, such as colors and textures mapped onto the surface of the three-dimensional blocks, further metrics can be expressed in Code-Cities [22]. Relations between entities (e.g., include dependencies, function calls, and so on) can be visualized with edges [28]. Hierarchical edge bundles, as proposed by Holten [29, 30], is a tried and tested means of diminishing the visual clutter that occurs when drawing lots of (overlapping) edges.

Code-Cities have long since arrived in practice and are now used in commercial products to visualize the change history and code quality of software. Examples include the software packages from *Hello2morrow* and *Serene* [31]. There is also a plug-in for the widespread software analysis platform *SonarQube*, *SoftVis3D*, which is based on Tree-maps and EvoStreets [32].

2.2. Software Visualization in AR/VR

As early as 2000 there were considerations to bring Code-Cities into virtual reality (VR) environments [7, 8]. Then as today it was hoped that the advantages of VR observed outside of computer science [33, 34, 35, 36] also apply in software visualization. Since then, Code-Cities were used in pseudo 3D (desktop computer monitors) and VR environments to visualize static [37, 38, 10, 39, 16, 40, 41, 42, 23] as well as dynamic [43, 15, 44, 45, 21] aspects of software. Studies have shown that head-mounted displays (HMDs) may have a positive effect on the orientation [33] and navigation speed [46] of users. That said, there are also studies where a positive effect could not be found [47] or where using HMDs had only little effect [48]. How these different results are to be assessed is a subject of further research.

2.3. Collaborative Software Visualization

Co-located collaborative software visualization where people interact with each other physically *in the same place* was studied by Isenberg et al. [49, 50] und Anslow et al. [51, 52]. The authors developed a multi-touch display mounted onto a table that can be used by several people (primarily pairs of users) at the same time. One of

the key findings of these studies was that working face-to-face around the table (i.e., sharing individual findings and communicating throughout) was a successful way for the pairs to solve complex problems, and that collaborative software visualization should support multiple users. An early attempt of implementing a distributed collaborative visualization (i.e., users do not have to be physically in the same place) in the context of software comprehension can be found in the work of Kot et al. [53]. Based on the *Quake 3* game engine, the authors developed a shared three-dimensional world where users can view, move, and arrange source-code files interactively. Further works in the area of collaborative software visualization are [38, 54, 55, 56, 3]—in a broader sense also [57]. Collaborative software visualization with focus on Code-Cities in shared virtual words was recently studied by Zirkelbach et al. [58] and Jung et al. [25]. In both studies users were represented as abstract avatars in the virtual world, allowing them to perceive each other and to support their verbal communication with a *simple form* of gesture. This aspect was in particular received positively in the study by Zirkelbach et al. [58], yet, the participants would have preferred a more realistic, human representation.

None of those studies aimed at architecture modeling and validation, which is the focus of our paper.

3. SEE Software Visualization

SEE (*Software Engineering Experience*) is a multi-purpose multi-user software visualization platform built upon the *Unity* game engine. The underlying data model visualized by SEE is a hierarchical graph with attributed nodes and edges—*hierarchical* means that nodes can be nested. Generally, SEE does not make any assumptions about what nodes, edges, and attributes represent and thus could be used to visualize anything that can be encoded as hierarchical graph. However, the main purpose of SEE lies on the visualization of aspects related to software (e.g., source-code files, packages, components, and so on) and software development (e.g., change history, quality metrics, and the like). Graphs can be imported from *GXL* files, a standard file format for exchanging arbitrary graphs that is used both in academia and industry [59]. The nodes and edges of an imported graph are visualized as three-dimensional blocks (leaf nodes), two-dimensional surfaces enclosing blocks (inner nodes), and splines connecting blocks or surfaces (edges) that can be hierarchically bundled. The visual components of the blocks (width, height, depth, and color), surfaces (shape and color), and splines (thickness, color gradient, and bundling strategy) can be freely configured based on the attributes attached to the corresponding nodes and edges. Using one of the built-in layout engines, blocks (and thus implicitly also surfaces and splines) can be arranged automatically—at the moment of writing, SEE supports *Circular Balloon*, *Circle Packing*, *Rectangle Packing*, *Tree-map*, and *EvoStreets* layouts. That said, it is also possible to change the position of blocks and splines, and insert and delete blocks and splines at any time (we will elaborate on that in Section 4.3).

The entirety of all visualized nodes and edges of a graph and their visual mappings form a Code-City. That is, there is a one-to-one relation between an imported graph and a Code-City. Code-Cities can be placed arbitrarily in the virtual world of SEE. For example, we created a virtual world—a Unity scene—that reminds one of a small library. Within this library are different tables on which a city could be placed as desired. Actually, SEE allows to visualize multiple cities at once. That is, with respect to our example, it is possible to import different graphs (or even the same graph multiple times), configure their visual mappings independently, and place each of the generated cities on its own table (cf. Figure 1).

