Preprint. Final version published in Proceedings of the 31st International Conference on Software Maintenance and Evolution ICSME (ERA Track), Bremen, Sept. 29- Oct. 1, 2015. https://doi.org/10.1109/ICSM.2015.7332518].

# Live Object Exploration: Observing and Manipulating Behavior and State of Java Objects

Benjamin Biegel, Benedikt Lesch, and Stephan Diehl
Department of Computer Science
University of Trier
Trier, Germany
Email: {biegel,diehl}@uni-trier.de

***Abstract*—In this paper we introduce a visual representation of Java objects that can be used for observing and manipulating behavior and state of currently developed classes. It runs separately, e.g., on a tablet, beside an integrated development environment. Within the visualization, developers are able to arbitrarily change the object state, then invoke any method with custom parameters and observe how the object state changes. When changing the source code of the related class, the visualization holds the previous object state and adapts the new behavior defined by the underlying source code. This instantly enables developers to observe functionalities objects of a certain class have and how they manipulate their state, and especially, how source code changes influence their behavior. We implemented a first prototype as a touch-enabled web application that is connected to a conventional integrated development environment. In order to gain first practical insights, we evaluated our approach in a pilot user study.**

## I. Introduction

Current tools and research methods introduce many techniques that help developers to comprehend and manage their source code [1]. When syntax or compiler errors occur, for example, modern integrated development environments provide built-in features that immediately notify developers. What is more, syntax highlighting, displaying structural context information, automatic source code formatting, links for source code navigation, specialized visualizations like UML [2] and other features support understanding the overall structure of a software project. Nevertheless, the behavior (or semantics) of a certain code fragment is not observable. While creating or changing source code, developers have to put themselves into the role of a computer [3]. As for that, they are forced to execute code fragments mentally and guess how a code change would affect state and behavior of the program at runtime [4]. This issue was nicely summarized by Chris Granger, who is mainly known for developing an interactive open-source IDE named 'Light Table'[1]:

> *"We are quite literally throwing darts in the dark and praying that we at least hit the board. We simply cannot see what our programs do and that's a huge problem whether you're just starting out or have written millions of lines of beautiful code."*
>
> —Chris Granger, 2014[2]

[1]http://lighttable.com/
[2]http://www.chris-granger.com/2014/03/27/toward-a-better-programming/

It is not surprising that developers try to avoid comprehending the program behavior by solely reading the source code. In about 40% of the time spent for understanding a certain code fragment, developers execute the program and investigate the code rational at runtime [5]. This includes a frequent use of debuggers [6], testing the program behavior by using the user interface as end-users will do [7], or simply using loggers or console outputs. Nevertheless, developers are forced to permanently switch between the structural and behavioral view on a program, which still is tedious and cognitively demanding. Furthermore, since the program state is only changed temporarily, for subsequent executions, all changes to the program state must be re-applied manually.

A promising paradigm that addresses the previously mentioned issues is *live programming* [8]–[10]. In general, live programming environments provide an infrastructure, that enables developers to immediately recognize how state and behavior of a running program are affected, when changing the underlying source code [11], [12]. Live programming occurs in several application scenarios, e.g., creating user interfaces [13], in continuous testing [14], or when using an in-situ debugger [15]. Recent studies show that live programming significantly helps developers in finding and fixing bugs [16], [17]. In particular, since live programming narrows the gap between editing and debugging, developers become aware of newly introduced bugs more early. Nevertheless, when using object-oriented languages like Java, current live programming approaches have some limitations. Some only allow to execute the entire program, which makes it hard to investigate smaller parts in isolation. Others use a fine-grained view on runtime data by putting it directly into or beside source code, e.g., line by line, which makes it hard to keep track of bigger parts of the program. Beside program and source code level, however, object-oriented languages also require observing on the object level, which is not explicitly supported by current live programming approaches.

This paper fills this gap by introducing an additional visualization representing Java objects. Such objects are instantiated directly from classes developed within an IDE and can be explored by changing their states and invoking their methods. Changes in the underlying class will be injected into the "living objects" and can immediately be observed in the visualization.

