

**JavelinaCode: A Web-Based Object-Oriented Programming Environment
with Static and Dynamic Visualization**

by

Jeong-sug Yang

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 6, 2016

Keywords: JavelinaCode, static visualization, dynamic visualization
web-based programming environment

Approved by

Kai H. Chang, Chair, Professor and Chair of Computer Science and Software Engineering
T. Dean Hendrix, Associate Professor of Computer Science and Software Engineering
David A. Umphress, Professor of Computer Science and Software Engineering

Abstract

This research proposes an approach to provide source code along with structural and behavioral aspects of visualizations synchronized in a web-based programming environment, JavelinaCode. The aim of the approach is to help student programmers better understand the structure and the runtime behavior of a Java program, and to improve their ability to comprehend object-oriented programming concepts, thereby reducing their cognitive workload in Java programming through an effective development environment. Using JavelinaCode, student programmers can write Java programs directly in a frontend web browser without any software or plug-in installation. They are provided with a view of the static state of a Java program in UML class diagrams and the dynamic run-time state of the program by stepping forward and backward through program execution.

In this dissertation, an overview of the JavelinaCode system, its unique design principles, and implementation are described in detail. To investigate the effectiveness of JavelinaCode, both quantitative and qualitative evaluations were carefully designed and conducted to test hypotheses on student performance on programming tasks. The quantitative study indicated that having both visualizations in JavelinaCode did positively impact the correctness of solving problems. The qualitative study supported the positive effect of JavelinaCode on helping students better understand Object-Oriented design concepts and meeting goals of providing satisfaction. The dissertation makes a conclusion with the discussion of contributions, benefits, and future work.

Acknowledgments

I would like to express a sincere appreciation to my advisor, Dr. Kai H. Chang, for his continued support. It is truly amazing to see him consistently support me after such a long time since I left Auburn. Throughout the years, I feel very lucky to have him in my work for both masters and doctoral degrees at Auburn. I would also like to extend my sincere appreciation to the advisory committee members, Dr. Dean Hendrix and Dr. David Umphress, for their help in many different ways, and Dr. Soo-young Lee for his generous support on serving as a University reader.

My deepest gratitude goes to my husband, Dr. Young Lee. Without his unconditional love, thoughtful guidance, and persistent support, it would not even be possible to finish the dissertation and this precious moment would not have come to me.

Table of Contents

Abstract	ii
Acknowledgments	iii
List of Tables	vii
List of Figures	viii
1 Introduction	1
2 Literature Review	7
2.1 BlueJ	7
2.2 CoffeeDregs	9
2.3 Jeliot 3	11
2.4 AguiaJ	12
2.5 JIVE	14
2.6 jGRASP	16
3 JavelinaCode	19
3.1 System Overview	19
3.2 Design Principles	20
3.3 Implementation	21
3.4 Unser Interface	24
3.5 Modeling Example	27
3.6 Mechanic	28

3.6.1 Getting Started	28
3.6.2 Editing with Ace Editor	29
3.6.3 Creating, Opening, and Deleting a Project	30
3.6.4 Adding, Closing, Renaming, and Deleting a Class	32
3.6.5 Saving, Compiling, and Running a Project	34
3.6.6 Viewing UML Class Diagrams	35
3.6.7 Visualizing Program Execution	38
3.7 Anticipated Benefits	38
4 Synchronized Static and Dynamic Visualization.....	41
4.1 Plant UML	41
4.2 Java Visualizer	42
4.3 Static and Dynamic Visualization in JavelinaCode	44
4.3.1 Customized Plant UML for Static Visualization	44
4.3.2 Customized Java Visualizer for Dynamic Visualization	49
4.3.3 Case Study: yo-yo effect with Synchronized Static and Dynamic Visualization	54
5 Comparative Analysis of Educational Programming Environmental Tool	57
5.1 Comparison of Download and Install Time	57
5.2 Comparison of Quality of User Interface and Aspects of Visualization	59
6 Evaluation	71
6.1 Introduction	71
6.2 Validation	72
6.3 Quantitative Evaluation	73