4. Telecollaborative Reflexion

If one has an architecture modeling tool, one can share it via a screen sharing tool. Generally, those generic screen sharing tools provide very limited modes of interactions. Often only a single person has control over the screen and everyone is forced to view its content from the same perspective, that is, the shared content is identical for everyone. Users cannot take their own perspective, see different details, or otherwise interact with the shown content independently from others. Screen sharing embedded in video conference systems allows to also view the other present members of the team, but just at the edges of the shared content. They may be able to point to particular areas of interest via their mouse cursor but this pointing gesture is disconnected from the small video showing the participant triggering this gesture. There are first attempts to provide truly collaborative UML modeling tools, where different users can work on the same diagrams [2]. Likewise, there are collaborative integrated development tools (IDE) such as Intelli/J IDEA with the feature *Code With Me*. Yet, we are not aware of any collaborative architecture modeling and checking tool. Moreover, those collaborative coding or modeling tools do not allow to actually see the other team members. A lot of information between humans is exchanged by non-verbal communication, however. For instance, delayed movements or frowning may express hesitation or uncertainty.

Our goal is to enable developers in distributed teams to model and validate software architectures even when they are not at the same place. To do that we integrate different technologies ranging from ordinary desktop

computers with 2D display, keyboard, mouse, and camera over tablet computers with touchscreens to modern hardware for augmented (AR) and virtual reality (VR). The decisions for our kind of visualization and the provided interactions are founded on concepts of cognitive psychology including but not limited to laws of Gestalt, cognitive schemes, and mirror neurons. In the following, we will delve into those details.

4.1. Cognitive Foundations

Visualization and interaction should be as intuitive as possible. Intuition provides insights without inferences of the conscious mind by drawing on processes that are entrenched in human cognition [60]. These cognitive mechanisms should be taken into account in the design of efficient and effective visualization and interactions.

One such cognitive mechanism are *cognitive schemas* that describe associative structures in a human brain by which knowledge and experience is organized. For instance, if an object is observed, typical perceptions of how one can interact with this object are activated in the memory. These associations are not limited to simple relations; they can also include more complex behavior. All cognitive schemas have in common that they are triggered by a particular perception, for instance, an object, a situation, or a sensation. Dominic et al., for instance, have investigated how the activation of such cognitive schemas have effected the behavior of participants in a VR scene [61]. They found that emotional reactions could be triggered based on a suitable stimulus. Kot et al. observed in a visualization in VR how the participants grabbed objects—here representing files—to take those objects to other present members to show them to others Kot et al. [53]. The observed behavior has not been foreseen, let alone purposefully implemented by the authors. The behavior just arose from a cognitive schema for small physical objects that can be grabbed. That means for our context that we should present objects in a visualization in a way that triggers the wanted behavior. For instance, the mapping of implementation elements onto architectural components can be expressed by simply stacking one on the other. Metaphorically, the architecture is a city map and an architectural component a district within this city map. Implementation elements are physical objects, for instance, blocks that can be grabbed and put on the map. If an entity is to be re-mapped, it is just moved to the other place. The possible interaction is naturally suggested by a human's experience with physical objects.

Another relevant cognitive mechanism are the laws of Gestalt, which hold independently of cultural and even interpersonal differences [62]. The laws of Gestalt are a set of principles of human perception of viewn sceneries. For instance, the law of similarity predicts that physically similar items will be perceived as the same kind of object. That is, in our context the same kind of implementation entities, for instance, all methods, should be depicted alike. The inverse implication of that for us is that it would be rather misleading if architectural and implementation components would look alike. Their form and color and other visual attributes should be different to make clear that they are different concepts. The law of proximity states that objects close to each other appear to form a group. Automatic layout algorithms are generally agnostic to this law. They place their objects according to other criteria. For instance, tree-maps just try to save space and the objects put in close neighborhood have generally no semantic relation. In our visualization, humans will model the architecture and in doing so obey to this law naturally. Moreover, because we have multiple humans modeling the same architecture together, a process to reach a consensus is enforced because an object can be at only one place. Implementation entities are generally not modeled by humans, they are typically extracted from the source code through static or dynamic analysis. They have no initial physical location in 2D or 3D. The only partial ordering criteria we can extract from the code is hierarchical nesting (e.g., syntactic nesting, physical containment in the file system, or type hierarchies), linear order of declaration within a file, and dependencies among declarations, e.g., call relations. There are many layout algorithms that consider hierarchies. For instance, our current implementation offers tree-maps, EvoStreets, circle packing, rectangle packing, and balloon layouting. Moreover, force-directed layouts will group together elements according to their dependencies. While these algorithms may provide a first good placement of the elements extracted from the code, they have no deeper knowledge of the semantics beyond direct dependencies and hierarchies. For this reason, humans are free to move the objects in our visualization arbitrarily once they have been laid out automatically. In particular, when it comes to the mapping of implementation entities onto architectural components, they may stack those at arbitrary places within the boundaries of the visual element representing the architectural component they are mapped to. This way, the law of proximity will again hold.

