Live Software Development Environment Using Virtual Reality: A Prototype and Experiment

Diogo Amaral[1], Gil Domingues[1], João Pedro Dias[1,2(✉)], Hugo Sereno Ferreira[1,2], Ademar Aguiar[1,2], Rui Nóbrega[1,2], and Filipe Figueiredo Correia[1,2]

[1] Faculty of Engineering, University of Porto, Porto, Portugal
[2] INESC TEC, Porto, Portugal
{diogo.amaral,gil.domingues,jpmdias,hugosf,aaguiar,ruinobrega,filipe.correia}@fe.up.pt

Abstract. Successful software systems tend to grow considerably, ending up suffering from essential complexity, and very hard to understand as a whole. Software visualization techniques have been explored as one approach to ease software understanding. This work presents a novel approach and environment for software development that explores the use of *liveness* and virtual reality (VR) as a way to shorten the feedback loop between developers and their software systems in an interactive and immersive way. As a proof-of-concept, the authors developed a prototype that uses a visual city metaphor and allows developers to visit and dive into the system, in a live way. To assess the usability and viability of the approach, the authors carried on experiments to evaluate the effectiveness of the approach, and how to best support a *live* approach for software development.

Keywords: Software engineering · Virtual reality · Live Software Development · Live programming · Software visualization

1 Introduction

Much of the software created today is built incrementally from an initial prototype that evolves gradually through the addition of new features. Along this process, the dimension of the system increases, and the productivity is hampered by the comprehension tasks.

Software systems can achieve very high complexity, to a great extent due to their size, which can reach millions of lines of code [2]. Software engineers, when adding new functionality or merely performing maintenance tasks, should first understand the system [23,32], which can be challenging due to scalability and complexity [7,19,33].

We argue that this difficulty can be reduced in many cases by applying the idea of *liveness*, i.e., the ability to modify a system while it is running [30],

E. Damiani et al. (Eds.): ENASE 2019, CCIS 1172, pp. 83–107, 2020.
https://doi.org/10.1007/978-3-030-40223-5_5

allowing the developer to receive immediate *feedback* of changes made, with the system continually rerunning while being edited.

Additionally, visualizing and interacting with software in a virtual reality environment can increase comprehension by using real-world-based visual metaphors that represent software in a familiar context that can easily be identified by the programmer [32]. In a fully-immersive virtual environment, it is possible to get closer to reality, creating a simulation of a real or virtual world in which the user can be present, dive, touch and feel objects [29].

In the research work presented in this paper, our goal is to explore how software comprehension improves by allowing users to view and change the system in a live way using virtual reality. A prototype was developed for Java systems that receives information about the static and dynamic analysis of the system, using reverse engineering approaches, such as the ones presented by Fauzi et al. [8] and Guéhéneue et al. [12]. The tool allows visualizing the software using visual metaphors, in real-time and during the execution of the system. The interaction with the system in full execution is a crucial factor, to get closer to the experience of live programming and create a fluid feedback-loop between the program and the programmer. In this work, the approach of *Live Software Development* resorts to the virtual reality for the construction of an environment, through which it is possible to understand the system and visit it in an interactive and immersive way.

The work here presented expands on previous work from the authors in the *Live Software Development* paradigm [1,3,18] and provides a more detailed account of a user study that was performed to validate the merits of the approach.

The paper is structured as follows: Sect. 2 overviews the current state-of-the-art on live programming, software visualization, software analysis, and virtual reality; Sect. 3 overviews our approach towards live software development, including architecture details; Sects. 4 and 5 present the user study and its results; and, finally, Sect. 6 provides some final remarks and hints for future work.

2 Literature Review

This work involves different areas, such as *Live Programming*, *Software Visualization*, *Software Analysis* and *Virtual Reality*.

2.1 Live Programming

The fundamental notion of live programming is not having a traditional program development cycle involving four phases—*edit, compile, link, run*—but only one phase, at least in principle. This phase consists simply of having the program always running, continuously, even if various editing events occur [30].

Live programming embraces the concept of *liveness* to ease the programming task by executing a program continuously during editing (real-time programming). Looking back at Hancock's analogy, consider hitting a target with

a stream of water: we *receive continuous feedback on where we are shooting*, whereas, with archery, we need to shoot (run the software) and rely on the discrete feedback (debugging) provided by the point the arrow hit, adjusting the aim if necessary [13,21].

Liveness is a notion originally observed in LISP machines and the Smalltalk language, as examples of live programming in the earlier days of computing. Liveness is closely related to visual programming, which provides a more straightforward and intuitive interface to develop and modify software.

2.2 Software Visualization

The software is inherently invisible, which does not help the task of understanding how it functions. Visualization tools are useful to associate a tangible representation to the code and the program execution. Visualizations are especially relevant in the maintenance, reverse engineering, and re-engineering cases [16].

Bassil et al. show evidence that the most common visualization methods are based on graphs, and there are plenty of examples in literature [4,26] that represent the relationships between levels of a system using graphs [5].

CodeCrawler [17] is a tool to visualize data retrieved from other reverse engineering tools, offering a visual encoding that allows representing five metrics per entity. For this type of visualization, we need to choose the layout, the five metrics out of a defined list [17], and the entities representing those metrics.

Jinsight is an example of a tool created to visualize program runtime data. It provides multiple views to increase the probability of the user detecting existing performance issues, unexpected behavior, or bugs. The *JVM* profiling agent [6] provides the data used by this tool.

While the most common software visualization methods are two-dimensional representations, some authors present a 3D representation of the architecture of software as a city, where the user can freely move around and observe and interact with the system [24,34]. This approach is a pure visualization system, and does not deal with real-time modifications of the running system.

Teyseyre et al. [31] discusses the use of 3D software representations and how they have been approach up until this point. Representations have mostly been in one of two ways: abstract visual or real-world representations. Abstract visual representations are graphs, trees, and other abstract geometric shapes, while an example of real-world representations is a city metaphor.

2.3 Static and Dynamic Analysis

The source code is the representation most familiar for developers. It is how software is built and modified. However, it is not necessarily the best when the goal is to ease software comprehension. For that purpose, different and higher levels of abstraction are useful to increase the developers' understanding of the software. *UML* is an example of a higher-level representation of a system's structure and behaviour [25], being amongst the most popular for object-oriented systems.

To develop a higher-level abstraction, firstly, it is required to obtain the existent structural information from the system. Feijs et al. [9] describe a model for analyzing architecture: Extract-Abstract-Present. *Extraction* consists of retrieving structural information from the system, *abstraction* is the derivation of new relationships between the components obtained in the earlier phase (i.e., further analysis of those components) and the *presentation* of that information through a graphical format.

Software Reverse Engineering. Fauzi et al. [8] identify reverse engineering as a valid approach to generate sequence diagrams that reflect a system's behavior.

Although one may assume reverse engineering makes use solely of static representations, such as source code or bytecode, this is not the case. There are several situations where the static and dynamic analysis must be combined. Guéhéneuc et al. [12] demonstrate how a mixture of static and dynamic models allows for a more precise automatic generation of class diagrams. Furthermore, Shi et al. [28] describe *PINOT*, a tool to automatically detect design patterns from both the source code and the system's behavior.

Abstract Syntax Trees. Abstract syntax trees (*AST*) are data structures used by compilers to create intermediate representations of the software that ignores unnecessary syntactic details [14]. This makes it an interesting starting point for analyzing the structure of a software system. Related works include visualizing the evolution of a software project by analyzing the *AST* between commits, as opposed to the typical *file diffs* done by version control systems [10].

Dynamic Analysis. Obtaining a software system's structure is not sufficient to understand how it behaves. There are multiple sources of variability that cannot be taken into account during static analysis, such as user input, the performance of shared resources and variable control flow paths [11].