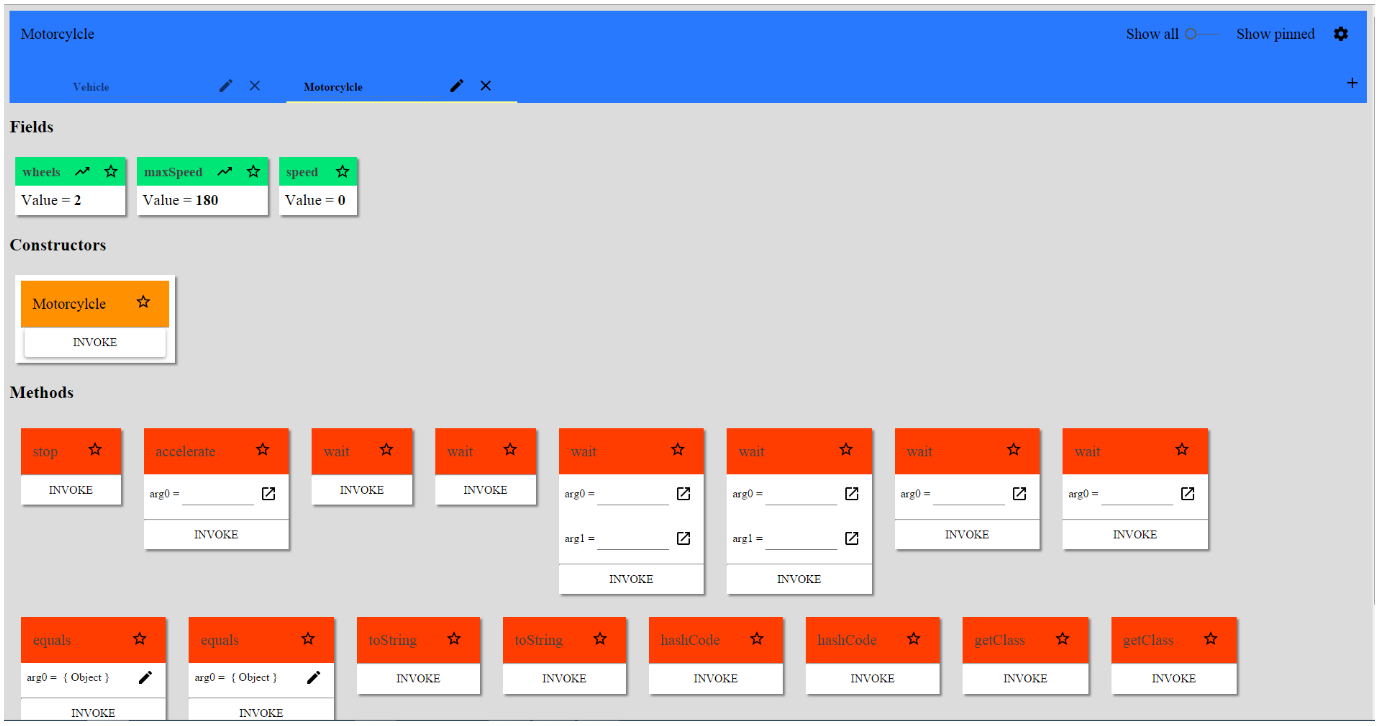

Fig. 1. A Screenshot from our prototype implementation displaying a living object of a class named "Motorcycle".

## II. Main Concept and Prototype Implementation

The main idea of this paper is to bring live execution feedback to the development of Java classes. An advantage of live programming is that effects on the program execution can be observed continuously while changing the source code in the editor. As for that, we propose a visualization that (1) is permanently connected to an IDE, (2) represents instances of currently developed classes, (3) ensures updating object behavior that comes with source code changes while keeping the current object state, and (4) enables to observe and manipulate behavior and state of currently represented objects within the visualization.