Mirror neurons describe a property of certain neurons that were first detected in brains of monkeys but were later shown to exist in human brains, too. Certain neurons, for instance, those responsible to control a particular movement of the human body are not only activated when this behavior is to be executed but also when a human just observes this behavior for another person [63]. Recent research has shown that this property is not limited to motor neurons specialized on muscle control. Even neurons deriving bodily reactions from certain sensations were shown to have this property [64]. The implication in the context of collaborative visualization may be that it could be advantageous to show other

present participants in a visualization as avatars such that their movements can be observed by the beholders. Triggering their respective mirror neurons could not only help in learning interactions by example but also provide non-verbal clues on the sentiment of the acting person, e.g., hesitant movements indicating uncertainty or forceful movements indicating definiteness or even anger. Our conjecture of the relevance of showing the behavior of collaborating partners triggering mirror neurons is indicated by several studies on collaborative visualization [58, 49, 51, 52, 25]. In particular the participants of the study by Zirkelbach et al. have explicitly stated that they appreciated the presence of other members [58]. That study is interesting in two ways due to the way the presence of the other participants was visualized: The participants were drawn only by a virtual head-mounted display and the two handheld controllers in VR, not as human-like avatars. This way, others could observe where someone was looking or where someone was pointing to with the hand, which was appraised to be useful by the participants of the study. However, the participants stated that they would have preferred a more human-like representation. This outcome is consistent with studies in robotics which found that more human-like robots are generally more accepted [65] (up to the point where these machines become too similar to real humans [66] and start to frighten humans). To further explore the advantages of avatars and also to overcome the said disadvantages of current video conference and screen sharing systems discussed above, we show the participants present by way of human avatars.

4.2. Visualization and Interaction

After having introduced some of the foundations of human cognition influencing our design decision for the visualization and interaction, we will describe the latter two in greater detail.

Our early ideas with Code-Cities was to present them true to scale, that is, the proportions of the human body and the buildings representing software entities were as in real world [24, 67, 68, 69]. This causes problems of orientation due to occlusion and the limitations of the human short-term memory. While some of that might be mitigated by mini maps showing the current position of a person as in real world, it is still difficult to see other participants if they are in other parts of the city. Of course, one could blend them into the visible area of the beholder either at its edges as in video conference systems or as part of a hand-held device analogously to video calls with a smart phone. Yet, that basically means to virtualize video conference systems in a virtual world, and we wanted to overcome the problems of those. Mini maps and embedded video conference calls are just technical crutches to remedy bad design decisions. Moreover, cognitive schemas will not be triggered by this design. Humans would not be tempted to move around buildings that are magnitudes larger than themselves.

Our new design is more similar to approaches to software visualization in co-located environments. In previous studies on collaborative visualization, researchers have experimented with large multi-touch displays integrated in tables for the joint interaction with software visualizations at the same physical location [49, 50, 51, 52]. The human beholders group around the physical table (display) and can see both the visualization and the other participants at the same time. They can communicate with each other both verbally and non-verbally, which has been observed as a great advantage in these studies. Our approach can be viewed as a virtualization of this setting.

We provide a virtual room with several tables, each showing one particular software architecture and its implementation. Large organizations may have multiple applications and all of them could be made available in the same virtual room. This way participants could walk from table to table and work on different programs or just compare these. Each program is represented by one Code-City and generally there is one Code-City on each table. Participants are, however, able to take a Code-City to another table if they want to make comparisons between different programs. The Code-Cities are shown in miniature. They can be scaled and zoomed, however. Unlike the physical monitors for co-located environments, there are no limits for scaling enforced; participants will stop scaling by themselves at the point when they think it becomes useless. And also unlike the physical displays in co-located environments, our visualization has three dimensions. In particular, for implementation components, height may have an important meaning, for instance, the size of a class. If they are embedded in architectural components, it becomes immediately visible which architectural components tend to have god classes.

