2022 Working Conference on Software Visualization (VISSOFT) | 978-1-6654-8092-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/VISSOFT55257.2022.00017

IDEVELOPAR: A Programming Interface to enhance Code Understanding in Augmented Reality

1st Lucas Kreber
University of Trier
Trier, Germany
kreberl@uni-trier.de

2nd Stephan Diehl
University of Trier
Trier, Germany
diehl@uni-trier.de

3th Patrick Weil
University of Trier
Trier, Germany
s4paweil@uni-trier.de

***Abstract*—During software maintenance developers spend a considerable amount of time on tasks like navigating, identifying required code locations or tracing various call hierarchies. The classical tabbed interfaces, as found in modern IDEs, are not ideal for such tasks, leading to an inefficient workflow containing many context switches. Therefore, several programming environments, like Code Bubbles, were proposed to overcome these issues by allowing users to freely arrange code fragments on a canvas to make relations more explicit and better understand the codebase. Relations are made explicit using visual links or extra space between groups of code fragments. As a consequence, these approach quickly run out of screen space. In this paper, we present IDEVELOPAR, a tool to enhance code understanding in augmented reality. Due to the use of AR, a user is not restricted anymore by limited display sizes and can use the entire physical space as a workspace for placing and grouping code fragments as well as making changes to the codebase. First, we introduce the views and interactive functionalities of our tool. Next, we illustrate the usefulness of the tool by navigating an example program to locate and fix a bug. Finally, we briefly discuss the results of a cognitive walk-through using the cognitive dimension framework as well as a formative user study to identify potential usability problems. Moreover, in this study the participants also mentioned several advantages of our approach over the classical one. Furthermore, we found that over time the participants developed their own placement strategies.**
Video URL: https://youtu.be/wCNkLS1qQfM

***Index Terms*—augmented reality, programming tool, program comprehension, code navigation**

I. INTRODUCTION

Software complexity has increased massively over the last decades, leading to an ever-growing codebase in software projects, where maintenance is becoming more and more critical. High-level maintenance tasks like correcting faults, improving run-time or memory performance, extending functionality, or adapting to a changed environment include low-level tasks like reading, navigating and editing source code. In particular, developers have to identify relevant code fragments when working on such tasks. Finding these code fragments is non-trivial and very time-consuming. For example, in a study Ko et al. [1] found that developers spend 35% of their time solely on navigation.

While modern integrated development environments (IDEs) facilitate code navigation, their visual interfaces have a bento-box design that partitions the available screen space into separate areas [2]. Programmers typically need to switch between tabs in order to navigate to a different file. As a result, it is quite hard to remember more than a few navigation steps and already seen code fragments.

Alwis et al. [3] identified several factors potentially triggering disorientation while working on a software project. There is an absence of connecting navigation context. Switching tabs does not lead to a visual connection between files. Furthermore, a developer can not see all the necessary information required for a specific task. So there is a lack of surrounding context regarding the viewed source code. This behavior finally leads to a redundant and inefficient workload. One approach to overcome these issues are the so-called ”Code Bubbles” that visually link different code views and display these views side by side on the same canvas [4]. In a study the authors compared their approach with a traditional IDE (Eclipse) and found that programmers using Code Bubbles successfully completed significantly more program understanding tasks as well as that it took them significantly less time [5].

However, the approach of integrating such a visualization directly into the IDE has the significant flaw of finite screen space. Even though the minimum display size of 24” recommended by the authors is today’s standard in almost every workplace, it reaches its limit when displaying many code fragments or using a laptop computer. Therefore, in this paper, we describe IDEVELOPAR, an Augmented Reality Application that adopts the approach of linked code views from Code Bubbles and leverages the HoloLens 2 to extend the developer’s workspace beyond the limited screen space to an almost infinite space using AR. When using AR, all the code fragments opened during navigation are placed in the surrounding room as 3D objects. These visual objects help users build spatial awareness of the opened code fragments. Navigating in the AR space also creates visual links between opened code fragments to visualize the current navigation history as well as call and usage dependencies. Due to the almost unlimited space, a single code fragment will not disappear, so context switches are brought to a minimum. All these aspects are promising to facilitate the code comprehension of existing projects and simplify the overall very challenging process of navigation.

Our tool provides a fully functional programming interface parallel to an existing IDE where code fragments can be freely

978-1-6654-8092-5/22/$31.00 ©2022 IEEE
DOI 10.1109/VISSOFT55257.2022.00017

arranged in the surrounding physical space. IDEVELOPAR supports basic functions to open, navigate and edit arbitrary code fragments. A developer can use the tool either simultaneously to an IDE or exclusively in the AR space, where all key features of the IDE are provided and accessible through gesture control. Using an augmented space can further support developers in reducing their mental load, as shown by Tang et al. [6], leading to a more efficient and effective workflow.

In the following, we first present the features of our tool in Section II, then we demonstrate its usefulness by describing how to identify and fix a bug in an example program in Section III. In Section IV we present the results of a cognitive walk-through and a formative user study. Finally, we briefly discuss related work in Section VI and Section VII concludes this paper.

II. IDEVELOPAR

IDEVELOPAR[1] is our approach for supporting developers with a fully functional programming environment in Augmented Reality. The tool consists of two parts, the actual augmented reality application running on the HoloLens 2 and any IDE out of the JetBrains family running on a regular computer. Although, our current prototype only supports Java, without much effort, the tool can be adapted to every programming language supported by one of the JetBrains IDEs.