To compensate for this lack of information, the system should be observed during runtime. For example, logging is a very common practice in software development to record dynamic information of a program's execution [36].

Dynamic analysis can be implemented in multiple ways. Gosain et al. [11] describe the different approaches and tools associated.

2.4 Virtual Reality

Virtual Reality (VR) is used to create real or virtual simulations, applies the theory of immersion in a 3D virtual space where the senses resemble the real world [29]. The presence of investments in the research and development of VR has been driven by the decrease in size and costs of VR equipment, such as headsets. For example, nowadays anyone can have a VR device, be it more sophisticated or cheap, created with a card.

Although still few software visualization tools use VR for comprehension tasks [22], some applications have already been developed. As an example, VR

City [32] uses an animation to demonstrate which classes and methods undergo changes in a sequence of commits, previously provided to the tool.

In general, the use of an immersive environment is an added value for visualization and interaction with the created representation of the software system.

3 Live Software Development Environment

This work aims at providing a Live Software Development environment for improving comprehension by visually representing the software structure and runtime behavior using VR. The overall architecture is depicted in Fig. 1.

To obtain relevant information about the software system under analysis, we developed an extraction and storage framework, which uses static and dynamic analysis strategies. The metadata extracted with the framework (both static and dynamic) is then used by the VR engine, which renders it following a city metaphor. The visualization provides different visual elements to help the developers to understand the software at hand, as, for example, representing class packages as city blocks and classes as city buildings.

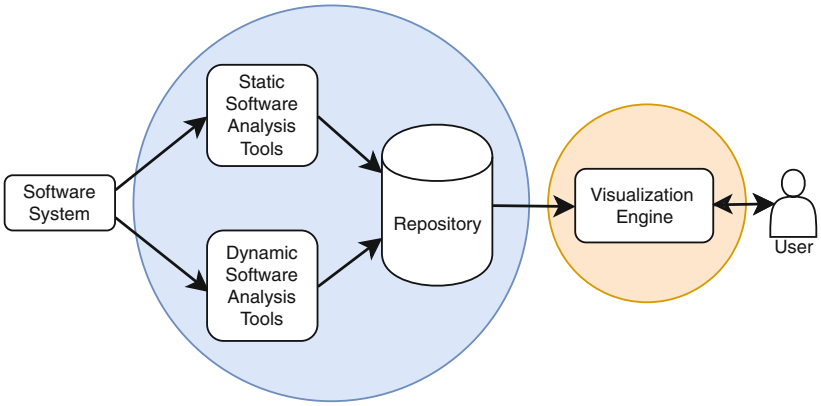

Fig. 1. Diagram of the idealized *Live Software Development* environment [3].

The framework is responsible for the extraction, storage, and provision of information about a software system, so far in Java. The information is extracted from the development environment, without the need to modify the source code itself, and stored in a repository. It is then possible to query this repository and be notified of any modifications in real-time.

As software comprehension is inherently tied to development and maintenance, we assume that the tools which request information from this framework will do so from within a development environment.

The environment allows the visualization of spatial and temporal content through the use of VR. Familiar metaphors allowing the 3D visualization and interaction favor the understanding of the information. At this point, the use of VR equipment, such as simple headsets and controls, allows the user to control the flow of software execution and to traverse the space created by the metaphor. The control of the execution visualization is in the hands of the user.

To develop the environment, the authors considered a series of design concerns, which we describe in the following sections.

3.1 General Approaches to Analyze Source Code

It was first defined how to best identify and analyze the structure and behavior of a *Java* system. This analysis focuses on the source code, and two main paths can be followed: *reverse engineering* and *forward engineering*.

Reverse Engineering. Through reverse engineering, higher-level representations of the software can be extracted, the basis of the static structural analysis. Two representations of a *Java project* are used. First, the *Java Model* used by the *Eclipse IDE*, containing information about the Java elements, such as compilation units, packages, and methods. Second, the AST of the software is used to overlook minor syntactic details of the code and arrive at an easier to understand representation of the source code structure, from package-level down to method-level. Combining these two representations provides the information on how the system is composed and empowers the next process.

Forward Engineering. Forward engineering supports lower-level representations of the system, a process through which we observe the system's behavior.

Approaches to forward engineering include instrumentation, virtual machine profiling, and aspect-oriented programming. After an overall analysis of how straightforward it is to implement these approaches, we concluded that the best fit was a mix of both virtual machine profiling and code instrumentation.

The approach was then used for execution tracing, through event logging, at a granularity that best fits the needs of the visualization component—e.g., logging called methods, the calling class, and the used arguments.

Monitoring would also be a viable option for relevant behavior information. We would need to define resource usage or function execution time thresholds so that an event is logged when one of those thresholds is violated.

3.2 Structural Analysis

The extraction of structural information regarding the software project focused explicitly on the *Java* language. The *Java* AST can be used to abstract syntactic details from the program and provides a structure of the elements considerably more straightforward to interpret than the code itself.

To have easier access to the AST, as well as some other structural details of a *Java* project, and given the assumption of a development environment, the software structure analysis was envisioned as a *IDE plug-in*. *Eclipse* is an *IDE* containing a set of *Java Development Tools* (JDT) which allows *plug-in* developers access to the internal representations of *Java* projects. For this reason, the structure analysis tool was developed as an *Eclipse plug-in*.

Sources of Program Structure. Before designing the internal representation of the workspace for the *plug-in*, it was necessary to understand the structures that *Eclipse JDT* provides access to: the abstract syntax tree (AST) and the *Java Model*. The AST is composed of *ASTNodes* that can be composed of other *ASTNodes*. Each *ASTNode* represents a *Java* source code construct, such as a name, type, expression, statement or declaration. Other classes exist that extend *ASTNode* to include attributes and methods specific to the source code construct that they represent.

Given its proximity with the source code, the *AST* allows fine-grained information about where elements are located in a source file. Nevertheless, the fact that the *AST* is a powerful representation of a project comes with a significant drawback. Due to its fine-grained structural nature, it is considerably more complicated to navigate than the *Eclipse Java Model*.

The *Java Model* is composed of the classes which model the elements that compose a *Java* program. These classes range from *IJavaModel*, which represents the workspace in question, *IJavaProject*, which represents the project itself, to *IMethod* and *IType*, which represent methods and classes respectively.

As the *Java Model* structure is considerably easier to traverse than the AST due to its coarser granularity, it was used as the primary source of information to build the internal model of the project.

Extraction of Program Structure. The actual process of extracting the structure of the projects in the workspace is based on a progressive descent through the *Java Model*. Before the *Java Model* can be analyzed, it has to be generated from the *IWorkspace* class, which represents the workspace in a language-agnostic manner. This is done by invoking *JavaCore* to create a method with the current *IWorkwspace* as an argument.

Once the *Java Model* is obtained, we analyze each project in the workspace. The analysis of an element of a certain level in the *Java Model* implies the analysis of all their child elements. For example, analyzing one project implies analyzing that project's package fragments, which further implies analyzing each package fragment's compilation units, and so on.

Although this process may seem trivial, there are some points worth noting regarding the extraction of the lower-level elements in the model. There are cases in which obtaining the child elements of a specific parent element is not as linear as calling a *getChildElements* method which returns an array of said child elements. This is the case when obtaining both the classes' methods and the method invocations within them.

The complexity in obtaining these two types of structural elements arises from being necessary to, in both cases, obtain information from the *AST*, to be used in conjunction with the information from the *Java Model*.

Live Changes. One of the crucial features of the *plug-in* developed for the statistical analysis is the ability to detect changes to the source code in real-time and reanalyzing the changed elements.