6.3.1 Hypotheses.....	74
6.3.2 Questionnaire.....	74
6.3.3 Data Analysis.....	79
6.3.4 Experiment 1.....	80
6.3.4.1 Participants.....	80
6.3.4.2 Method and Procedure.....	80
6.3.4.3 Results.....	89
6.3.5 Experiment 2.....	93
6.3.5.1 Participants.....	93
6.3.5.2 Method and Procedure.....	93
6.3.5.3 Results.....	93
6.4 Qualitative Evaluation.....	97
6.4.1 Objectives and Questionnaire.....	98
6.4.2 Participants.....	104
6.4.3 Method.....	100
6.4.4 Results and Discussion for Aspect of Visualization.....	101
6.4.5 Results and Discussion for Usability of the System.....	106
6.4.6 Results and Discussion for Associated Objectives.....	112
7 Conclusion and Future Work.....	115
7.1 Contributions and Benefits.....	120
7.2 Future Work.....	121
8 References.....	124

List of Tables

Table 1. Symbols used in Plant UML to draw a class diagram	41
Table 2. Comparison of download and install time of programming environmental tools	58
Table 3. No. of subjects participated in both experiments	80
Table 4. Statistical evaluation of response time in Experiment 1	90
Table 5. Statistical evaluation of correctness in Experiment 1	92
Table 6. Statistical evaluation of response time in Experiment 2	95
Table 7. Statistical evaluation of correctness in Experiment 2	97
Table 8. Background questions.....	98
Table 9. Visualization related questions	98
Table 10. Usability related questions.....	99
Table 11. Mean rating and percent agreement for visualization related questions.....	107
Table 12. Mean rating and percent agreement for usability related questions	108
Table 13. Mean rating and percent agreement for associated objectives	113

List of Figures

Figure 1. JavelinaCode system overview.....	20
Figure 2. JavelinaCode infrastructure on AWS cloud computing platform	23
Figure 3. User interface of JavelinaCode.....	26
Figure 4. Modeling example linking source code with static and dynamic visualization	28
Figure 5. Login window.....	29
Figure 6. Creating a new project.....	30
Figure 7. Opening an existing project.....	32
Figure 8. Deleting a project	32
Figure 9. Adding a class.....	33
Figure 10. Closing a class	33
Figure 11. Renaming a class	34
Figure 12. Deleting a class.....	34
Figure 13. Output of the project in the terminal window	35
Figure 14. Example of an individual class diagram.....	36
Figure 15. Example of a compact class diagram	36
Figure 16. Example of a detailed class diagram	37
Figure 17. Run time state of program execution.....	39
Figure 18. Textual input and its class diagram by Plant UML	42
Figure 19. Visualization of Java program execution by Java Visualizer.....	43
Figure 20. Class, extended class, and implemented interface.....	45

Figure 21. Dependency	46
Figure 22. Method.....	47
Figure 23. Variables.....	48
Figure 24. Flow of drawing a UML class diagram in JavelinaCode	48
Figure 25. Dynamic visualization of Java program using customized Java Visualizer.....	50
Figure 26. Flow of generating a run-time visualization in JavelinaCode	51
Figure 27. Modeling example of yo-yo effect	55
Figure 28. Screenshot of BlueJ	60
Figure 29. Screenshot of Jeliot 3 with a Theater	62
Figure 30. Screenshot of jGRASP with a Canvas window.....	63
Figure 31. Screenshot of AguiasJ.....	65
Figure 32. Screenshot of JIVE with object and sequence diagrams	67
Figure 33. Screenshot of JavelinaCode with static and dynamic visualization	68
Figure 34. Java classes used for Session 1 in both Experiments 1 & 2.....	77
Figure 35. Java classes used for Session 2 in both Experiments 1 & 2	78
Figure 36. Screenshots of (a) PolyShape project in NetBeans IDE, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question.....	82
Figure 37. Screenshots of (a) yo-yo problem project in NetBeans IDE, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question.....	83
Figure 38. Java classes used to familiarize with JavelinaCode system	85
Figure 39. Screenshots of (a) PolyShape project in JavelinaCode, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question.....	87
Figure 40. Screenshots of (a) yo-yo problem project in JavelinaCode, (b) Web Page for ID and class selection, (c) Web Page for the first question, and (d) sample page for usability questions	88