To prepare the IDE for the use of our tool, one only needs to install a plugin that starts all the required infrastructure used by the tool directly in the IDE. All requested information will be sent over the established network connection from the IDE directly to the HoloLens 2, including source code and code completions. All the logic, code analysis, etc., runs in the IDE.

A. Linked Code Panels in AR

The code panel is the core element of the tool, which visually represents a code fragment of the project using augmented reality. A code fragment is either the code of an entire class, a single method or a single constructor. For example, the panel in Figure 3:A shows the code of a Java class. Most of the space is taken up by the code, which automatically uses the color scheme of the IDE. The fully qualified name of a class or method is shown in the lower area of the panel. All changes in a code panel are directly synced to the connected IDE. Various buttons are available in the upper right area of the panel. These are used resize the panel or to close the panel or the complete sub-tree with the panel as its root.

In a code panel a user can click at a class name, a method or a constructor call. As a result a new code panel with the related code fragment will be opened. To indicate the relation to the original code panel a visual link is drawn to connect it to the newly opened code panel. If a user followed a method or constructor call the link is colored green, if the user opened a class declaration the link is colored blue. The emerging code-panel tree, see f.e. Figure 3:B, not only visually represents the navigation history, but depending on the navigation strategy of the user shows parts of the static call graph or the aggregation structure.

[1]The acronym is a composition of the three central core features: IDE, development and AR

Fig. 1. Hand Menu. Menu for controlling basic functions (switch to programming mode, run code on IDE and show project view). The menu appears as soon as the user raises a hand and turns its palm towards their face.

Depending on the distance between the code panel and the user, the font may be too small, making it impossible to read the source code in it. Therefore, if a certain distance is exceeded the information in the code panel is shown at a different level of detail (semantic zooming) with a larger font size. The code panel will automatically switch to an overview representation consisting of only basic information such as the name of the opened code fragment, the corresponding package path or the type of the opened code (class or method). Semantic zooming allows the user to keep track of all open code panels without placing them in close proximity, see Figure 3:B.

B. Code Navigation

We strived to design all interactions to feel as natural and intuitive as possible for a user, allowing an efficient workflow without putting too much effort into managing only the tool's controls.

Basic functionalities are controlled via the hand menu (Figure 1). It appears as soon as a user raises their hand and points the palms towards their face. In the first prototype version, the hand menu has three options: Switching the programming mode on and off, compiling and running the program in the desktop IDE, and opening the project overview. A drop-down menu appears on the right side of the hand menu by clicking the project view option, displaying all available classes structured in the standard package-sorted way in a hierarchical menu (an indented tree similar to the way they are shown in the project tab in the IDE). Packages can be folded and unfolded by clicking on them. The user can scroll the menu using up and down buttons. Clicking on one of the classes will open a new code panel with the code of the class.

The user can scroll through the program code using the HoloLens 2 eye-tracking functionality. By either looking at the top or the bottom of a code panel, the text will scroll in the appropriate direction. When reading longer passages of code,

the text will automatically scroll at the speed of the current reader, without any additional action by the user.

Once the user has opened the first code panel, there are two ways of opening new code panels: via the hand menu as described above or by clicking at a method call or class name in the code shown in a code panel.

When clicking on a method call, the user can choose if the newly opened code panel should display only the requested method code or the entire class code containing the specific method (See Figure 3:C). The equivalent applies when clicking on a class name. Here a user can choose to open only the constructor of the class or the entire code of the class. Whether class or method, every newly opened code panel is connected to its parent panel via a visual link. As a result the navigation history becomes visible in form of a code-panel tree. Every navigation step following a method or constructor call is indicated by a green link. All other navigation steps are visually represented with blue links. Due to this approach, a user can easily distinguish parts of the explored call graph within the navigation tree.

When new code panels are created by navigation, an automatic layout places the new panel right of its parent below already existing siblings. The new panel and its siblings are vertically centered relative to their parent. Figure 3:B shows the layout strategy applied to new panels. Panels created over the hand menu appear in front of the user. At any time, the user can adapt the placement to their needs by simply grabbing a panel and moving it around. The grabbing action is triggered by the pinch gesture, i.e. by pinching the pointing finger and the thumb together. While moving a panel around, the user can rotate the panel by rotating their hand.

Each code panel provides a button to activate an expansion mode. While in this mode, the corresponding code panel can be reduced or enlarged in size by grabbing one of its corners and dragging it either inward or outward. Through this kind of functionality, a user can adjust every code panel to their own preferred size or just put individual, essential elements in the foreground for better recognition.

C. Programming within Code Panels

The source code can be changed directly in each code panel when the programming mode is activated over the hand menu. After activation, a cursor appears in the focused code panel. Through the connected Bluetooth keyboard, a user can make changes as usual. The use of a virtual keyboard is possible but currently not implemented. While typing, full code completion is provided to support the user (See Figure 2).

All code changes are synced to the connected IDE if the programming mode is activated. Thus a potential switch between the 2D desktop and the augmented reality environment is no problem at all and possible without any further delays. This option leads to a free choice of the preferred workplace for a different set of work tasks.

III. Demonstration Study

In this section, we look at an example to demonstrate how to use IDEVELOPAR to identify and fix a bug. The bug is located in a version of the game Super Mario [7] which is implemented in Java. The bug manifests itself in the lack of collision between opponents and the environment. We used the same program and two more bugs in our formative user evaluation in Section V-B.