The *Eclipse JDT* provides the mechanism to implement an element change listener, which calls a predefined function once there is a change to a *Java* element inside the *Eclipse IDE*. The callback function will receive as an argument the *ElementChangedEvent*, from which we can obtain the *IJavaElementDelta* that contains information about the element changed.

As *IJavaElementDelta* informs us of the element changed, the representation of the project in the *plug-in* does not have to be rebuilt from the start. Processing time is thus saved by only analyzing the affected elements, from the *Project* level to the *Compilation Unit* level.

Although it would be interesting to allow modifications at the *Method* level, *Eclipse JDT* does not provide a notification of a change in a *IMethod* when the method body is changed, only a *ICompilationUnit* level notification. The lowest change listener implemented was therefore at the *Compilation Unit* level.

When communicating the result of this partial analysis to the repository, the *JSON* data sent is the part of the aforementioned *JSON* structure relevant to the element level analyzed. The request is then sent to the endpoint corresponding to the respective element: `/projects`, `/packages` or `/i-classes`.

Another critical factor in guaranteeing consistency is the analysis of the workspace when the *IDE* is launched. This compensates for any changes that may have been done to the source code from an external tool. Also, this establishes a mechanism to restore the projects' representations to a safe state if any inconsistency issues occur during the detection of live changes.

It is also important to note that if there are any issues with the analysis as a result of incorrect source code (i.e., invoking nonexistent functions), the model is not generated, and the changes are not propagated.

3.3 Runtime Analysis

The software's behavior upon execution is also important, to know how a piece of software functions.

However, a runtime analyzer should be minimally invasive—the logging concerns should be as decoupled from the software to be analyzed as possible. This concern excludes the case of merely implementing a logger as a class in the project and then calling a *log* method whenever it is relevant, adapting it to whichever context it is called.

AspectJ provides a way to achieve such segregation of concerns, by weaving *advices* into the original code. For the analyzer code to be weaved into the project in question, we need to choose the relevant *join points*, define the *pointcuts* and the *advices* [15].

The first concern is to choose the relevant **joint points**. These are the points in a *Java* project in which *AspectJ* allows us to introduce advice. Examples of *join points* are method calls, method executions, constructor calls, field reference, and exception handlers, among others. For our analyzer, however, we chose to only focus on method calls.

Secondly, it is necessary to define the **pointcuts**, that is, exactly what instances of the *joint points* are weaved with the *advice*. Since the goal is to

build a generic method call logger, the conjunction of *pointcuts* must include all the calls of the system to be analyzed. The *pointcuts* used by the analyzer are the *call pointcut*, which gathers all method calls, and the *within pointcut*, to exclude all method calls from within the classes of the runtime analyzer itself.

Given the fact that the analyzer is provided as a *AspectJ* project, the user can add pointcuts to the existing advice. One possible application for this would be to select method calls originating from a specific class or package by using the *within* pointcut. Besides allowing for more targeted analysis, it would help the communication process run more smoothly since the amount of information being sent would be reduced.

Finally, we need to define the aspect **advice**, which specifies the code that is weaved into the original source code upon compilation, at each *pointcut*. As we want to have a notion of the order of method calls, the advice is weaved to run before the method calls.

Figure 2 shows the partial definition of the aspect used to monitor method calls (missing the rest of the advice). The joinpoint corresponds to *call*, while the rest of the pointcut specifies that the advice should not be weaved into method calls of the execution analyzer. Finally, the advice recovers information from the method call and hands it over to the communication interface to send the method call to the repository.

```
public aspect MethodInvocation {
    pointcut methodInvocation() :
        call(* *(..)) && (!within(MethodInvocation)) &&
            (!within(communication.Logger)) && (!within(communication.RepositoryInterface)) &&
            (!within(communication.Startup)) ; //&&
            //( insert other calls here || call);

    before() : methodInvocation(){
        System.out.println("NOW\n");

        Startup.getInstance();

        JSONObject event = new JSONObject();

        event.put("this", thisJoinPoint.getThis() == null ? "static" : "instance");
        event.put("target", thisJoinPoint.getTarget() == null ? "null" : "exists");
        event.put("kind",  thisJoinPoint.getKind());
```

Fig. 2. Definition of the aspect which monitors the execution [3].

A user-interface was not built for this, but a developer could easily modify the aspect where the comment *"insert other calls here"* is done in Fig. 2, and add *within* pointcuts to focus the extraction on classes or packages of interest. This reduces the toll on the repository and allows them to focus specifically on the particular method calls of a small set of classes.

Upon compiling the project, *AspectJ* instruments the resulting code by inserting the code defined in the advice in the points specified by the advice.

The main goal of this process is to extract the most valuable information without compromising the dimension of each *event*, considering there is a massive amount of method calls in a typical piece of software and that these *events* have to be handled by the repository.

The analyzer also obtains an array of the arguments used in the method call and for each stores its type (*type* field) and whether it is null or not (*value* field).

3.4 Communication

Communication is of utter importance, given the large amount of data it may transmit. To reduce the impact of the analysis and the latency with which *events* arrive at the repository, and, consequently, to the visualization engine, two approaches are adopted: *asynchronous requests* and *buffering*.

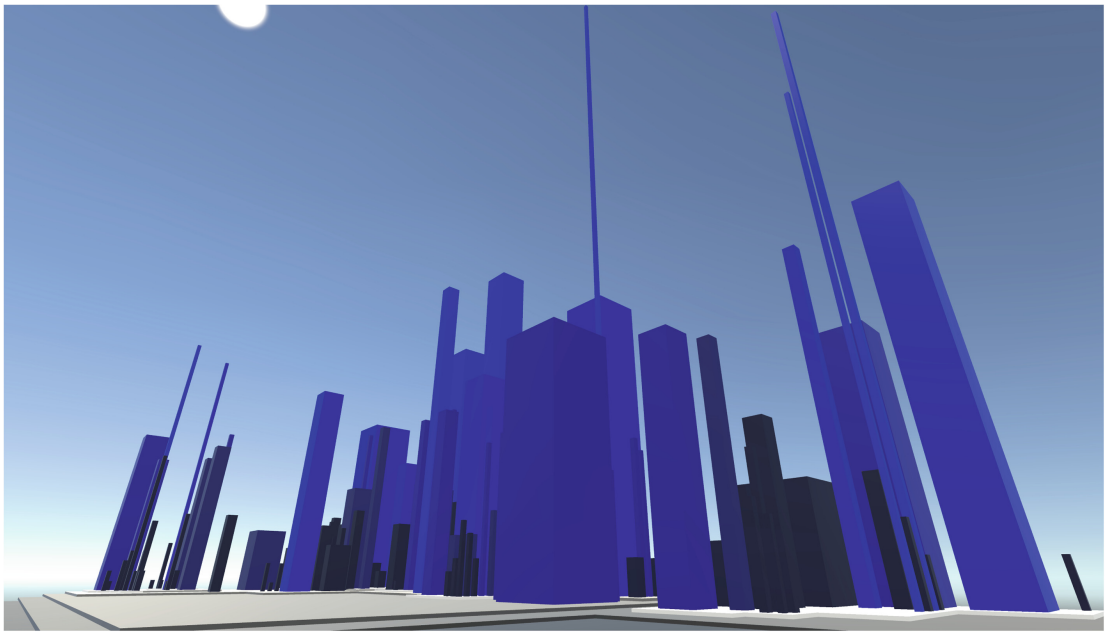

Fig. 3. Structure of the visualization using the engine tool on *JUnit* project [3].

Asynchronous requests are the most straightforward improvement that can be implemented, especially taking into account that no return information must be processed. As we favor reduced latency over the guarantee that all *events* are received, asynchronous requests avoid stopping the execution of the original software from sending a request and await the server's response. This significantly reduces the performance impact of the analyzer.