Another advantage over physical multi-touch tables is that we can individualize what can be seen for each beholder. We take great care that the virtual room is consistent among all participants; for instance, if one participant grabs an object, this object must move in all representations of the virtual room for all participants. Thus, our visualization is essentially a distributed real-time application. We even have a global undo/redo history identifying and prohibiting conflicting actions, e.g., one participant renames an element and the other one removes the same element. Nonetheless, there are aspects for which it makes sense to drawn them specifically to one beholder. For instance, on physical multi-touch tables, labels necessarily have exactly one orientation. A person on the opposite side will have difficulties to read them. Our virtual labels always face the beholder. Moreover, participants can query additional details on demand,

for instance, the source code of an implementation entity presented in a code viewer or additional metrics shown in a scatter plot. These additional views can be shared or not, depending upon whether the beholder has only a personal interest or whether she or he wants to talk about it. If they were always shown as it would be the case for physical multi-touch tables, they could distract others.

The participants are visible as human avatars and can communicate via voice to each other. The avatar's lips are synchronized with the spoken word so that it can be seen who is talking. To synchronize the movement of a human in the real world with his or her avatar, we leverage the position data of the head-mounted display and hand-held controllers in VR. In case of an ordinary desktop environment, we can derive the viewing angle of the avatar's head by the viewpoint (in game-engine lingo, the game camera angle) of his human counterpart. We currently do not have sensors for the human's hands in desktop environments. We plan to derive this information from 3D depth cameras or even ordinary cameras with suitable image-recognition software or physical trackers such as HTC's Vive trackers. Neither do we have a way to capture, transfer, and present mimics yet. Again, we will attempt to capture this data through cameras and ideally present them as real-time videos on the avatar's face. There are already commercial applications[1] for animating avatars according to the mimics of a human face showing that this is doable.

4.3. Virtual Reflexion Analysis

We are using the reflexion analysis [4, 70, 71] to reconstruct and validate the software architecture. This section describes in more detail how each step is implemented in terms of the design of the visualization and interaction.

Architecture modeling The first step of the reflexion analysis is to create a model of the architecture. Conceptually, the model forms a graph where nodes represent architectural components and directed edges specify expected dependencies between the connected components. Nodes can be nested in other nodes expressing hierarchical systems [70]. Our users can create such models on various devices, namely, ordinary desktop computers with mouse interactions, tablets with a pen recognizing shapes and edge-drawing actions (a user can virtually draw an architecture with the pen), and with physical actions with the handheld controllers in VR environments. We plan to support MicroSoft's HoloLens for AR, too. For desktops, we also experimented with a hand-tracking device named *Leap Motion* enabling a user to create nodes and edges with hand gestures. The problem with this approach is that the device has a limited range of visibility forcing a user's hands to be held out uncomfortably, gesture detection can be difficult if fingers occlude each other, and the precision was not enough for fine interactions through direct manipulation with selected objects.

The architecture can be modeled on the plane of a table around which all participants group. The creation of objects is instantaneous on all connected computers so that all participants always have the same view. If hand-tracking data is available, all participants can observe who is creating the new node or edge by visually following the hand. New and deleted nodes and edges are animated so that changes are highlighted to everyone present. It is important that everyone can see who initiated a change even before the change is actually finalized so that they can possibly intervene and that all recent changes are obvious. To make sure, the architecture is syntactically consistent (e.g., there are no dangling edges), conflicting changes are detected and refused.

Users want to name the nodes they created, which can be a challenge in VR. We offer a virtual keyboard, but the haptic feedback is of course missing. For this reason, we allow a user to dictate a name for a new node through voice recognition, which works surprisingly well when the name is not cryptic—and maybe it is better to avoid cryptic names anyhow. At least, the recognized name may a good starting pointing that can then be corrected by way of virtual keyboard keeping the necessary virtual key strokes at a minimum. Similarly to Seipel et. [72], we also offer a conversational interface to initiate other actions (e.g., for showing the code of a component) to free the user from the need to use a keyboard.

Mapping the implementation onto the architecture Once the architecture model is created to the point that one can move on to relate the implementation components to the architecture components they implement, users can drag and drop implementation components onto architecture components. This kind of mapping is expressed through nesting, that is, the objects representing an implementation entity are stacked on the area of an object representing an architectural component. This interaction leverages the conceptual schemas (small objects can be grabbed and moved) and the laws of Gestalt (an implementation entity is enclosed by an architecture component). It also leverages the mirror neurons as the physical action of the movement can be observed by the other participants.

Initially, the implementation is drawn as a Code-City next to the space where the architecture model is created by the user. Because the Code-City's elements are extracted by a static analysis, an automated layout will first decide how to place them. The user has the choice among various layouts we offer. The implementation nodes can afterwards be moved around within the limits of their

[1] https://facewaretech.com

containing node. The user can select the code metrics determining the width, height, depth, and color range of the nodes. All nodes of the same type (e.g., classes) have the same shape, which can be selected by the user, too. The type of the edges is depicted by color. Their direction is shown as a color gradient of the chosen color. Many other applications use arrow heads instead, but they may overlap for nodes with many connecting edges. Edges are laid out through hierarchical bundling [29], which helps to reduce visual clutter when there are many dependencies. Incoming and outgoing direct and transitive edges can be hidden or highlighted on demand. The source code leading to a particular node or edge can be opened in an in-game window with syntax highlighting.