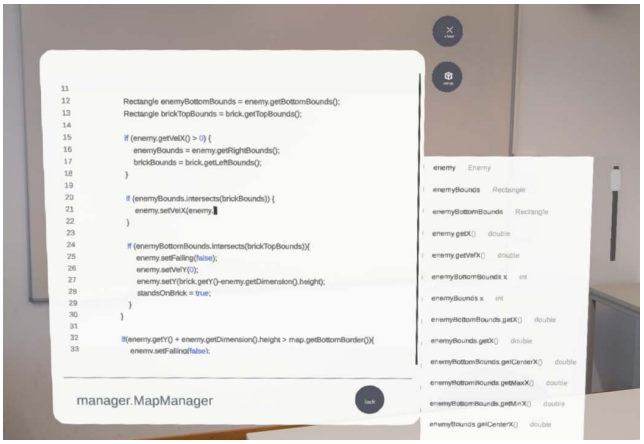

Fig. 2. Code panel in programming mode. In Line 22, the cursor is visible. At the right, a list of code completions suggested by the IDE is shown.

Assume Jane has has already imported the game project in Intellij.To start the IDEVELOPAR-Plugin she clicks at the corresponding icon in the toolbar and puts on the AR glasses. First, she executes the code by opening the hand menu and clicking on “run code” to get a first impression of how the bug affects the game. The game starts, and she can see that there exists no collision between Goombas and bricks. Mario is not affected by this bug. Therefore she starts searching for the affected code location by opening the first code fragment. She opens the hand menu again and clicks on “project view”. The project view appears on the right side of the menu. Because the collision is only missing for Goombas, she opens the class `Goomba`. But the class `Goomba` does not reveal any possible defects. Thus she follows the inheritance hierarchy from `Goomba` to `Enemy` and continues to its superclass `GameObject`. It contains the potentially important method `updateLocation()` where the new location of an object is set (line 67) by adding the horizontal velocity to the previous x coordinate. This is not wrong but could result in incorrect positions if faulty values are used for the velocity. For this reason, she only closes the two previous code panels and moves the code fragment containing the `GameObject` code aside for later inspection. Next, she needs to identify the code location of the collision detection. She starts searching in the class `GameEngine` that contains the `main()` method. This class consists of several hundred lines of code, so she scrolls down the code by looking at the bottom of the displayed code panel until she spots a `run()` method. This method calls the method `gameLoop()` regularly. By clicking on `gameLoop()` and choosing “open method”, she follows the call and opens a new code panel containing only the corresponding method code. In this code the method `checkCollisions()` is

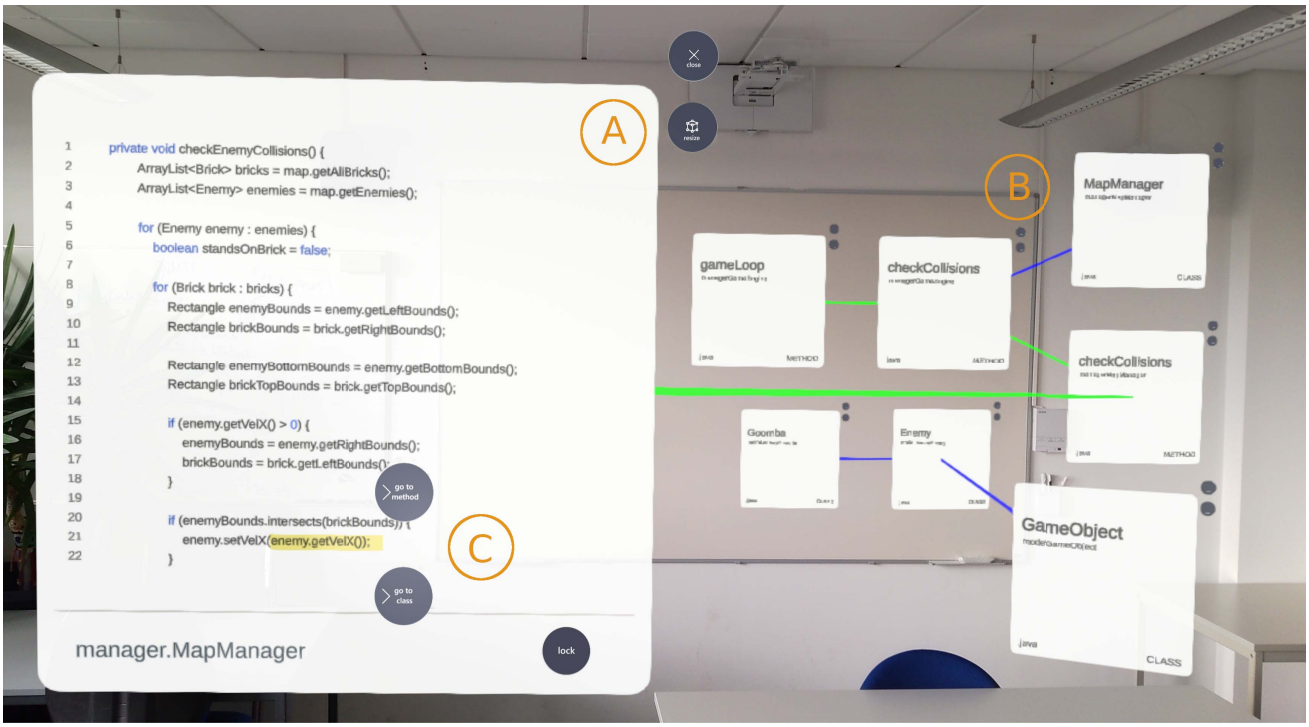

Fig. 3. IDEVELOPAR in practice. The exemplary procedure from section III is shown. **(A)** Opened code panel **(B)** Two independent, visual linked emerging code panel trees shown at a different level of detail (semantic zooming) **(C)** Code navigation, when clicking on a followable code element