The second mechanism is buffering *events*, that is, storing events in an array and sending a request with all the stored events, clearing the array afterward, and repeating this process at a fixed time interval. The reasoning behind using buffering is to minimize the impact of the inherent latency of communicating with the server. Similarly to the reasoning behind sending the whole project structure in a single request, it is better to send one large request and allow the server to process it than to send a large batch of smaller requests.

Though buffering may affect the notion of *liveness*, it prevents unordered *events* and avoids, or at least reduces the likelihood of overwhelming the communication channel with massive amounts of small requests.

3.5 Visualization Engine

The visualization engine seeks to combine the best of both worlds: liveness and virtual reality. The virtual environment is responsible for visualizing static and dynamic content, while the use of liveness increases and improves the feedback of the software transmitted to the user. The VR feature for visualizing the 3D content is critical for the immersion. Figure 3 is a static sample of the tool's features.

City-Based Metaphor. The city-based metaphor was selected for this project due to its frequent appearance among different literature about software visualization. Further, this metaphor is easily recognized by a developer as it is based on city buildings, roads, and typical city blocks.

The mapping performs the conversion from packages, classes, and invocations information into districts, buildings, and connections. The whole environment is built using blocks. The dimensions and colors of the blocks are defined through metrics obtained from the software. Block allocation also follows a predefined rule, to minimize the total space required for the construction of the city, maintaining a rectangular space and instantiating the elements by dimension.

The tangibility created with the city metaphor allows us to take a different stance on code understanding.

Interaction Actions and Interface. Being a virtual live environment, the invocations that occur would be imperceptible, since they may happen in less than a millisecond. To view and analyze the software, the engine generates the connections when it receives them, and adds 3 s of duration so that the user has the necessary time to understand what is happening. Also, the user has in his possession other time controls in the environment menu.

Using the controllers and sensors of the VR devices, the user can perform several actions. **Pause**—block any changes in the environment, either with connections or with districts and packages. This is the ideal time for the user to make his analysis because he has total temporal freedom. **Start Live**—return to the live state, after a pause; i.e., back to real-time operation, ignoring everything that might have happened at the time it was paused. **Continue**—continue to execute at the next point to the one that was in the moment that paused the execution. All events that the engine received and were not viewed are cached and can thus be recovered. **Go back 1 second**—despite the intentional delay created in the changes that occur in the environment, the user may lose some detail, and may want to go *back in time*. This feature asks the server for the events that happened in the last second and returns to show them.

Navigating the virtual world is achieved by physically moving the user, or by using the *teleport* functionality.

The user interaction with the virtual environment is possible using only one monitor of a computer. However, the visualization loses its immersiveness and the interaction becomes impracticable due to the non-existence of controllers.

As a result, it is advised to use VR devices with hand controllers and sensors, such as HTC Vive or Oculus Rift.

4 User Study Design

The environment presented in Sect. 3 has the goal of reducing the effort of understanding a software system, hence shortening the length of the feedback loop required to change it or debug it. The controlled experiment detailed in this section has the goals of exploring the potential of the environment for understanding concrete software systems, and of validating the relative effectiveness and efficiency of developers when using it.

4.1 Guidelines

A user study should have into account multiple concerns to reach its goals. The following guidelines were considered when designing the experiment [34].

- **Pedagogical Goals.** As other empirical studies with students, there was the goal of aligning it with educational objectives and the learning process [35].
- **Software Development Experience.** The participant must be familiar with the mechanisms of understanding software systems. Thus, all participants should have experience in software development.
- **Participant Motivation.** Using new technologies as the case of virtual reality devices often arouses the interest of potential participants and is a motivation for signing up to participate in the experiment [27].
- **Familiarity with the Environment.** A reliable and fair comparison of the tool requires prior training with the goal of giving participants the basic knowledge to use the hardware and software. Such a tutorial should be done, if possible, sometime in advance from the experience [20,27,34].
- **Duration.** Participants should have a maximum length of time to perform the tasks and be informed about this limit [27].
- **Project Selection.** Identify student project cases to use in research without interfering with educational goals [35].
- **Prior Knowledge.** To maximize confidence in the results, the participants should have roughly the same experience using the environment [27].

4.2 Experimental Design

The guidelines described in the previous section were considered when designing the experiment, as detailed below.

Participants. The participants were 25 subjects from an academic context—students, researchers, and professors. All had strong programming training and participated freely and with the interest of knowing and exploring the tool.

Physical Environment. The experiment was carried out in a closed room in the Department of Informatics Engineering of the Faculty of Engineering of the University of Porto. The room had a computer screen, which showed what was visualized by the participant, and a free space of 3×3 m for the user's movement. If the participant approached the boundaries of free space, a bounding grid would appear in the virtual environment, prompting it to move away. For interaction with virtual reality, the *Oculus Rift* device was used.

Questionnaire. The experiment included filling out different parts of a questionnaire, one for each of the tasks described in Sect. 4.3. The questionnaire allowed to characterize the participants, to provide insights about the tasks, the interaction, and the usability of the environment.

The questions were designed using a Likert scale by dividing the possible answers into *Strongly Disagree*, *Disagree*, *I Have No Opinion*, *Agree* and *Strongly Agree*, or *Very difficult*, *Difficult*, *I have no opinion*, *Easy* and *Very easy*. For analysis, values from 1 to 5 were assigned, respectively, to the two previous scales, with 1 being the negative evaluation and 5 the most positive evaluation.

Duration. Participants could use the environment for 25 min. Having a hard limit for the duration of the experiment is valued by the participants as it allows them to manage their time better. It is also important because it creates the same time base for the completion of tasks for all participants. Sensalire et al. investigated the average duration of experiments taking up to several hours [27].

Exposure to the Environment. The participants received, before starting the experiment, a tutorial of use of the tool and its main functionalities. The first task had the goal of exposing the participant to all the features of the environment and obtaining feedback on the ease of interacting or visualizing elements of the software system. The time allowed for this first task was what was necessary for the participant to know and understand the functionalities.

Data Integrity. All participants were also informed of the project's authorship before the experiment started. To try to maximize criticism and reduce participants' generosity in responses, which would bias the data, criticism was encouraged by making all responses anonymous.

4.3 Tasks

A small project called *Maze* was selected as the target for the tasks. The project was developed by students in an undergraduate course and had the right dimension to make understanding attainable while still having some degree of architectural complexity, that is, it contains a reasonable number of each one of the structural elements (e.g., packages, classes).

The experiment consists of three tasks, accompanied by different sets of questions, to evaluate different factors in the virtual environment.

Task T1: **Learning to use the Virtual Environment.**
The participant must learn how to use the available features and controls of the VR device. After this introductory phase, the objective is to experience the visualization of all possible components and interactions. Figure 4 illustrates several participants performing T1 on their first exposure to the tool.

Fig. 4. Participants in the first exposure to the virtual environment (Task 1).

Task T2: **Identifying an Infinite Loop.**
The participant had the goal of identifying an infinite loop by using the virtual environment (Fig. 5). The invocations within the loop occur sequentially. The participant could not interfere with the loop, which had no possibility of being terminated. The participant is informed that on the IDE side, there is no error, and the system is blocked.