To map an implementation entity onto the architecture, the user just grabs the object and drags it to the architecture component. This constitutes an explicit mapping. All descendants of the moved object in the node hierarchy are moved along with it—unless they have been mapped before. Those are implicitly mapped. If a user wants to map the implicitly mapped entities to somewhere else, he or she just moves its node in the architecture to another target. Again, syntactic checks are in place to make sure that an implementation node cannot be moved to another implementation node (unless that one is its original parent), because that would create a node hierarchy that is inconsistent to the code.

Nodes that were mapped explicitly or implicitly are marked visually as such. The mapping creates a logical copy of a node when it was moved into an architecture node. Its original representation in the separate Code-City for the implementation becomes transparent to make clear that it has already been mapped. If either of the two nodes is selected, its counterpart is selected, too, to make their connection clear. Preserving the original node in the implementation representation helps to study its relation to other nodes in the context of the original implementation, which may be a useful information for the decision where to map its neighbors. It also helps to assess the progress of the mapping process.

Reflexion analysis As soon as two ends of an implementation dependency (source and target nodes of the corresponding edge) are mapped (implicitly or explicitly), the automated reflexion analysis can determine whether the implementation dependency is *allowed* (i.e., covered by a corresponding architecture dependency) or represents a *divergence* (i.e., there is no such corresponding architecture dependency allowing it). Likewise, initially when nothing has been mapped, all architecture dependencies are so called *absences*, that is, there is no implementation dependency confirming them. Whenever nodes are mapped, these could turn into so called *convergences*, that is, there is actually an implementation dependency confirming them. We are using our incremental reflexion analysis [71] to compute the effect of each mapping decision thereby keeping the effort of the recalculation to a minimum. This on-the-fly computation also supports what-if scenarios, where a user can drag an implementation node over different architecture nodes to see the possible effect of a mapping immediately. All edges effected by a new mapping are animated so that the mapping effects can be observed among the many other edges present in the scene.

Edges in both the implementation and architecture are typed. Typed dependencies in an architecture make sense, for instance, to allow calls between components but not accesses to attributes. As mentioned above, colors are used for the type of edges. As a consequence, edge color cannot be used to distinguish between implementation and architecture dependencies. To show this distinction, architecture dependencies are drawn noticably thicker than implementation edges. For realistic systems, there are many more implementation than architecture edges and the focus in this visualization is the architecture, hence, it makes sense to show the architecture dependencies more prominently.

The remaining question is now: how to show whether an edge is allowed, divergent, absent, or convergent when color is no option because it is already used for edge types? Animation is already used for changed edges and animation in generally should be kept to a minimum; otherwise it may become annoying. We see no need to highlight allowed implementation edges and convergent architecture edges, because everything is in order with these. A user wants to see primarily where implementation and architecture differ, that is, divergent and absent edges should be easy to spot. Some tools decorate edges with a symbol to mark them as divergent or absent, but we find such decorations be difficult to relate to a particular edge when there are many edges, in particular, if edge bundling is applied. For this reason, we are using a radiance effect and dashed lines for absences and divergences.

Added value of architecture Architecture conformance checking is an important measure to ensure architecture and implementation are in sync, but there is potential for more added value. Architecture is a suitable abstraction to discuss other aspects of the implementation within distributed teams. Because the implementation is visually embedded in the architecture in our visualization it is easy to relate details of the implementation to the architecture. For instance, we support dynamic analysis where a user can trace the control flow by way of animated edges for dynamic calls, which raises the level of abstraction in the context of debugging. This way, the static architecture gets also a dynamic view. Similarly, we visualize performance data by way of spheres above methods whose radius represents the CPU time spent

within those and the number of calls by way of a color gradient, which allows to relate performance bottlenecks to the architecture. Test coverage metrics can be visualized through coloring such that the untested parts of the architecture can be spotted easily. Also, we show the change history along with the trends in code erosion as a kind of movie that allows how the system evolved both in terms of changes and quality. The implementation embedded in the architecture drawn as a Code-City is a uniform representation in all these development practices and may provide insights that can be discussed in a distributed team.

5. Conclusions

In this paper, we have described our software visualization platform SEE and how it can be used to support the reflexion analysis collaboratively for distributed teams. We explained the most important design decisions for the visualization and interaction based on current knowledge of cognitive psychology. It is still work in progress. As a next step, we plan to evaluate our design decisions empirically.