called. Because Jane searches for the location of the collision detection, she navigates into `checkCollisions()`. The method body shown in the new code panel only contains the line `mapManager.checkCollisions()`, so calls are actually delegated to another class. To get an overview of the class `MapManager`, she clicks on the called method, but this time she opens the entire code of the class and not only the corresponding method. The class `MapManager` contains various methods for loading the map, assigning points, managing objects, and indeed many methods checking different cases of collisions. To fade out all the methods of no interest, Jane clicks again on `mapManager.checkCollisions()` in the previous code panel, but now she opens only the corresponding method code. In this code she quickly identifies a call of the method `checkEnemyCollisions()`. Navigating to the code of this method leads her to the program logic handling the collisions of enemies, especially of Goombas. Jane finds a line of code (line 229) where the horizontal velocity is set to the current value of the object's velocity if there is an intersection with a brick. By looking at the code panel of the class `GameObject` which she opened at the beginning, she recalls that the direction of enemies depends on the corresponding velocity. And because this velocity does not change, it results in the seen behavior that Goombas do not invert their direction when they collide with a brick. To fix that, Jane she opens the hand menu and enables the programming mode. Then she changes the code in line 229 with the connected keyboard such that the velocity is multiplied by -1. To test if her fix actually solved the problem, Jane runs the program through the hand menu. The bug is fixed.

IV. EVALUATION

To investigate the effectiveness and usability of IDEVELOPAR, we conducted a two-stage evaluation. First, our work group analyzed the tool based on the cognitive dimension framework [8], a technique to evaluate the usability of an existing system. The second stage was a formative user study with a total of eight participants.

1) Cognitive Dimensions Framework: To get a first impression regarding the state of IDEVELOPAR, we conducted a lightweight analysis based on the cognitive dimension framework. This framework "is a broad-brush evaluation technique for interactive devices and for non-interactive notations" [8], defining 14 different cognitive dimensions. Table I gives a short overview of the different dimensions (for more detailed descriptions of each dimension see the paper by Blackwell and Green [9]).

Through the discussion of the different dimensions, we identified the potential usability issues shown in Table II.

In order to not delay the formative user evaluation, we decided to prioritize the identified problems and fix only those that we found to be critical for the user evaluation. For the remaining issues we decided that we would combine them with the insights gained by the user study for a later revision of our tool. Thus, we before the user study, we extended our tool to address the first two points in Table II. This extended version

TABLE I
LIST OF COGNITIVE DIMENSIONS BASED ON [9]

Dimension	Description
Abstraction [CD1]	types and availability of abstraction mechanisms
Closeness of mapping [CD2]	closeness of representation to domain
Consistency [CD3]	similar semantics are expressed in similar syntactic forms
Diffuseness [CD4]	verbosity of language
Error-proneness [CD5]	notation invites mistakes
Hard mental operations [CD6]	high demand on cognitive resources
Hidden dependencies [CD7]	important links between entities are not visible
Premature commitment [CD8]	constraints on the order of doing things
Progressive evaluation [CD9]	work-to-date can be checked at any time
Role-expressiveness [CD10]	the purpose of a component is readily inferred
Secondary notation [CD11]	extra information in means other than formal syntax
Viscosity [CD12]	resistance to change
Visibility [CD13]	ability to view components easily
Provisionality [CD14]	degree of commitment to actions or marks

TABLE II
ISSUES IDENTIFIED USING THE COGNITIVE DIMENSIONS FRAMEWORK

	Issue	Dimensions
1	It should be possible to close a complete path of code panels with one click.	CD3, CD12
2	Visual links should provide more information, e.g. by using labeled links or different link types for different dependencies.	CD5, CD7, CD11
3	Currently, only classes and methods can be opened. In addition, it should also be possible to display arbitrary files.	CD1, CD2
4	Code panels should be more distinguishable by using different shapes for different kinds of data (class, method, or file).	CD1, CD13
5	A visual indication should be shown when several identical code panels (same class or method) are open.	CD6, CD7
6	It should be possible to split a code-panel tree, e.g., for rearranging subgroups of panels.	CD8
7	If navigating to an abstract class, the concrete implementation should be opened in a new code panel, or at least the user should have a choice of opening it.	CD7
8	The state of the application should be persistent. The previous session should be restored if the application is closed and reopened.	CD11
9	It should be possible to pin comments to arbitrary code panels in addition to accustomed comments directly in the code.	CD11
10	It could help users to keep track of displayed code panels if the code panels align with the direction of their gaze. Similarly, it could be helpful if users could pin selected code panels into their field of view.	CD13

is actually the one that we described in Section II. In the earlier prototype, a user could only close a single code panel at a time. So to close a complete sub-tree required many interactions and thus a poor user experience. Thus, we implemented the possibility to close complete (sub)-trees of code panels with one action. Furthermore, we identified a lack of information content on visual links. Therefore, we added a color-coding to each visual link, depending on whether the link leads to a class (blue link) or to a method (green link).