### A. Live Object Exploration

As can be seen in Figure 1, the entire view is used to visualize an object. Its fields (green), constructors (orange) and methods (red) are represented as floating boxes. These boxes can be rearranged by simple drag and drop gestures with mouse or touch input. In the toolbar at the top, there are tabs for switching between currently living objects. In order to distinguish between multiple objects of the same class, it is possible to rename the tabs. The + symbol on the right allows to create new instances of classes from the currently opened project within the IDE.

*1) Filtering:* Because all members of an object, including the inherited members, are shown per default, representing larger objects can be very complex. To solve this problem, we use two different filtering mechanisms. First, by clicking on the cog symbol in the upper right of the toolbar, a menu opens with settings for showing and hiding several types of members, like inherited members or constructors. Another way for filtering specific members is pinning. By clicking on the star symbol in the upper right corner of any box, the related box can be pinned or unpinned. Changing the mode from "Show all" to "Show pinned" (right side of the toolbar) hides all unpinned boxes and shows only the selected ones.

*2) Fields:* In the head of a field box (green part) the name of the field is displayed. In this first concept, we refrain from displaying type information to keep the layout more tidy. The value of the field is displayed in the lower white area. Depending on the type, the value is represented differently, e.g., a toggle button is used for boolean values, whereas for string values an input field is used. Arrays and objects are not displayed directly on the main screen. They are only indicated by buttons that lead to a special dialog showing their array elements and members respectively. If any number is saved in a field, developers can chose between a regular representation, that is only displaying the current value, and a line chart representation, that makes it possible to observe how the underlying value changed over time. With the button next to the star symbol one can toggle between both representations.

*3) Changing Values:* The object state can be changed directly within the visualization by entering new field values. In order to support an explorative interaction style, we also provide, besides entering new values solely with the keyboard, touch-optimized visual components like sliders and buttons to change the values interactively. Furthermore, instead of typing a new value, the developer is also able to choose from a list

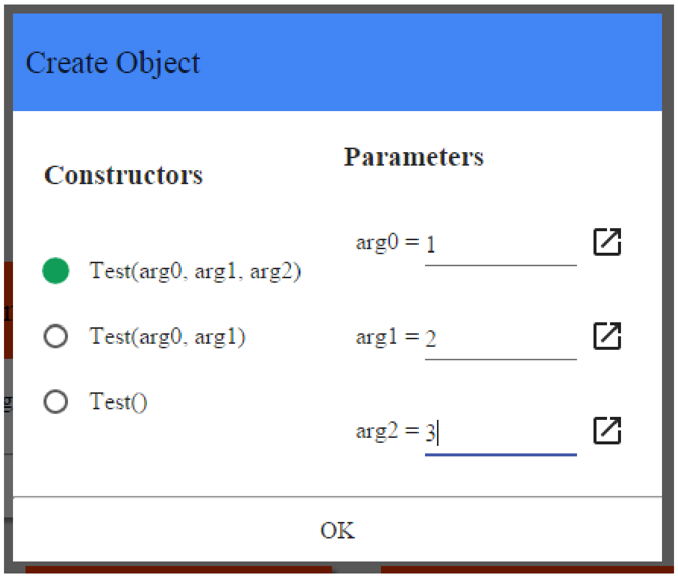

Fig. 2. A dialog for creating a new object.

of predefined values. Field boxes that contain changed values are temporarily highlighted by changing its color.

*4) Method Invocation:* In order to explore the behavior of a living object, all its methods can be invoked. Parameter values can be entered as described in Section II-A2. If a field value is changed after invoking a method, the field box is highlighted temporarily. Furthermore, if only pinned boxes are shown and a hidden box was affected by the invocation, it is then automatically highlighted for a certain time. By invoking constructors, a new instance will be created and the current living object will be replaced.

*5) Creating Arrays and Objects:* Since primitive values can be entered directly, arrays and objects must be created in special dialogs. In the array dialog, elements of an array can be added and removed. Developers are also able to navigate through multi-dimensional arrays. In order to keep track of inner array elements, they can also be pinned, and are then be shown on the main screen. In contrast to arrays, objects can only be instantiated by invoking one of its constructors. Figure 2 shows a dialog for creating objects. On the left side, developers can chose the preferred constructor. Then, on the right side, they have to enter values for its parameters. If a parameter requires an object or array, another dialog can be opened in order to create one of these.