References

[1] C. Ebert, M. Kuhrmann, R. Prikladnicki, Global software engineering: evolution and trends, in: International Conference on Global Software Engineering, 2016, pp. 144–153.

[2] M. Magin, S. Kopf, A Collaborative Multi-Touch UML Design Tool, Technical Report TR-2013-001, University of Mannheim, Germany, 2013.

[3] M. Ferenc, I. Polasek, J. Vincur, Collaborative modeling and visualization of software systems using multidimensional UML, in: IEEE Working Conference on Software Visualization, 2017, pp. 99–103.

[4] G. C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: Bridging the gap between source and high-level models, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, 1995, pp. 18–28.

[5] B. Johnson, B. Shneiderman, Tree-maps: A space-filling approach to the visualization of hierarchical information structures, in: Proceedings of the Conference on Visualization, IEEE Computer Society Press, 1991, pp. 284–291.

[6] K. Andrews, J. Wolte, M. Pichler, Information pyramids: A new approach to visualising large hierarchies, in: IEEE Conference on Visualization, 1997, pp. 49–52.

[7] C. Knight, M. Munro, Virtual but visible software, in: International Conference on Information Visualization, IEEE, 2000, pp. 198–205.

[8] S. M. Charters, C. Knight, N. Thomas, M. Munro, Visualisation for informed decision making; from code to components, in: International Conference on Software Engineering and Knowledge Engineering, 2002, pp. 765–772.

[9] M. Balzer, A. Noack, O. Deussen, C. Lewerentz, Software landscapes: Visualizing the structure of large software systems, in: IEEE TCVG Symposium on Visualization, 2004, pp. 261–266.

[10] T. Panas, R. Berrigan, J. Grundy, A 3d metaphor for software production visualization, in: International Conference on Information Visualization, IEEE, 2003, pp. 314–319.

[11] A. Marcus, L. Feng, J. I. Maletic, 3D representations for software visualization, in: ACM International Symposium on Software Visualization, 2003, pp. 27–36.

[12] R. Wettel, M. Lanza, Visualizing software systems as cities, in: IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007, pp. 92–99.

[13] R. Wettel, M. Lanza, Codecity: 3d visualization of large-scale software, in: Companion of the 30th International Conference on Software Engineering, ACM, 2008, pp. 921–922.

[14] R. Wettel, M. Lanza, Visual exploration of large-scale system evolution, in: IEEE Working Conference on Reverse Engineering, 2008, pp. 219–228.

[15] F. Fittkau, S. Roth, W. Hasselbring, ExplorViz: visual runtime behavior analysis of enterprise application landscapes, in: European Conference on Information Systems, 2015, pp. 1–13.

[16] F. Fittkau, A. Krause, W. Hasselbring, Exploring software cities in virtual reality, in: IEEE Working Conference on Software Visualization, 2015, pp. 130–134.

[17] G. o. Balogh, A. Szabolics, A. Beszédes, Codemetropolis: Eclipse over the city of source code, in: IEEE International Working Conference on Source Code Analysis and Manipulation, 2015, pp. 271–276.

[18] L. Merino, M. Ghafari, C. Anslow, O. Nierstrasz, Cityvr: Gameful software visualization, in: IEEE International Conference on Software Maintenance and Evolution (TD Track), 2017, pp. 633–637.

[19] L. Merino, A. Bergel, O. Nierstrasz, Overcoming issues of 3d software visualization through immersive augmented reality, in: IEEE Working Conference on Software Visualization, 2018, pp. 54–64.

[20] W. Scheibel, C. Weyand, J. Döllner, Evocells - A treemap layout algorithm for evolving tree data, in: International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, 2018, pp. 273–280.

[21] L. Merino, M. Hess, A. Bergel, O. Nierstrasz,

D. Weiskopf, Perfvis: Pervasive visualization in immersive augmented reality for performance awareness, in: ACM/SPEC International Conference on Performance Engineering, 2019, pp. 13–16.

[22] D. Limberger, W. Scheibel, J. Döllner, M. Trapp, Advanced visual metaphors and techniques for software maps, in: International Symposium on Visual Information Communication and Interaction, 2019, pp. 1–8.

[23] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, M. Misiak, Visualization of software architectures in virtual reality and augmented reality, IEEE Aerospace Conference (2019) 1–12.

[24] M. Steinbeck, R. Koschke, M.-O. Rüdel, How EvoStreets are observed in three-dimensional and virtual reality environments, in: IEEE International Conference on Software Analysis, Evolution and Reengineering, 2020, pp. 332–343.