Figure 41. Comparison of average response time with PolyShape project	90
Figure 42. Comparison of average response time with yo-yo problem project.....	90
Figure 43. Comparison of correctness with PolyShape project.....	92
Figure 44. Comparison of correctness with yo-yo problem project	92
Figure 45. Comparison of average response time with PolyShape project	94
Figure 46. Comparison of average response time with yo-yo problem project.....	95
Figure 47. Comparison of correctness with PolyShape project.....	96
Figure 48. Comparison of correctness with yo-yo problem project	96
Figure 49. Comparison of average response time with PolyShape project	98
Figure 50. Comparison of average response time with yo-yo problem project.....	99
Figure 51. Comparison of correctness with PolyShape project.....	100
Figure 52. Comparison of correctness with yo-yo problem project	100
Figure 53. Rate students in Java	104
Figure 54. UML class diagram related question 1	105
Figure 55. UML class diagram related question 2.....	105
Figure 56. Run time visualization related question 1	107
Figure 57. Run time visualization related question 2	107
Figure 58. Both static and dynamic visualization related question 1	109
Figure 59. Both static and dynamic visualization related question 2	109
Figure 60. Both static and dynamic visualization related question 3	109
Figure 61. Mean rating for visualization related questions.....	110
Figure 62. Mean rating for usability related questions	111

1 Introduction

Brian Kerningham stated, “No matter what, the way to learn to program is to write code, and rewrite it, and see it used, and rewrite again. Reading other people's code is invaluable as well” [1].

Object-Oriented Programming (OOP) is one of the core areas in Computer Science, and learning OOP becomes a major challenge in Computer Science education. While Java has been widely used in teaching and learning of an object-oriented programming language and it is one of the most popular programming languages taught at universities and colleges, studies have discovered it is difficult for students to learn due to the underlying OO concepts and principles, such as encapsulation, abstraction, inheritance, and polymorphism [2, 3]. With inheritance, polymorphism, and dynamic binding features, objects in OOP typically interact with each other asynchronously, and method calls are difficult to track.

In learning OOP in Java, difficulties, including programming language syntax, programming environment, and problem solving skills, have been also commonly mentioned. The reasons for learning difficulty have been originated from different sources, such as programming language itself, complexity and domain of a problem, program design, programming environment, programmer’s logical thinking ability and programming skills as well as OO concepts and principles. Comprehension of the functionality, structure, and behavior of a program is a crucial

component of the programming learning process. OO programs are also known for being complex to visualize and their control flow difficult to follow, making the learning process challenging.

A student's development environment is another factor that influences learning OOP, because it requires students to manage issues, such as platform dependencies and conceptual understanding of classes, objects, and Object-Oriented Design (OOD). To help student programmers better understand the structure of a program and the concepts of Object-Oriented design, visualizations in various formats have been applied to programming environments. Software visualization tools are seldom employed in the OOP development environment. This is because current tools provide visualization for a single aspect (i.e., structure or behavior) of the software. Visualizations from these tools are not enough to support the understanding of OOP.

Many programming tools have been developed to visualize source code as graphs, diagrams, or animations in such a way they attempt to help students to enhance source code comprehension. Other visual programming tools provide visual notations as objects to develop a program, without writing a single line of program code. One problem in visualizing the source code is the visualization does not help programmers write code, and code writing is still the programmer's responsibility. Source code visualization can only help programmers understand the code. It does not help the programmers to write code. Even software developers can write code, using visual programming tools, but they still must know how to code in programming languages.

Studies have pointed out that novice student programmers often encounter difficulties in installing Integrated Development Environment (IDE), Java Development Kit (JDK), and plugins, and setting up and modifying system environment variables on their own machines [4, 5, 6]. In

addition, some IDEs, like Dev-C++ and Visual Studio, run on only the Windows Operating System (OS), while Mac OS is in high demand these days. Students who own a Mac, but must program on Windows, would have more restricted access to computers in program development.

To address the limitation of current visualization techniques, this research proposes an approach to integrate abstraction with structure and behavioral aspects of OOP. The research presents a web-based interactive and educational programming environment, JavelinaCode, designed for teaching object-oriented programming in Java. It aims to enhance student programmers' programming and logical thinking skills and to help them gain object-oriented design concepts. JavelinaCode provides integrated static and dynamic visualizations, representing the static state of a Java program in a UML class diagram and the dynamic run-time state of the program execution. When a student is writing a line of code, its corresponding structural information of the program is highlighted in the class diagram, and the functional information of data is synchronized in the run-time state visualization. Through the synchronized multi-view real time visualization, both the structural and functional feedback of the current line of the source code is immediately provided to the student while coding. Students access JavelinaCode through a web browser to program in Java, with no required software or plugin installation on their local computer.