## B. Prototype Implementation and Infrastructure

For our prototype implementation we strongly rely on state-of-the-art web technologies like HTML5, JavaScript, Polymer[3], and web sockets. This enables us to independently run our prototype visualization on diverse devices, e.g., on a tablet used with touch gestures as well as in a classical desktop environment with mouse and keyboard. Inspired by CodePad [18] and our previous work [19] we suggest using a conventional desktop environment with an additional touch device. Furthermore, we wrote a plug-in for the IntelliJ IDE[4] that handles the communication between IDE and web application. By using the reflection API of Java, this plug-in instantiates new objects, manipulates their state and invokes their methods. After changing the source code, a new object is being created based on previously chosen parameter values for the constructor. Then, based on the saved object state in the web application, the newly created object in the IDE will be updated immediately. It is worth mentioning that living objects in the web application and objects within the IDE are only loosely connected. That means, we synchronize structure and state manually. In contrast to standard Java techniques like serializing objects, this enables us to also change the structure of a living object without losing its state.

## C. Application Scenarios

While elaborating our approach, we discussed several application scenarios. A classical live programming scenario is using the tool next to an IDE by a single developer. But also a pair programming setting is conceivable, as we have seen in our study. The driver could concentrate on writing code and the navigator could use our tool for testing the expected behavior. Since our prototype was implemented with web technologies, this approach can also be applied in teaching. While the lecturer presents source code of a Java class, for example, students could use our tool simultaneously on tablets to explore its structure and behavior.

## III. Evaluation

In order to gain first practical insights, we asked 9 developers to participate in a think-aloud user study. Each session lasted about 45 minutes in a pair programming setup (one session with three participants, other three in pairs). During the study, we observed the sessions and took notes. After the study, we performed a semi-structured interview and asked each participant to fill in a questionnaire.

## A. Questionnaire

Before filling in the questionnaire, we informed the participants that their answers will be evaluated and published anonymously. In the first part of the questionnaire, the participants were asked to provide some demographic data, their prior knowledge and skills. The rest of the questionnaire is subdivided in three parts, addressing the proposed approach in general, and the used visualization and interaction in particular. In order to assess our approach, we provided both free-text forms (5 in total) as well as statements (25 in total), that can be rated by a 5-point Likert scale ranging from ‘Disagree’ (rating 0) to ‘Agree’ (rating 4). The scale value ‘2’ refers to a neutral position.

[3]http://www.polymer-project.org

[4]http://www.jetbrains.com/idea/

### B. Participants

The participants of the study were 9 male developers, three master and two PhD students from the University of Trier, Germany, three bachelor students from the HTW Saarbrücken, Germany, and one professional software developer working in the industry. They were between 23 and 31 years old. All participants rated themselves as 'intermediates' in Java. In object-oriented programming, two see themselves as 'experts' (rest as 'intermediates'), and two specify only an 'average' experience in debugging, whereas the rest are 'intermediates'. In summary, all participants are quite experienced in Java, object-oriented programming, and debugging. Experiences in visualizations and JUnit are more diverse. 4 participants had only little experience with program or object visualizations (2 'average', 2 'advanced'). The same distribution also apply for the experience with JUnit.

### C. Experimental Design

The participants had to solve ten exercises by using our prototype implementation. We used a classical desktop environment running IntelliJ. Next to this, we used an additional touch-monitor running our visualization connected to the IDE. The overall study was divided into three parts, focusing on (1) visualization, (2) interaction, (3) integration. In the first two parts, we evaluated the external view without using the IDE only. In the third part, the participants used both IDE and external view to investigate how our approach integrates in their usual work flow.

### D. Results

The overall impression was quite positive. Although we did not introduced our prototype to the participants on purpose, no additional help was needed or requested. Since our observations suggest that using our tool was intuitive and easy, one participant slightly disagreed (rating 1), whereas 6 participants (rating 3 or 4) confirmed our observations and 2 took a neutral position (rating 2). Most participants, however, agreed that using our prototype was beneficial. Also the following statements were mostly rated by 3 or 4, whereas 2 ('neutral') was the worst rating. They confirmed that our approach offered sufficient support for object exploration, helped in understanding the related class structure, supported in exploring the functional behavior of the object, and made the object state easy to explore. 6 participants said they would use our prototype more often in the future.