2) Formative User study: In addition to identifying more usability issues, our user study focused on evaluating the general usability our tool and potential advantages or disadvantages of the approach. A total of 8 computer science students participated in this study. Each participant had previously completed our Advanced Programming course and thus has appropriate programming skills. No one had used a HoloLens 2 before, so this kind of HMD was new to all of them. In each evaluation run, two participants took part simultaneously as a team. The task of each team was to identify and fix three bugs in the given software, an object-oriented implementation of Super Mario bros. implemented in Java [7]. The third bug was mainly meant as a backup, in case the participants fixed the first two bugs in less then 60 minutes.

Before the actual experiment, all participants completed a two-phase tutorial to become familiar with the use of the HoloLens 2 in general and the usage of our tool in particular. First, they completed the interactive tutorial that comes with the HoloLens 2, learning all the controls and gestures provided by the HMD. Second, they worked through an interactive tutorial designed in the same style, but tailored to the use of our tool. At the beginning of each run, both participants got a description of the current bug. Each bug considered on its own was not hard to fix, but it was not trivial to locate the defect in the code. So navigating to that location in the code was one major challenge. Although we encouraged the teams to fix all three bugs, it was not essential for the study that they successfully completed all three tasks, but we were more interested in their experience and that they actually used the functionalities provided by the tool. All bugs could be reproduced by playing the game for a few seconds. The first bug prevents enemies from colliding with the world, the second bug leads to the possibility of unlimited jumping of Mario, and the third bug left the points counter unchanged when coins were collected. Each team worked in a pair programming setting [10] with a driver and a navigator role. The driver is the person who is actually writing and changing code. In our setting, the driver wears the HoloLens 2 and works with our IDEVELOPAR tool. The other participant, the navigator, observes and corrects the work of the driver and suggests strategies on how to solve the given task. The navigator sits at a desktop computer with two screen. One screen shows the IntelliJ IDE with our plugin, while the second screen shows a live stream of the driver's sight through the HoloLens 2. Both participants were encouraged to communicate and think aloud as much as possible [11]. We recorded the HoloLens 2 display during the experiment, capturing every interaction performed

by the driver together with all holograms in the room. We used the HoloLens 2 audio input to record the communication among the team members. Additionally, we created logs of every action performed by the driver with our tool. When a team started to work on the next bug, they changed the driver and navigator roles. At the end, the participants were asked to fill in a short questionnaire containing a System Usability Scale (SUS) part [12], [13], a User Experience Questionnaire (UEQ) [14], and some rating questions about the usability of particular features our the tool. Finally, we conducted a short interview, which was also recorded.

After the first run, we found that the participants of the first team had severe problems opening new code panels via the hand menu. It turned out that during the development of the tool, we had got accustomed to a workaround such that we we were no longer aware that it initially was a workaround. Thus, we decided to fix this problem first, as it almost made the tool unusable. The other three teams all used this revised prototype.

V. Results

A. Quantitative Analysis

For the quantitative analysis, we used the data of the SUS, the UEQ and the feature rating. Due to the low number of participants, we did not perform any statistical tests but only provide descriptive statistics in form of mean values in Figure 4 and Figure 5. In these barcharts the 95% confidence intervals are also shown to indicate the degree of uncertainty of the result. Both the classical SUS score 72.19 (with 95% confidence interval [52.72 - 85.00]) as well as improved SUS score for small samples [15] 69.45 indicate that the participants perceived the overall usability of the prototype as acceptable (in the OK-to-good range).

Figure 4 shows the results of the perceived usability and usefulness of the features of the tool. We used a 6-point scale from *1=Very hard to use* to *6=Very easy to use*. The participants could also indicate that they didn't use a feature.

Almost every feature of IDEVELOPAR is generally rated as useful, with values between five and six. Only the usefulness of *resizing a code panel* and *semantic zooming* was rated slightly below average.

In general, the usability was primarily positive. All but one values lie at four or above. However, we could identify some features with usability problems. Code editing (F6) was rated as highly useful but only reached an average usability rating. The same applies to other features like opening new code panels or scrolling the code. Thus, inconsistent ratings of a feature (F2, F9-F11) suggest that it should be improved, but do not indicate what the concrete problems are. To gain more insight about these cases, we actually looked at the ratings of each team before the interviews, and asked them to comment on features which they rated with high usefulness, but lower usability. One participant mentioned, "You must become familiar with the tool first until you can operate these functions properly." So it is definitely plausible that the usability ratings may raise after a longer period of use.

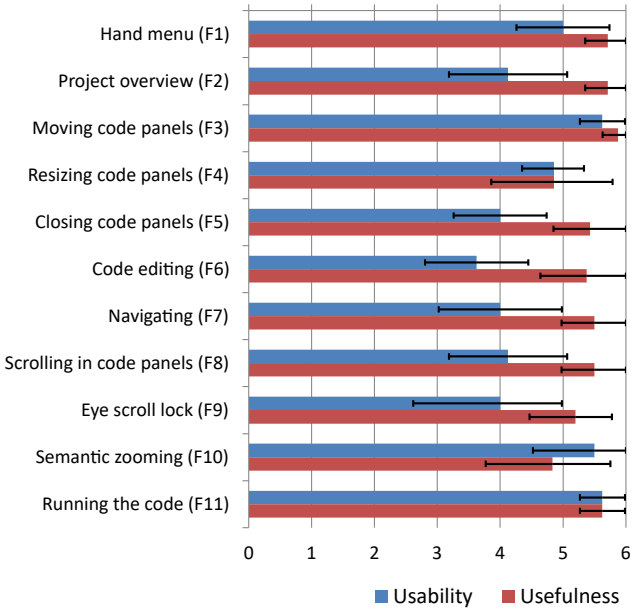

Fig. 4. Average ratings per feature regarding usability and usefulness. The error bars show the 95% confidence interval.