The motivation for this work is the lack of an interactive OOP environment integrated with both static and dynamic visualizations. A static class diagram is generated from the static information derived from source code. Dynamic tracing visualization is generated from the dynamic information of the program execution. When polymorphism is taught using class diagrams, sequence diagrams, and source code, students must simulate running a program in

their minds to understand how it works. As the combined visualization of both static and dynamic aspects suggests [7], visualizing the dynamic run-time state corresponding to the source code and static diagrams will reduce the burden of a complex simulation in students' minds. This interactive hybrid visualization approach is expected to help students better understand current code and choose the right design alternatives. The research hypothesis explored in this dissertation is whether an interactive static and dynamic visualization that incorporates structural and behavioral views supports OOP comprehension in an OOP development environment.

The aims of this research are: 1) to provide an effective OOP environment with a static and dynamic visualization; 2) to enhance OOP learning by improving the effectiveness of the OOP development environment; and 3) to support OOP teaching by inspecting students' behavioral patterns in OOP learning, which would be a future work to be accomplished.

Main objectives of this research are: 1) to provide student programmers an easier set-up process for an online programming environment; 2) to help student programmers better understand their programming by linking program source code with the visualizations of UML diagrams and run-time execution; 3) to make programming easier for students by understanding the structure of a program and the execution state of data while they are coding.

The dissertation comprises seven chapters, structured as follows:

In Chapter 2, various educational programming environmental tools are introduced and compared, based upon how the OO features are highlighted in visualization in terms of static and dynamic components of an object-oriented program. It describes a literature review and

discusses how this research is directly and/or indirectly different from the past and current works.

Chapter 3 presents a web-based educational programming environment, JavelinaCode, for enhancing student learning capabilities in OOP and OOD. JavelinaCode proposes an approach to program in Java, supporting static and dynamic visualizations in a real time multi-view model. The chapter demonstrates the overall architecture of JavelinaCode system, its unique design principles, a user interface of the system, and various techniques and strategies used in implementation.

Chapter 4 highlights how the static structural information of a program and the functional information of the program data are customized, deployed, and used in both visualizations and how this hybrid static and dynamic visualization with source code will help students better understand the structure of the program and the execution of the program data. Motivating modeling examples are used to indicate the benefits of using the JavelinaCode system.

Chapter 5 reports on the results of a comparative analysis of the educational programming environmental tools discussed in Chapter 2, on the basis of time constraints for download and installation, complexity of the download process and tool's interface, and the provision of static and dynamic visualizations.

Chapter 6 reports on the results of experiments conducted in evaluating the educational effectiveness of JavelinaCode. Both qualitative and quantitative experiments are carefully designed and conducted to correlate student users' usage and performance in program comprehension by comparing the results of data from a group of student users using Java source

with two aspects of visualizations in JavelinaCode and another group of student users using the same code in plain text in the standard IDE, NetBeans.

Chapter 7 summarizes the dissertation with the goals and objectives of the research, its anticipated benefits and contributions, and the results of the experiments, and finally, makes a conclusion with general observations about the JavelinaCode system. The dissertation is finalized with a description of future work.

2 Literature Review

This chapter compares programming environmental tools, based upon how the OO features are highlighted in visualization. Unlike JavelinaCode, these tools must be downloaded and installed as a stand-alone program or plugged into Eclipse (AguiaJ and JIVE) or NetBeans (CoffeeDregs).

2.1 BlueJ

Description: Blue is an integrated system for both a programming language and a software development environment [8]. BlueJ originated from Blue and is a development environment designed for teaching how to develop Java programs. Its main strengths are interactivity, visualization, and simplicity. According to Kölling and Rosenberg [9], BlueJ provides an interactive creation of objects from any class in a project. Once an object is created, the object is visible to programmers; any of the object's public methods can be invoked by selecting it from a pop-up menu, method parameters may be entered, and its results are presented in dialog windows. It allows programmers to interact with objects by inspecting their values, calling methods, and passing them as parameters.