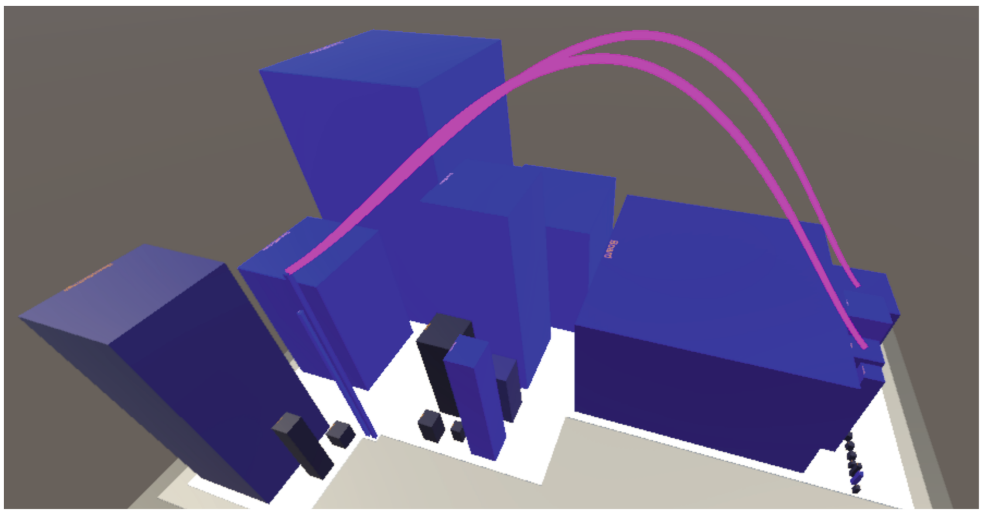

Fig. 5. Display of infinite loop invocations (Task 2).

Task T3: **Identifying a *null* Method Argument.**
The participant had the goal of identifying the class that made a method call with a *null* argument Fig. 6 shows such class. Its color flashes red, and an audible alert is triggered so that the participant is alerted of the event. The sound is useful when the class that performed the invocation is not in the participant's field of vision.

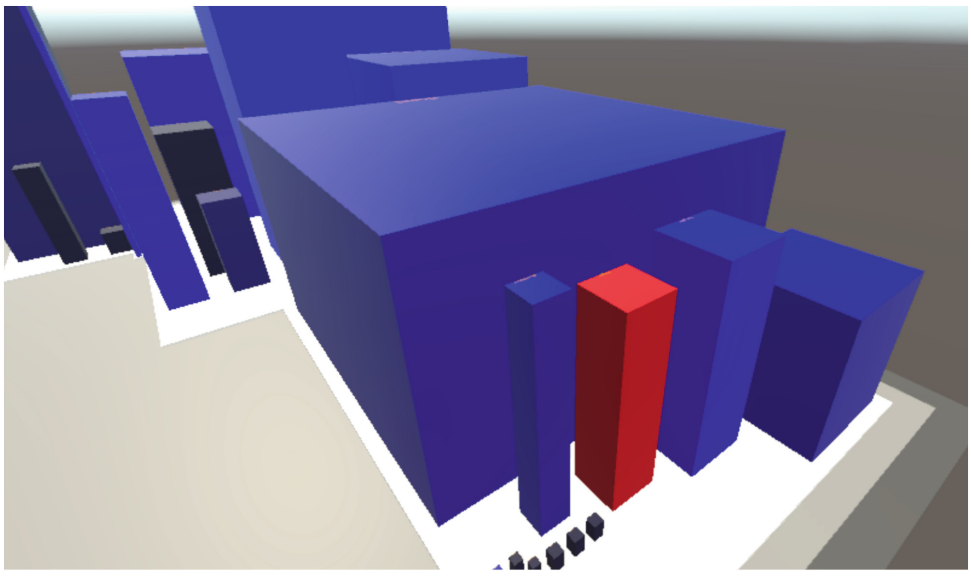

Fig. 6. Display of class performing an invocation with a null argument (Task 3). (Color figure online)

4.4 Data Sources

In addition to the answers in the questionnaires, two attributes of interest were collected throughout the experiment.

- **Duration.** Each of the three tasks was timed. This metric is important to compare participants and to calculate the mean.
- **Difficulty.** The perception obtained by a control agent—the first author—of the execution of each task. This perception is quantified from 0 to 5, with 5 corresponding to a task performed integrally and with a good performance.

Participants had no notion of time in the virtual environment and weren't aware of the difficulty they showed in solving the tasks as perceived by the control agent. Duration is a quantitative attribute, easy to measure, and difficulty is a qualitative attribute, that was based on the opinion of the control agent.

5 Results

The data collected by the questionnaire and the control agent is analyzed and discussed hereafter.

5.1 Subject Characterization

The experiment had the participation of 19 male subjects and 6 female subjects. They included undergraduate students (12%), graduate students (72%), people with a master's degree (12%) and with a Ph.D. degree (4%). Ages range from 18 to 36 years ($\bar{x} = 23.44 \pm 3.15$).

All participants confirmed they had programming experience, with 88% of them programming practically daily. More than half of the participants confirmed that they often experienced some difficulty in understanding software systems (76%) and almost all use tools to help them understand them (92%), but 88% never used *visual* tools to aid in code comprehension.

5.2 Experience with the Oculus Rift

The familiarity of the participants with the device used to perform the tasks, the Oculus Rift VR Headset, in particular with its controls, is shown in Table 1.

Table 1. Number of participants with Oculus Rift experience.

Oculus rift experience	# of participants
None	18
Some	4
Considerable	3

Having previous experience with the device, either in the use of controllers or in the experience of immersing in virtual reality, could make the adaptation to the environment easier. A small number mentioned that this was their first experience with VR, but it's to be expected that many users in a broader context would also not have had previous access and experience with VR headsets.

5.3 Task 1 (T1)

The first of the three tasks (T1) is explained in Sect. 4.3. It exposed the participant for the first time to the virtual environment and its features.

Figure 7 presents the data collected via de questionnaire concerning T1, and shows a general agreement that it was easy to perform.

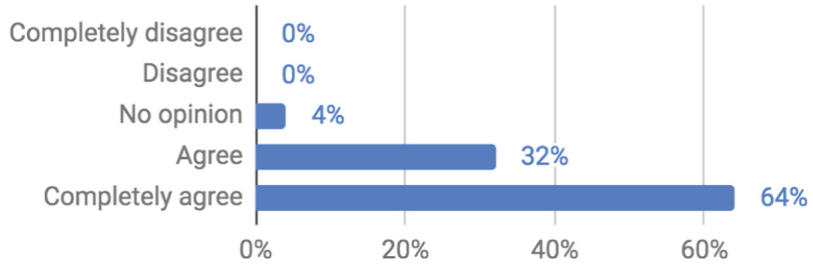

Fig. 7. Answers to T1.1—"Task 1 in general was easy to perform".

The easiness of the recognition and exploration activities of the major features of the environment were also classified. The answers to T1.2a and T1.2b can be seen in Figs. 8 and 9. The first of these figures shows that it's not as

easy to think about packages. This is due to the small dimension of the system being used, which contained only 3 package levels—i.e., 3 district levels—and additionally, all buildings were at the same district level.

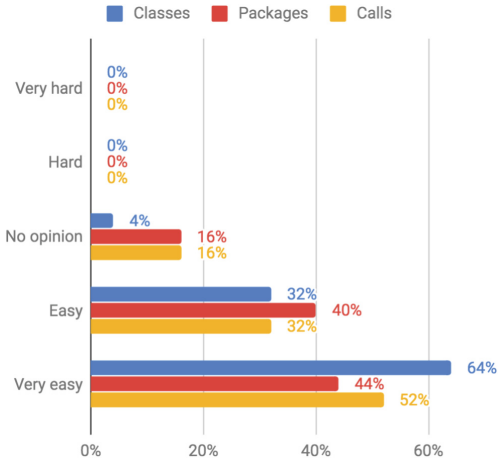

Fig. 8. Answers to T1.2a—"These activities were easy to perform" (identify classes; identify packages; identify method calls).