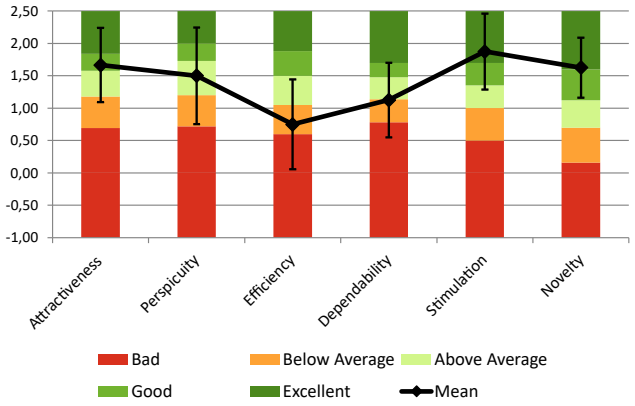

Fig. 5. Results of the User Experience Questionnaire. The error bars show the 95% confidence interval.

Figure 5 shows the results of the UEQ. Compared with the benchmark, our tool classifies above average except for the efficiency category. However, because of the small number of participants, it is also difficult to make statistically reliable statements here. Still, we could observe that the results coincide with the participants' comments. Regarding the efficiency aspect, a few participants perceived the tool as not as efficient as their normal IDE: "In a normal IDE, the errors would have been found more efficiently." and for some, it was too slow to operate: "So some things eat up too much time." On the other hand, the good values of the perspicuity category are reflected in the user comments. One participant stated, "I had more of an overview now of where I am, what I need to do right now."

B. Qualitative Analysis

For our analysis we first transcribed the recorded interviews and used a mix of open coding and theoretical (for the features) coding to identify interesting phenomena in the data. As of yet, we did not analyze the video and log data. In the following we use the abbreviation GiPj to refer to participant j of group i.

a) Features: For several features, no problems were reported at all. For example, no group experienced problems with *semantic zooming* (F10), "this minimized view was really good" (G1P2) was the general opinion across all groups. Additionally, *moving and placing code panels* (F3) was described as a good usable feature.

On the other hand, for some of the features with inconsistent ratings, we found very concrete descriptions of usability problems. With respect to *code editing* (F6), it was mentioned several times, that it was cumbersome to place the cursor at the desired location. Although it was introduced to the participants, that they can position the cursor by holding Ctrl on the keyboard while looking at the corresponding code location, most of them had problems to use it that way: "What also didn't work so well is clicking into the code that you are actually in the line to edit. I tried that for a while, but couldn't get it to work" (G1P1). Another participant stated, that it was unusual to move the cursor with the eyes and that based on previous experience the Ctrl shortcut is associated with a different kind of functionality: "With Ctrl, I am just used to jump over words in normal IDEs" (G3P1).

The feature *eye-scroll lock* (F9) was almost not used during the evaluation. But one participant had the problem to press the corresponding button, located at the bottom of each code panel, because the code scrolled unintentionally while moving the eyes to focus the button: "You have to look at the button somehow to press it, but during this time it keeps scrolling" (G2P2).

The participants had very opposing opinions about *scrolling in code panels* (F8) using eye-tracking. One said: "I find scrolling with my eyes extremely exhausting" (G1P2), but in contrast, another participant mentioned: "The scrolling over the code, I found that actually very simple" (G3P2). In addition, the participant (G1P2) has noted, "while scrolling, you are looking for something in the code and can't scroll down and read at the same time", but this is actually a misconception of how eye-tracking based scrolling works. When reading code in a code panel, the text will automatically scroll down adapted to the reading speed. During the evaluation, several participants considered scrolling and reading as two completely separate tasks, which leads to unnecessary complexity. As a result, these participants may find scrolling more tiring than those who understand scrolling and reading as one task, leading to these opposing opinions.

b) Code-Panel Tree: Using IDEVELOPAR, the participants get a better overview of the project. They could build a graph representation of the code fragments of interest and navigate to their desired code locations. During the interview, we got overall positive feedback regarding the question if the tool improves code comprehension. One participant answered: "I found it significantly easier to understand the program that way, through all the graphs and such, than if I had just seen it on the computer" (G1P2). Further, the participant mentioned that "this [the overview] is much better here through this graph. I see I started here, then I went that way, and now I got there via these three detours because one method always calls the next. And that's not so easy with normal windows" (G1P2).

c) Learning curve: A learnability effect was observable within the last group, which found all three bugs. In their last run, they used the tool very effectively and improved their approach with the experiences gained from the previous tasks. "I have now noticed the biggest advantage on the third try. I now had much more overview of the code. I had one code panel for the method and one for the class" (G4P2). This feedback supports our hypothesis that a significant learning effect occurs with prolonged use, which could increase the effectiveness of the tool over time. But in general, we could observe a wide range of different learning curves. Some got along quickly, and others took a little longer to operate the tool properly. But in the end, we often got the feedback that "it was like you would kind of expect it to be" (G4P1), which is an essential aspect of an intuitive, usable tool.

d) Strategies: Furthermore, the participants described different approaches how they used the tool to solve the tasks. They often followed the same approach as in a classical IDE, but with increasing time of use, most participants developed some placement strategies. These range from more enhanced code view placements to usage of the surrounding physical space: "For example, you can go to the corner now, say I'm going to open the class here, and a few methods that I need and maybe go somewhere else and open a few more classes there" (G4P2) towards unexpected strategies. One participant used a office swivel chair to build a 360-degree workspace to place the code views all around him.

e) Suggested extensions: Besides all this substantial feedback, the participants missed some advanced features they knew from their IDE. For example, they missed a visual cue if a class, method, or variable is not used somewhere in the code. Furthermore, they missed some basic error highlighting in the code panels. They additionally had some problems regarding the efficiency of controlling the tool. Features do not work on the first try due to a mix of a faulty implementation of the software and some technical tracking problems of the AR glasses. Some participants have used wrong gestures, leading to incorrect tracking and, therefore, to a poor user experience.

f) Beyond usability: Walking opens the free flow of ideas and is a simple method to boost creative thinking [16]. As one participant put it "if you sit, then I feel, it is harder to think, as if you walk around a bit."