*1) Visualization:* In contrast to the general impression, the opinions concerning our visualization, however, diverged. 8 participants (rating 3 or 4) found the visualization well structured and clear, one participant had a 'neutral' position and another partly disagreed (rating 1). In order to get an even clearer structure, half of the participants suggested to use a table layout instead of floating boxes. Moreover, in order to get a better understanding of the structure of an object, some participants also requested displaying type information of fields and parameters. Since 7 participants still thought that the visualization of primitives and strings was clear, some participants had problems with the proposed object and array visualizations. One participant was not completely convinced (rating 1) that objects were visualized in a good way. 2 participants partly disagreed that arrays were easy to use. Instead of using modal dialogs, most participants wanted to expand objects and arrays directly on the main screen, e.g., as a tree map. Furthermore, modal dialogs made it difficult to keep the current context. Nevertheless, 8 participants (rating 3 or 4) confirmed that changes in the object state were very well highlighted. Still 7 participants thought that the line charts helped them to keep track of those changes. Remarks were missing labels of the axis, access to the complete history in a scrollable graph, and fixing several bugs, like vanishing values. In general, the line chart view was preferred to the regular view.

*2) Interaction:* In general, all participant found that method invocation was easy and helped them to understand the functional behavior. Most noted that they got no visual feedback if an method invocation caused no change in the object state or if an exception was thrown. As setting up primitive or string values was consistently considered as an easy task, only one participant found it challenging to create an array and an entire object. Using predefined values were only questionable for one participant (rating 1), whereas 6 participants found this feature sufficient for exploration purposes. Half of them said that the given examples were not very useful and seemed to be randomly chosen. All participants thought that the pin mode helped well in focusing on a specific set of elements. But they suggested to expand this feature, e.g., to allow hiding elements, or pin all children of an object. Another helpful filtering feature would be a text search for elements.

*3) Integration:* Using our visualization together with the IDE was considered differently. Only 4 participants thought that this combination supported them in the development process, and two participants were skeptical (rating 1) about it. Nevertheless, many of them mentioned that they liked to quickly test new implemented or changed functionality, without executing the program or starting the debugger. Using an additional touch-enabled device for displaying the visualization was also considered very differently. Only 3 participants (rating 3 or 4) liked the idea of a touch input, whereas 3 took a neutral position, one was doubtful (rating 1), and two gave no rating at all. One half of the participants preferred using a second monitor, while the other half thought that touch interaction fits more in an exploration scenario like debugging.

## IV. Related Work

Besides the related work in the field of live programming already mentioned in the introduction, there are several other approaches, that visualize the program state at runtime. Algorithm animation systems like Jeliot [20] or LIVE [21] provide automatically generated animations for visualizing the execution of an application step-by-step. In contrast to our approach, they are strongly focused on source code lines instead of a more abstract view on object level. Jive [22] is a visual debugger that visualizes the runtime state, displayed as an UML object diagram [2], and the call history of an application, displayed as an UML sequence diagram [2]. A similar strategy was used in jGRASP [23], but instead of using general visualizations, they use specialized dynamic object viewers that consider the underlying data structure of an object. Since both approaches offer more detailed visualizations, they can only visualize data provided by a debugger, thus they are not able to provide live execution feedback as our approach does. With BlueJ [24] it is possible to invoke code without starting the program manually. The result can then be inspected or used for subsequent invocations. BlueJ also allows to create class instances that can be explored. In contrast to our approach, after changing the underlying code, interactively created objects are dropped. Thus, BlueJ does not support live programming. Finally, all above mentioned approaches have integrated visualizations that cannot be run on other devices separated from the IDE . That is why they are less suitable for the application scenarios described in Section II-C.