BlueJ also presents a graphical visualization through UML class diagram. When a Java project is opened, the main window displays a UML class diagram visualizing its structure, so programmers can directly interact with classes and objects [10]. Kölling et al. claimed BlueJ is

simple for students to use [9, 10]. They need not spend too much time getting used to the programming environment; instead, they get started programming easily. BlueJ is claimed for first year object-oriented programming teaching due to its simplicity and pedagogy. Kölling et al. suggested a set of guidelines and a sequence of assignments confirming the guidelines with BlueJ for teaching object-oriented programming [9, 10]. The guidelines are: a) Objects first; b) Don't start with a blank screen; c) Read Code; d) Use "large" projects; e) Don't start with "main"; f) Don't use "Hello World"; g) Show program structure; and h) Be careful with user interface.

Evaluation: The effectiveness of BlueJ was initially evaluated at Monash University during the first semester of using BlueJ in 1999, and its results were reported [5]. Students participating in the evaluation felt frustration initially, but by the end of the semester, they felt that BlueJ helped them learn Java programming. A follow-up evaluation study was also done with students in the second of the two consecutive first year programming courses at the same university [11]. According to the paper, the student participants indicated BlueJ helped them understand object-oriented paradigm [11]. The biggest strengths of using BlueJ include an ability to help students link source code and visualizations and an ability to support learning in the cognitive domain, while some issues, such as usability and performance, were identified.

Issues: While the positive effectiveness of BlueJ has been discussed and evaluated, these issues were raised as well: a) Hagan and Markham reported students participated in the study had difficulties with installing and running BlueJ system [5]. Although it was highlighted the installation difficulties were resolved [11], the paper mentioned speed or performance had been raised due to the computers in the student labs needing to be upgraded with optimal

configuration for running BlueJ. To run BlueJ, users must download and install it on their own platform. Each platform has its own installation guideline, and proper configuration is needed for some platforms; b) Linking Java source code and its UML class diagram visualization is done in separate windows. To see the source code and its correspondent class diagram, users must switch windows from one to another. It is hard to synchronize them with each other; c) BlueJ does not emphasize communication between objects or illustrate the association or aggregation of classes [12]; d) BlueJ provides no dynamic visualization of program execution [13, 14]; and e) One of BlueJ's guidelines, 'Don't start with main' makes users get confused how to run a Java program.

2.2 CoffeeDregs

Description: CoffeeDregs is an educational visualization tool of executing Java programs to support teaching. The aim of CoffeeDregs is clear to teach Object-Oriented programming and to build a conceptual semantic model through the visualization of the connection between static source code and dynamic execution in objects [13]. CoffeeDregs's approach in teaching OO programming differs greatly from BlueJ's 'Objects first' approach. CoffeeDregs introduces the activity of a Java program, providing its semantics of the execution model and presenting an abstract view of variables and objects. A program execution is visualized in snapshots of run-time state of a Java program, showing its structure and changes made during the execution.

The main concepts CoffeeDregs seeks to visualize are objects (variables and methods), the lifetime of objects and methods, the structures of classes and objects, and the difference between classes and objects [13]. The contours and boxes are used to visualize objects, variables, and methods inside the objects and local variables inside the methods. The contours are also used to

indicate the scope and lifetime of the objects and method calls. Methods are only visualized while they are active. The references between objects and their method calls are visualized through visible arrow pointers, showing the structure of objects and method calls from outside objects. Each object is separately displayed in the main window for an active class. The visualization is done by stepping through the program code, and reverse stepping is also possible. CoffeeDregs is a stand-alone visualization tool, but also, it can be incorporated as a plug-in to NetBeans IDE.

Evaluation: To assess the educational effectiveness of CoffeeDregs, the qualitative experiments with the students in Innovation Sciences at Eindhoven University of Technology were conducted, and their results were reported in [15]. The student participants could find the delicate bug in the assignment, while running the program with the visualization. Another student used CoffeeDreg's visualization to check if the execution model and program code were equal. From the experiments, CoffeeDregs can be considered to represent for a classroom lecture, and it was beneficial for students learning OOP [15].

Issues: Luijten found the link between source code and run-time execution visualization was not directly recognized, in particular, to trace instance variables and method arguments when fixing a problem [15]. However, the paper pointed out the problems would not occur if the student participants became more familiar with using the tool. To make its visualization more feasible, CoffeeDregs is under development to better suit programmers covering the use of library code, multi-threads, and user-interaction GUI [13].