[25] F. Jung, V. Dashuber, M. Philippsen, Towards collaborative and dynamic software visualization in vr, in: Proceedings of the International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP, INSTICC, SciTePress, 2020, pp. 149–156.

[26] V. Dashuber, M. Philippsen, J. Weigend, A layered software city for dependency visualization, in: International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, volume 3, SciTePress, 2021, pp. 15–26.

[27] F. Steinbrückner, C. Lewerentz, Representing development history in software cities, in: ACM International Symposium on Software Visualization, ACM, 2010, pp. 193–202.

[28] R. Koschke, Software visualization in software maintenance, reverse engineering, and reengineering: A research survey, Journal on Software Maintenance and Evolution 15 (2003) 87–109.

[29] D. H. R. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data, IEEE Transactions on Visualization and Computer Graphics 12 (2006) 741–748.

[30] D. H. R. Holten, Visualization of graphs and trees for software analysis, Ph.D. thesis, Technical University of Delft, 2009.

[31] J. Bohnet, Visualization of Execution Traces and its Application to Software Maintenance, Dissertation, Hasso-Plattner-Institut, Universität Potsdam, 2010.

[32] SoftVis3D, Softvis3d website, https://softvis3d.com, 2021. Online; accessed 30-June-2021.

[33] S. S. Chance, F. Gaunet, A. C. Beall, J. M. Loomis, Locomotion mode affects the updating of objects encountered during travel: The contribution of vestibular and proprioceptive inputs to path integration, Presence: Teleoper. Virtual Environ. 7 (1998) 168–178.

[34] D. A. Bowman, E. T. Davis, L. F. Hodges, A. N. Badre, Maintaining spatial orientation during travel in an immersive virtual environment, Presence: Teleoper. Virtual Environ. 8 (1999) 618–631.

[35] B. E. Riecke, D. W. Cunningham, H. H. Bülthoff, Spatial updating in virtual reality: the sufficiency of visual information, Psychological Research 71 (2007) 298–313.

[36] J. W. Regian, W. L. Shebilske, J. M. Monk, Virtual reality: An instructional medium for visual-spatial tasks, Journal of Communication 42 (1992) 136–149.

[37] N. Capece, U. Erra, S. Romano, G. Scanniello, Visualising a software system as a city through virtual reality, in: L. T. De Paolis, P. Bourdot, A. Mongelli (Eds.), Augmented Reality, Virtual Reality, and Computer Graphics, Springer International Publishing, Cham, 2017, pp. 319–327.

[38] J. I. Maletic, J. Leigh, A. Marcus, G. Dunlap, Visualizing object-oriented software in virtual reality, in: International Workshop on Program Comprehension, 2001, pp. 26–35.

[39] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, R. Vuduc, Communicating software architecture using a unified single-view visualization, in: IEEE International Conference on Engineering Complex Computer Systems, IEEE, 2007, pp. 217–228.

[40] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, J. Laviola, Code park: A new 3d code visualization tool, in: IEEE Working Conference on Software Visualization, IEEE, 2017, pp. 43–53.

[41] L. Merino, J. Fuchs, M. Blumenschein, C. Anslow, M. Ghafari, O. Nierstrasz, M. Behrisch, D. A. Keim, On the impact of the medium in the effectiveness of 3d software visualizations, in: IEEE Working Conference on Software Visualization, IEEE, 2017, pp. 11–21.

[42] A. Schreiber, M. Brüggemann, Interactive visualization of software components with virtual reality headsets, in: IEEE Working Conference on Software Visualization, IEEE, 2017, pp. 119–123.

[43] J. Waller, C. Wulf, F. Fittkau, P. Döhring, W. Hasselbring, Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency, in: IEEE Working Conference on Software Visualization, 2013, pp. 1–4.

[44] K. Ogami, R. G. Kula, H. Hata, T. Ishio, K. Matsumoto, Using high-rising cities to visualize performance in real-time, in: Software Visualization (VISSOFT), 2017 IEEE Working Conference on, IEEE, 2017, pp. 33–42.

[45] F. Fernandes, C. S. Rodrigues, C. Werner, Dynamic analysis of software systems through virtual reality, in: Symposium on Virtual and Augmented Reality, 2017, pp. 331–340. In Spanish.

[46] R. A. Ruddle, S. J. Payne, D. M. Jones, Navigating

large-scale virtual environments: what differences occur between helmet-mounted and desk-top displays?, Presence: Teleoperators & Virtual Environments 8 (1999) 157–168.

[47] B. Sousa Santos, P. Dias, A. Pimentel, J.-W. Baggerman, C. Ferreira, S. Silva, J. Madeira, Head-mounted display versus desktop for 3d navigation in virtual reality: A user study, Multimedia Tools and Applications 41 (2009) 161–181.