## V. Conclusion

In this paper, we introduced *live object exploration*, a live programming approach for representing instances of currently developed Java classes. The visualization enables developers to interactively observe and manipulate behavior and state of such "living objects". After changing the underlying source code, the behavior and structure of living objects also change, while their current states still remain.

The user study shows that the approach does work in general and helps to understand structure and behavior of Java objects, especially, to observe impacts of source code changes on runtime. The study results, gathered from questionnaires, interviews and observations, also provide important information for further improvements. More specialized visualizations, for example, were requested for different data structures as used in previous work (cf. Section IV). Partly it was tedious to navigate through highly nested data structures, especially, because the current visualization also uses modal dialogs that sometimes distract attention from the main context. Another drawback is creating new instances of complex classes or arrays. Some participants suggest to provide also techniques for defining objects and arrays within the IDE.

## References

[1] M. D. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[2] G. Booch, J. E. Rumbaugh, and I. Jacobson, *The unified modeling language user guide - the ultimate tutorial to the UML from the original designers*, ser. Addison-Wesley object technology series. Addison-Wesley-Longman, 1999.

[3] H. Lieberman and C. Fry, "Bridging the gulf between code and behavior in programming," in *Human Factors in Computing Systems, CHI '95 Conference Proceedings, Denver, Colorado, USA, May 7-11*, 1995, pp. 480–486.

[4] J. L. Snell, "Ahead-of-time debugging, or programming not in the dark," in *Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering]*. IEEE, 1997, pp. 288–293.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28*, 2006, pp. 492–501.

[6] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.

[7] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 255–265.

[8] S. L. Tanimoto, "VIVA: A visual language for image processing," *J. Vis. Lang. Comput.*, vol. 1, no. 2, pp. 127–139, 1990.

[9] C. M. Hancock, "Real-time programming and the big ideas of computational literacy," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[10] S. McDirmid, "Living it up with a live programming language," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, October 21-25, Montreal, Quebec, Canada*, 2007, pp. 623–638.

[11] B. Victor, "Inventing on principle," Invited talk at the Canadian University Software Engineering Conference (CUSEC), January 2012.

[12] S. L. Tanimoto, "A perspective on the evolution of live programming," in *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19*, 2013, pp. 31–34.

[13] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, "It's alive! continuous feedback in UI programming," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19*, 2013, pp. 95–104.

[14] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14*, 2004, pp. 76–85.

[15] S. McDirmid, "Usable live programming," in *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31*, 2013, pp. 53–62.

[16] J. Krämer, J. Kurz, T. Karrer, and J. O. Borchers, "How live coding affects developers' coding behavior," in *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2014, Melbourne, VIC, Australia, July 28 - August 1*, 2014, pp. 5–8.

[17] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook, "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" in *Human Factors in Computing Systems, CHI '97 Conference Proceedings, Atlanta, Georgia, USA, March 22-27*, 1997, pp. 258–265.

[18] C. Parnin, C. Görg, and S. Rugaber, "Codepad: interactive spaces for maintaining concentration in programming environments," in *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26*, 2010, pp. 15–24.

[19] B. Biegel, S. Baltes, I. Scarpellini, and S. Diehl, "Codebasket: Making developers' mental model visible and explorable," in *2nd Workshop on Context for Software Development (CSD), Florence, Italy, May 19*, 2015.

[20] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing programs with jeliot 3," in *Proceedings of the working conference on Advanced visual interfaces, AVI 2004, Gallipoli, Italy, May 25-28*, 2004, pp. 373–376.

[21] A. E. R. Campbell, G. L. Catto, and E. E. Hansen, "Language-independent interactive data visualization," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 2003, Reno, Nevada, USA, February 19-23*, 2003, pp. 215–219.

[22] P. V. Gestwicki and B. Jayaraman, "JIVE: java interactive visualization environment," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, Vancouver, BC, Canada*, 2004, pp. 226–228.

[23] J. H. C. II and T. D. Hendrix, "jgrasp: a lightweight IDE with dynamic object viewers for CS1 and CS2," in *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2006, Bologna, Italy, June 26-28*, 2006, p. 356.

[24] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003.