VI. Related Work

Researchers introduced many approaches and visualizations to support developers in gaining a better understanding of their source code and facilitate the overall programming process.

The survey by Sulir et al. [17] on visual augmentation of source-code editors includes more than 100 approaches that were published between 2002 and 2017. Many more have been published since.

A. Information Needs and Diagrams

In general, it is hard for the human working memory to keep track of all the related programming tasks: Navigate through code, remember already visited code fragments, keep an overview of call hierarchies, and much more. Therefore developers tend to create visual representations of the current software project, often in the form of various diagrams or sketches. These kinds of visual representations of software systems are promising for supporting developers during their work [18]. In an empirical study Lee et al. [19] investigated how to support developers with diagramming tools. In a first step, the authors asked the participants what a diagram should show to a developer. The most desired information mentioned by the participants are: "Who calls a method/call hierarchy", "Who uses/references who", "The paths navigated through methods" and "Type hierarchy". In general, it turns out that such diagrams help to find a way around the software. One participant said: "It would be useful to develop a cognitive map of the software, and it would help to navigate relationships". While this quote shows the possibilities of such visualizations, participants also mentioned that the available screen size is not sufficient for showing diagrams and that it is hard to display all desired information.

Fleming et al. [20] applied Information Foraging Theory to investigate how software developers use tools to perform tasks like debugging, refactoring and code reuse. Here, programmers are seen as information predators who gather information (prey) using evolutionary foraging mechanisms to reduce energy. In particular, predators need to predict how much useful prey they will gather on a path. In an empirical study with professional developers Piorkowski et al. [21] found that over 50% of developers' navigation choices produced less value than they had predicted and nearly 40% cost more than they had predicted.

B. Novel User Interfaces for Programming

Code Canvas [2] was proposed as an alternative to bento-box design of existing development environments. In Code Canvas the files of a software project are placed on a possibly infinite canvas, such that developers can create their own software map to better exploit their spatial memory.

Code Bubbles [4], [5], that we briefly described in the introduction, provides a working set-based interface for IDEs (especially for Eclipse). It allows a user to create side-by-side code views, displaying not necessarily file-based data but making it possible to show only the essential fragments out of a method or class. When navigating through the code, new views will open automatically when following, e.g., method calls. Connecting edges between the views visually highlights all the emerging call hierarchies and thus gives an structured overview of the navigation history. In subsequent work, the Code Bubbles approach was reused to create a user interface for debugging [22]. Patchworks [23] and its successor CodeRibbon [24] provide a ribbon-based interface. A ribbon is basically a canvas that is only infinite in horizontal direction where visual elements, here document editors, can be placed on a grid.

C. AR for Software Engineering

In a vision paper [25] Merino et al. discussed the potential of AR in the context of software development with respect to several general aspects like collaboration, communication, embodiment, mediated reality, mobility, multi-device und pervasiveness. Currently, there exist only few papers in the software engineering community (and many of those are short papers) on the use of AR in software engineering. Most papers use AR to place existing 3D software visualizations (e.g. city metaphor [26], [27] or 3D tree visualizations [28] in the physical space and use these visualization to analyze software architecture [29]–[31], software performance [26], [32] or project management [32], [33].

VII. Conclusion

We have presented IDEVELOPAR, a novel AR-based user interface to enhance code understanding. By allowing the user to place code panels freely in the physical space around them they are no longer limited by the display size nor are they bound to sit next to their desktop computer. While the main goal of our formative study was to gain insights on how to improve the overall usability of our tool, we also found that the participants felt that the tool improved code comprehension compared to their classical IDE. Although the statistical analysis has to be considered with caution, it is in line with the results of the qualitative study that a user can get a better overview with the help of the tool and that for some of the features with inconsistent rating the participants mentioned concrete usability problems.

The main goal of the work presented in this paper was to develop linked code views in AR and improve their usability. As a next step we intend to leverage AR to enable novel usage scenarios where code views are linked to objects in the real world. As part of our future work, we also want to extend the quantitative analysis by performing additional user sessions. Furthermore, we will further improve the user-interface based on the results of our study and add more features.

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, dec 2006.

[2] R. DeLine and K. Rowan, "Code canvas: zooming towards better development environments," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 207–210.

[3] B. de Alwis and G. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 51–54.

[4] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, 2010, pp. 455–464.

[5] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 2503–2512. [Online]. Available: https://doi.org/10.1145/1753326.1753706

[6] A. Tang, C. Owen, F. Biocca, and W. Mou, *Performance Evaluation of Augmented Reality for Directed Assembly*. London: Springer London, 2004, pp. 311–331. [Online]. Available: https://doi.org/10.1007/978-1-4471-3873-01_6

[7] Çandıroğlu. Ahmet, S. Unas, and B. Umut, "Classic Super Mario Bros. game implemented with Java," https://github.com/ahmetcandiroglu/Super-Mario-Bros, 2022.