[48] R. A. Ruddle, P. Péruch, Effects of proprioceptive feedback and environmental characteristics on spatial learning in virtual environments, International Journal of Human-Computer Studies 60 (2004) 299–326.

[49] P. Isenberg, D. Fisher, M. R. Morris, K. Inkpen Quinn, M. Czerwinski, An exploratory study of co-located collaborative visual analytics around a tabletop display, IEEE Symposium on Visual Analytics Science and Technology (2010) 179–186.

[50] P. Isenberg, D. Fisher, S. A. Paul, M. R. Morris, K. Inkpen, M. Czerwinski, Co-located collaborative visual analytics around a tabletop display, IEEE Transactions on Visualization and Computer Graphics 18 (2012) 689–702.

[51] C. Anslow, S. Marshall, J. Noble, R. Biddle, Sourcevis: Collaborative software visualization for co-located environments, in: IEEE Working Conference on Software Visualization, 2013, pp. 1–10.

[52] C. Anslow, Reflections on collaborative software visualization in co-located environments, in: IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 645–650.

[53] B. Kot, B. Wuensche, J. Grundy, J. Hosking, Information visualisation utilising 3d computer game engines case study: A source code comprehension tool, in: ACM SIGCHI New Zealand Chapter's International Conference on Computer-Human Interaction: Making CHI Natural, 2005, pp. 53–60.

[54] M. D'Ambros, M. Lanza, A flexible framework to support collaborative software evolution analysis, in: European Conference on Software Maintenance and Reengineering, 2008, pp. 3–12.

[55] M. D'Ambros, M. Lanza, Distributed and collaborative software evolution analysis with Churrasco, Science of Computer Programming 75 (2010) 276–287.

[56] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, R. Vuduc, Communicating software architecture using a unified single-view visualization, in: IEEE International Conference on Engineering Complex Computer Systems, 2007, pp. 217–228.

[57] E. Stroulia, I. Matichuk, F. Rocha, K. Bauer, Interactive exploration of collaborative software-development data, in: IEEE International Conference on Software Maintenance, 2013, pp. 504–507.

[58] C. Zirkelbach, A. Krause, W. Hasselbring, Hands-On: Experiencing Software Architecture in Virtual Reality, Research Report 1809, Christian-Albrechts-Universität zu Kiel, 2019.

[59] R. Holt, A. Winter, A. Schürr, GXL: toward a standard exchange format, in: IEEE Working Conference on Reverse Engineering, 2000, pp. 162–171.

[60] C. G. Jung, Gesammelte Werke, Band 6: Psychologische Typen, Walter Verlag, 1995, p. 474 f.

[61] J. Dominic, B. Tubre, J. Houser, C. Ritter, D. Kunkel, P. Rodeghero, Program comprehension in virtual reality, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 391–395.

[62] W. MacNamara, Evaluating the effectiveness of the gestalt principles of perceptual observation for virtual reality user interface design (2017).

[63] M. Fabbri-Destro, G. Rizzolatti, Mirror neurons and mirror systems in monkeys and humans, Physiology 23 (2008) 171–179.

[64] F. De Vignemont, T. Singer, The empathic brain: how, when and why?, Trends in cognitive sciences 10 (2006) 435–441.

[65] A. Prakash, W. A. Rogers, Why some humanoid faces are perceived more positively than others: Effects of human-likeness and task, International Journal of Social Robotics 7 (2015) 309–331.

[66] M. Mori, K. F. MacDorman, N. Kageki, The uncanny valley [from the field], IEEE Robotics & Automation Magazine 19 (2012) 98–100.

[67] R. Koschke, M. Steinbeck, Clustering paths with dynamic time warping, in: IEEE Working Conference on Software Visualization, 2020, pp. 89–99.

[68] M. Steinbeck, R. Koschke, M.-O. Rüdel, Comparing the EvoStreet visualization technique in two- and three-dimensional environments—a controlled experiment, in: International Conference on Program Comprehension, 2019, pp. 231–242.

[69] M. Rüdel, J. Ganser, R. Koschke, A controlled experiment on spatial orientation in VR-based software cities, in: IEEE Working Conference on Software Visualization, 2018, pp. 21–31.

[70] R. Koschke, D. Simon, Hierarchical reflexion models, in: IEEE Working Conference on Reverse Engineering, 2003, pp. 36–45.

[71] R. Koschke, Incremental reflexion analysis, Journal on Software Maintenance and Evolution 25 (2013) 601–637.

[72] P. Seipel, A. Stock, S. Santhanam, A. Baranowski, N. Hochgeschwender, A. Schreiber, Adopting conversational interfaces for exploring OSGi-based software architectures in augmented reality, IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE) (2019) 20–21.