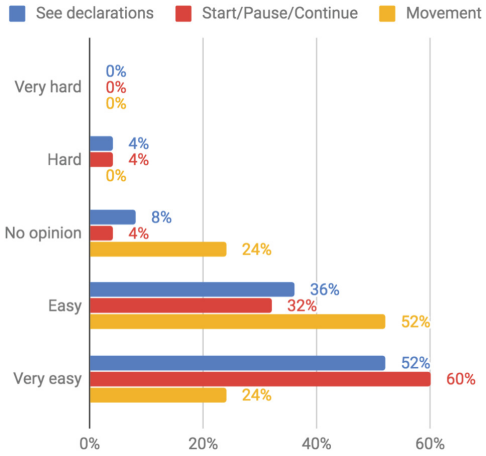

Fig. 9. Answers to T1.2b—"These activities were easy to perform" (see declarations; start/pause/continue execution; movement in the environment).

The T1.2b task required more interaction with the *headset* controllers, which created difficulties for participants who had less experience using Oculus. Physical movement was not used as much as it could, because the participants relied mostly on the teleport functionality.

The evaluation done by the experiment's control agent regarding the difficulty shown by participants when doing T1 had a mean of $\bar{x} = 4.56 \pm 0.65$. This value represents the agent's perception, scored from 1 to 5. This reflects the fact that the task was, in general terms, done positively and entirely by all participants.

In conclusion, the collected data highlights the low difficulty in using the environment by users that are exposed to it for the first time.

5.4 Task 2 (T2)

The focus of T2 was to debug an infinite loop, which blocked the system and the IDE without any error or warning. Upon concluding the task, most of the participants reported no difficulty in identifying the infinite loop (Fig. 10). Figure 11 shows the perception of participants regarding how easy it was to find the issue using the virtual environment when compared with an IDE. Participants found that finding the problem using the visualization was easier, as the IDE wouldn't provide feedback about what is happening and where it is happening, prompting the user to scrutinize the source code or use additional tools.

The controlling agent's assessment of the difficulty shown in solving Task 2 had an average rating of $\bar{x} = 4.24 \pm 0.78$. The average result is lower than that obtained by the participants in the previous task, possibly because it was intrinsically harder to accomplishment, requiring more reasoning.

In short, the participants considered it advantageous to use the virtual environment to perform this type of activity.

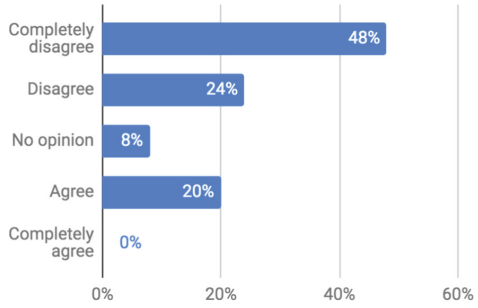

Fig. 10. Answers to T2.1—"I had difficult identifying the infinite loop".

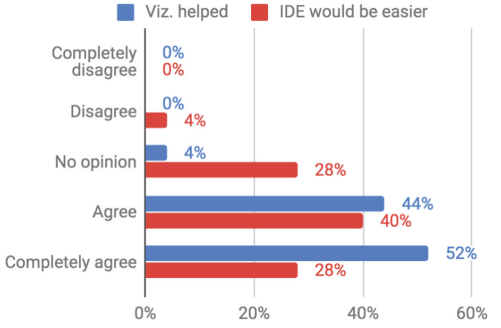

Fig. 11. Answers to T2.2—"In my current IDE I would have a harder time finding the system locked in the loop".

5.5 Task 3 (T3)

Task 3 sought to determine the participants' ability to find a method call in which one of the arguments was *null*, leading to a `NullPointerException`. In the environment, this translated into a building with a flashing red color.

The solution would be to use the *scroll* feature to increase elevation. From the top of the tallest building, participants would not be able to see the object, so this strategy would not be enough to spot the building flashing red.

From the analysis of Figs. 12 and 13, we can conclude that most participants did not have difficulty in performing the task and consider that the visualization helped them to find the problem.

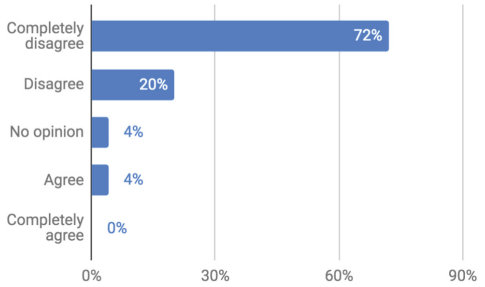

Fig. 12. Answers to T3.1—"I found it difficult to identify the invocation that threw the null exception".

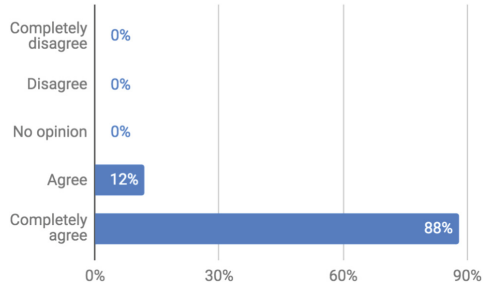

Fig. 13. Answers to T3.2—"The visualization helped me find the problem".

Most participants eventually found the most practical solution and used the *scroll* functionality. Some others decided to walk all the streets of the city on the ground and took more time to find the object.

The controlling agent's assessment of the difficulty shown in solving T3 had an average rating of $\bar{x} = 4.84 \pm 0.37$. The task had a low difficulty, and this value is not higher because some participants didn't immediately realize that gaining a top view of the city was the most straightforward way to answer the task.

In conclusion, this feature was found helpful in debugging *null* exceptions. Even though the sound alert feature was not used in this experiment, we expect it to aid in determining the building's location, as the sound propagates in space and can be heard from farther or closer.

5.6 Virtual Environment Participant Assessment

After completing the tasks, participants evaluated their experience using the virtual environment (Figs. 14 and 15).

The set of questionnaire items A1.1a focused on the perception of static and execution information. The results are similar and positive for both scenarios, as participants were able to obtain the bits of information that they needed.

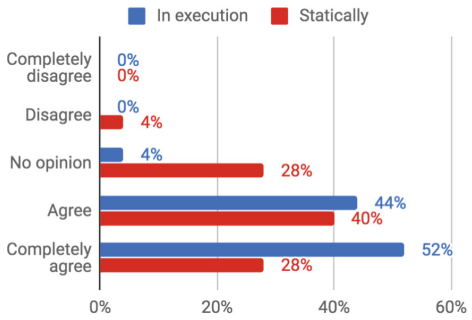

Fig. 14. Answers to A1.1a—"Experience using the virtual environment" (easy perception of static scenario; easy perception of execution scenario).

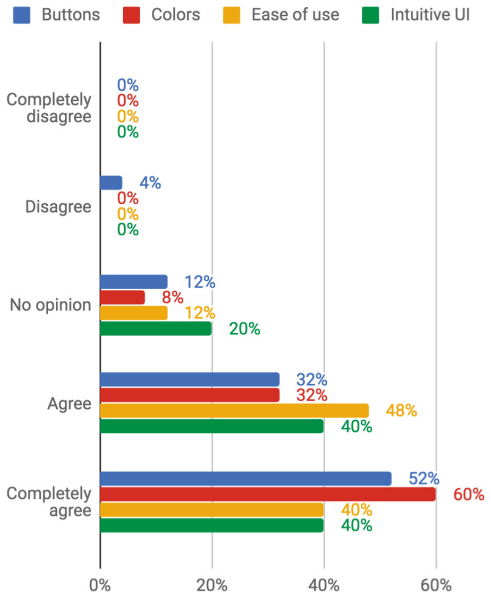

Fig. 15. Answers to A1.1b—"Experience using the virtual environment" (easy understanding of the use of color; button usefulness; intuitive user interface; ease of use). (Color figure online)

The set A1.1b consisted of four statements. The usefulness of the buttons was confirmed for the vast majority of participants except for one who considered

that the buttons should not be in the virtual environment but should be physical buttons present in the Oculus controllers. The use of colors stood out positively because it is one more way of expressing information without requiring any interaction with objects. Regarding ease of use and interface, despite the majority of positive responses, participants considered that a longer exposure time favored the use of the tool and interaction with the interface.