[8] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X96900099

[9] A. Blackwell and T. Green, "Notational systems–the cognitive dimensions of notations framework," *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann*, vol. 234, 2003.

[10] L. A. Williams, "Pair programming." *Encyclopedia of software engineering*, vol. 2, 2010.

[11] M. Van Someren, Y. F. Barnard, and J. Sandberg, "The think aloud method: a practical approach to modelling cognitive," *London: AcademicPress*, vol. 11, 1994.

[12] J. Brooke, "Sus: a retrospective," *Journal of usability studies*, vol. 8, no. 2, pp. 29–40, 2013.

[13] ——, "SUS: a "quick and dirty'usability," *Usability evaluation in industry*, vol. 189, 1996.

[14] B. Laugwitz, T. Held, and M. Schrepp, "Construction and evaluation of a user experience questionnaire," in *Symposium of the Austrian HCI and usability engineering group*. Springer, 2008.

[15] N. Clark, M. Dabkowski, P. J. Driscoll, D. Kennedy, I. Kloo, and H. Shi, "Empirical decision rules for improving the uncertainty reporting of small sample system usability scale scores," *International Journal of Human–Computer Interaction*, vol. 37, no. 13, pp. 1191–1206, 2021.

[16] M. Oppezzo and D. Schwartz, "Give your ideas some legs: The positive effect of walking on creative thinking," *Journal of experimental psychology. Learning, memory, and cognition*, vol. 40, 04 2014.

[17] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubän, "Visual augmentation of source code editors: A systematic mapping study," *Journal of Visual Languages & Computing*, vol. 49, pp. 46–59, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X18301861

[18] S. Baltes and S. Diehl, "Sketches and diagrams in practice," in *FSE 2014*, 2014.

[19] S. Lee, G. C. Murphy, T. Fritz, and M. Allen, "How can diagramming tools help support programming activities?" in *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, pp. 246–249.

[20] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. M. Burnett, R. K. E. Bellamy, J. Lawrance, and I. Kwan, "An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, 2013. [Online]. Available: https://doi.org/10.1145/2430545.2430551

[21] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett, "Foraging and navigations, fundamentally: Developers' predictions of value and cost," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 97–108. [Online]. Available: https://doi.org/10.1145/2950290.2950302

[22] A. Bragdon, K. Rowan, J. Jacobsen, R. DeLine, and R. DeLIne, "Debugger canvas: Industrial experience with the code bubbles paradigm," in *International Conference on Software Engineering*, June 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/debugger-canvas-industrial-experience-with-the-code-bubbles-paradigm/

[23] A. Z. Henley and S. D. Fleming, "The patchworks code editor: Toward faster navigation with less code arranging and fewer navigation mistakes," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 2511–2520. [Online]. Available: https://doi.org/10.1145/2556288.2557073

[24] B. P. Klein and A. Z. Henley, "Coderibbon: More efficient workspace management and navigation for mainstream development environments," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 604–608.

[25] L. Merino, M. Lungu, and C. Seidl, "Unleashing the potentials of immersive augmented reality for software engineering," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou, Eds. IEEE, 2020. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054812

[26] L. Merino, M. Hess, A. Bergel, O. Nierstrasz, and D. Weiskopf, "PerfVis: Pervasive visualization in immersive augmented reality for performance awareness," in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019*, V. Apte, A. D. Marco, M. Litoiu, and J. Merseguer, Eds. ACM, 2019. [Online]. Available: https://doi.org/10.1145/3302541.3313104

[27] D. Baum, S. Bechert, U. W. Eisenecker, I. Meichsner, and R. Müller, "Identifying usability issues of software analytics applications in immersive augmented reality," in *Working Conference on Software Visualization, VISSOFT 2020, Adelaide, Australia*. IEEE, 2020. [Online]. Available: https://doi.org/10.1109/VISSOFT51673.2020.00015

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

[29] R. Mehra, V. S. Sharma, V. Kaulgud, and S. Podder, "XRaSE: Towards virtually tangible software using augmented reality," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA*. IEEE, 2019. [Online]. Available: https://doi.org/10.1109/ASE.2019.00135

[30] R. Mehra, V. S. Sharma, V. Kaulgud, S. Podder, and A. P. Burden, "Towards immersive comprehension of software systems using augmented reality - an empirical evaluation," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia*. IEEE, 2020. [Online]. Available: https://doi.org/10.1145/3324884.3418907

[31] C. S. C. Rodrigues, C. M. L. Werner, and L. Landau, "VisAr3D: an innovative 3D visualization of UML models," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2889160.2889199

[32] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, "Synchrovis: 3D visualization of monitoring traces in the city metaphor for analyzing concurrency," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT), Eindhoven, The Netherlands*, A. Telea, A. Kerren, and A. Marcus, Eds. IEEE Computer Society, 2013. [Online]. Available: https://doi.org/10.1109/VISSOFT.2013.6650520

[33] V. S. Sharma, R. Mehra, V. Kaulgud, and S. Podder, "An extended reality approach for creating immersive software project workspaces," in *Proceedings of the 12th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2019, Montréal, QC, Canada*, Y. Dittrich, F. Fagerholm, R. Hoda, D. Socha, and I. Steinmacher, Eds. IEEE / ACM, 2019. [Online]. Available: https://doi.org/10.1109/CHASE.2019.00013