Regarding the participants' interest in using the tool again, 96% agreed, and only one participant did not express an opinion on this statement (Fig. 16).

This interest is positive because participants in the VR experience, in some cases in their first experience, felt no discomfort in using it and considered an idea with potential for use in real-world contexts.

Participants also expressed if they found the virtual environment beneficial to understanding software systems (Fig. 17). The results are encouraging to the objective of the virtual environment, due to the significant presence of positive responses compared to no negative ones.

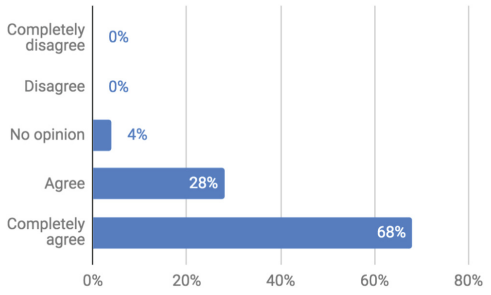

Fig. 16. Answers to A1-2—"Would use the tool again".

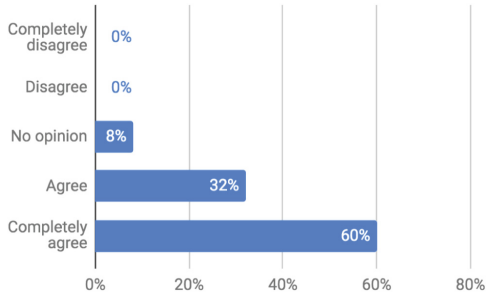

Fig. 17. Answers to A1-3—"I find the virtual environment beneficial in understanding software systems".

In conclusion, considering the overall assessment of the virtual environment, participants found advantages in using the tool.

5.7 System Usability Scale

Finally, the participants answered the system usability scale, which consists of 10 questions, to help us identify usability aspects that deserve further attention.

Figure 18 shows the percentage of answers given for each of the 10 questions. The colors used in the table are intended to reduce the effort to understand it, with values close to 0% being red and those around 100% green.

After analyzing the results and following the calculation of the System Usability Scale, a global rating of 83.8 was calculated, which translates into an A grade, i.e., the highest possible grade. Participants thus show interest in the usability of the tested tool.

	I think that I would like to use this system frequently.	I found the system unnecessarily complex.	I thought the system was easy to use.	I think that I'd need support of a technical person to use this system.	I found the various functions in this system were well integrated.	I thought there was too much inconsistency in this system.	I'd imagine that most people would learn to use this system very quickly.	I found the system very cumbersome to use.	I felt very confident using the system.	I needed to learn a lot of things before I could get going with this system.
Completely agree	40%	0%	52%	0%	56%	0%	44%	0%	44%	0%
Agree	48%	4%	44%	28%	32%	0%	52%	0%	48%	8%
No opinion	12%	4%	4%	16%	12%	4%	0%	8%	8%	16%
Disagree	0%	36%	0%	28%	0%	20%	4%	0%	0%	24%
Completely disagree	0%	56%	0%	28%	0%	76%	0%	0%	0%	52%

Fig. 18. Overall assessment of the virtual environment. (Color figure online)

5.8 Challenges

From the analysis of the presented data and the feedback received from the participants, their biggest challenge was the little experience they had using VR headsets. This issue represented some learning time and greater difficulty in interactions due to inadequate knowledge of how the controllers worked.

Figure 19 highlights that Oculus Rift's user experience has favored T1, reducing the time required to realize it, i.e., the exposure time required to explore the virtual environment reduced. The coefficient of determination was approximately 0.6339.

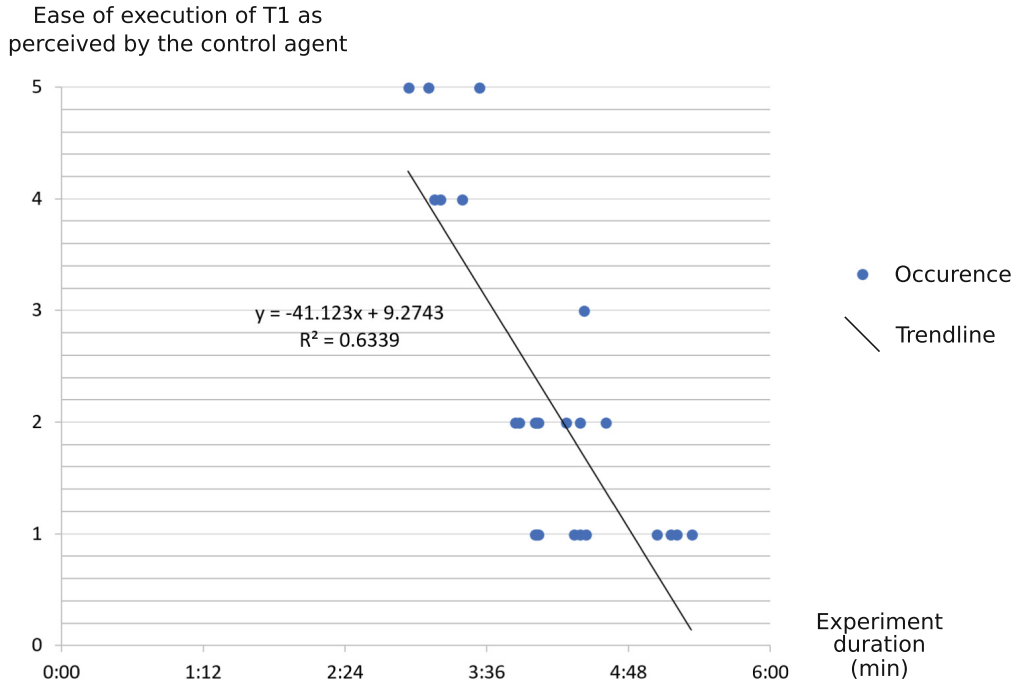

Fig. 19. Oculus experiment durations spread for T1.

The controlling agent of the experiment noticed an evolution in the ability to interact with the system throughout the experiment, suggesting that if there were additional tasks, or if the same participants would participate of future experiments, they would feel more confident in using the VR device and would achieve better results.

A similar analysis was performed for tasks T2 and T3, comparing the duration of the experiment to the Oculus usage experience, but no significant results emerged. That is, having more or less experience with the VR device had no impact on the time required to solve the problem. In T2, it may be more significant to compare participant programming practice with duration. However, as all are programmers, in particular, 88% of participants said they program practically daily, such an analysis would not produce meaningful results.

6 Conclusions

The contributions described in this paper consist of **(1)** a structural analysis tool for *Java* projects that can be included in any *JDT*-enabled *Eclipse IDE* as a plug-in, and is capable of recognizing changes to several levels of the *Java Model* tree; **(2)** an execution analysis tool for *Java* projects that can be included in the relevant workspace and added to a project with minimal required modifications to the concerning source code; **(3)** a software repository ready to receive information from the previously mentioned analysis tools, and provide it in real-time through a *API*; **(4)** a VR application for the live visualization of a software system that makes use of visual and spatial metaphors to facilitate software understanding; **(5)** a controlled experiment to directly validate the merits of the combined use liveness and VR visualization, and indirectly validate the remaining contributions.

The controlled experiment showed mostly agreement for the benefits and usefulness of this approach for visualizing software and provided some insights to be used in future work. Regarding future improvements, these include adding new spatial and temporal interactions, and two-way communication between the visualization, the repository, and the running system, which would allow modifications in the virtual environment to be passed to the running system more instantaneously, therefore improving the liveness of the developers experience.

References

1. Aguiar, A., Restivo, A., Figueiredo Correia, F., Ferreira, H.S., Dias, J.P.: Live software development – tightening the feedback loops. In: Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming. Programming 2019 Companion (2019)
2. Alam, S., Dugerdil, P.: EvoSpaces: 3D visualization of software architecture. In: 19th International Conference on Software Engineering and Knowledge Engineering, vol. 7, pp. 500–505. IEEE (2007)

3. Amaral, D., Domingues, G., Dias, J.P., Ferreira, H.S., Aguiar, A., Nóbrega, R.: Live software development: an environment for Java using virtual reality. In: Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, vol. 1, pp. 37–46. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007699800370046
4. Bartoszuk, C., Timoszuk, G., Dabrowski, R., Stencel, K.: Magnify - a new tool for software visualization. In: 2013 Federated Conference on Computer Science and Information Systems, pp. 1485–1488, September 2013
5. Bassil, S., Keller, R.K.: Software visualization tools: survey and analysis. In: Proceedings of the 9th International Workshop on Program Comprehension, IWPC 2001, pp. 7–17. IEEE Computer Society, Washington, DC (2001)
6. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the execution of Java programs. In: Diehl, S. (ed.) Software Visualization. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45875-1_12
7. Elliott, A., Peiris, B., Parnin, C.: Virtual reality in software engineering: affordances, applications, and challenges. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, vol. 2, pp. 547–550. IEEE Press, Piscataway (2015)
8. Fauzi, E., Hendradjaya, B., Sunindyo, W.D.: Reverse engineering of source code to sequence diagram using abstract syntax tree. In: 2016 International Conference on Data and Software Engineering (ICoDSE), pp. 1–6, October 2016. https://doi.org/10.1109/ICODSE.2016.7936137
9. Feijs, L., Krikhaar, R., Ommering, R.V.: A relational approach to support software architecture analysis. Softw. Pract. Exp. **28**(4), 371–400 (1998)
10. Feist, M.D., Santos, E.A., Watts, I., Hindle, A.: Visualizing project evolution through abstract syntax tree analysis. In: 2016 IEEE Working Conference on Software Visualization (VISSOFT), pp. 11–20, October 2016. https://doi.org/10.1109/VISSOFT.2016.6
11. Gosain, A., Sharma, G.: A survey of dynamic program analysis techniques and tools. In: Satapathy, S.C., Biswal, B.N., Udgata, S.K., Mandal, J.K. (eds.) Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014. AISC, vol. 327, pp. 113–122. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-11933-5_13
12. Guéhéneuc, Y.G.: A reverse engineering tool for precise class diagrams. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON 2004, pp. 28–41. IBM Press (2004)
13. Hancock, C.M.: Real-time programming and the big ideas of computational literacy. Ph.D. thesis, Massachusetts Institute of Technology (2003)
14. Jones, J.: Abstract syntax tree implementation idioms. In: Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) (2003)
15. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45337-7_18
16. Koschke, R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. J. Softw. Maint. Evol.: Res. Pract. **15**(2), 87–109 (2003). https://doi.org/10.1002/smr.270
17. Lanza, M., Ducasse, S.: Polymetric views-a lightweight visual approach to reverse engineering. IEEE Trans. Softw. Eng. **29**(9), 782–795 (2003). https://doi.org/10.1109/TSE.2003.1232284

18. Lourenço, P., Dias, J.P., Aguiar, A., Ferreira, H.S., Restivo, A.: CloudCity: a live approach and environment for the management of cloud infrastructures. Commun. Comput. Inf. Sci. (2019)
19. Maalej, W., Tiarks, R., Roehm, T., Koschke, R.: On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. **23**(4), 31:1–31:37 (2014). https://doi.org/10.1145/2622669
20. Marcus, A., Comorski, D., Sergeyev, A.: Supporting the evolution of a software visualization tool through usability studies. In: 13th International Workshop on Program Comprehension (IWPC 2005), pp. 307–316, May 2005. https://doi.org/10.1109/WPC.2005.34
21. McDirmid, S.: Usable live programming. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! 2013, pp. 53–62. ACM Press, New York (2013). https://doi.org/10.1145/2509578.2509585
22. Merino, L., Ghafari, M., Anslow, C., Nierstrasz, O.: CityVR: gameful software visualization. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 633–637, September 2017. https://doi.org/10.1109/ICSME.2017.70
23. Panas, T., Berrigan, R., Grundy, J.: A 3D metaphor for software production visualization. In: Proceedings on Seventh International Conference on Information Visualization, IV 2003, pp. 314–319, July 2003. https://doi.org/10.1109/IV.2003.1217996
24. Romano, S., Capece, N., Erra, U., Scanniello, G., Lanza, M.: The city metaphor in software visualization: feelings, emotions, and thinking. Multimed. Tools Appl. **78**, 1–37 (2019)
25. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education (2004)
26. Sadar, A., Panicker, V.: DocTool - a tool for visualizing software projects using graph database. In: 2015 Eighth International Conference on Contemporary Computing (IC3), pp. 439–442, August 2015. https://doi.org/10.1109/IC3.2015.7346721
27. Sensalire, M., Ogao, P., Telea, A.: Evaluation of software visualization tools: lessons learned. In: 2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 19–26, September 2009. https://doi.org/10.1109/VISSOF.2009.5336431
28. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from Java source code. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006, pp. 123–134. IEEE Computer Society, Washington, DC (2006). https://doi.org/10.1109/ASE.2006.57
29. Singh, N., Singh, S.: Virtual reality: a brief survey. In: 2017 International Conference on Information Communication and Embedded Systems (ICICES), pp. 1–6, February 2017. https://doi.org/10.1109/ICICES.2017.8070720
30. Tanimoto, S.L.: A perspective on the evolution of live programming. In: Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, pp. 31–34. IEEE Press, Piscataway (2013)
31. Teyseyre, A.R., Campo, M.R.: An overview of 3D software visualization. IEEE Trans. Vis. Comput. Graph. **15**(1), 87–105 (2009). https://doi.org/10.1109/TVCG.2008.86

32. Vincur, J., Navrat, P., Polasek, I.: VR city: software analysis in virtual reality environment. In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 509–516, July 2017. https://doi.org/10.1109/QRS-C.2017.88
33. Wettel, R.: Software systems as cities. Ph.D. thesis, Faculty of Informatics of the Università della Svizzera Italiana, September 2010
34. Wettel, R., Lanza, M., Robbes, R.: Software systems as cities: a controlled experiment. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 551–560. ACM, New York (2011). https://doi.org/10.1145/1985793.1985868
35. Wohlin, C.: Empirical software engineering: teaching methods and conducting studies. In: Basili, V.R., Rombach, D., Schneider, K., Kitchenham, B., Pfahl, D., Selby, R.W. (eds.) Empirical Software Engineering Issues. Critical Assessment and Future Directions. LNCS, vol. 4336, pp. 135–142. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71301-2_42
36. Yuan, D., Park, S., Zhou, Y.: Characterizing logging practices in open-source software. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 102–112, June 2012. https://doi.org/10.1109/ICSE.2012.6227202