To run CoffeeDregs, its stand-alone version must be downloaded and installed, or its plug-in module plugged-in to NetBeans IDE, with JDK and NetBeans installation required on a local computer. CoffeeDregs only provides the stand-alone version for Windows.

2.3 Jeliot 3

Description: Jeliot 3 is a program visualization tool designed to help novice students learn both procedural and object-oriented programming [14]. Jeliot 3 originated from Eliot and Jeliot I, with the basic production of algorithm animation for the data flow of variables in different platforms and Jeliot 2000 with the supporting visualization of control flow and expression evaluation.

The main goal of Jeliot 3 is to have the system easy to use, consistent in its visualization in all cases, and extensible, both internally and externally. Although Jeliot 3 is aimed for novice programmers, the system provides visualization of a large subset of Java programs, supporting object-oriented concepts, such as objects and inheritance. The main structure of the animation frame used in visualization includes method frame, expression evaluation frame, constant frame, and instances frame. Each element appears in its own frame. UML class diagram-like notation is used to visualize objects shown as boxes containing attributes and values and references shown as lines connecting the object and its variable. The link between program source code and its corresponding visualization area is also synchronized and highlighted to identify them together at a time.

Jeliot 3 is claimed to be used for an introductory programming course, since it helps students in providing semantics and engaging them to the learning process, and it helps students and

teachers in sharing graphical and verbal vocabularies with animation visualization that makes it easier to discuss programming concepts [14]. Jeliot 3 is integrated to BlueJ 1.3.5 and BlueJ 2.0.

Evaluation: A qualitative investigation of Jeliot 3 was conducted to see how undergraduate students at the University of Warwick in Finland used the tool in solving programming assignments and how the animation was understood by the students, and its preliminary findings were reported [16]. It was found that the animation of Jelot 3 was easy to use and useful for the students to debug programs.

Another experimental study to examine students' performance and attitudes on OO programming was conducted at Thammasat University in Thailand [17]. Based on the regression analysis with the collected data from the experiment, the paper concluded the student group significantly outperformed on their exam scores by achieving higher exam scores through Jeliot 3. However, the use of Jeliot 3 showed no significantly difference on students' attitudes toward OO programming.

Issues: The paper reported Jeliot 3 animation was difficult for the novice student programmers to understand, and it was suggested to add verbal explanations to the Jeliot 3 visual animation, which might optimize students' learning with graphics and audio or textual captions [16]. As the length of code and its complexity increase, the visual animation becomes difficult to understand, and objects are sometimes overlapped, which makes it more challenging to follow the program execution.

2.4 AguiaJ

Description: AguiaJ is a pedagogical tool for interactive experimentation and visualization of OOP in Java [18]. AguiJ uses a visual representation of metaphors to illustrate OO concepts (encapsulation, interface, polymorphism, and inheritance) by showing and hiding private instance variables, indicating inherited members and overridden operations, and using image domains [19].

The AguiaJ window comprises two major areas: a class area and an object area. Defined classes in source code are adapted into the class area, and created objects are populated in the object area. The class area contains constructors, static methods, and static variables. The static members are illustrated in this area to distinguish instance members in the object area. When a method or constructor is called by a user, its object appears in the object area. The object is illustrated as a box containing interaction widgets to its constructor, and references are represented by pointers to the objects referenced. The objects can be directly controlled by the user by typing Java instructions, for instance, to call methods of the object in the Java bar at the bottom of the window.

AguiaJ uses interaction metaphors to aid understanding of OOP concepts, such as encapsulation, interface, polymorphism, and inheritance. This novel metaphor supports the user interaction with the objects, for instance, if a method in a program may not be executed due to encapsulation [19]. The metaphor supports polymorphic behaviors by associating reference types and structural elements of objects. The metaphor also supports inheritance concept by visualizing inherited objects and identifying the difference between the objects of superclass and subclass.

Evaluation: An evaluation study of AguiaJ was conducted with 47 Computer Science and Engineering (CSE) major students and 53 Informatics and Management (IM) major students at

the University of Lisbon [20]. The study was to measure the impact of using the AguiaJ environment in terms of success rates to pass a 12 week course and student satisfaction. Some observed approval rates were increased in both major groups. In terms of student satisfaction, the survey data revealed most the students positively answered the process of developing a project with AguiaJ was fun [20].

Another evaluation study was done through a two-week intensive training with 13 employees in a consulting company, whose background was not Computer Science [19]. At the end of the training, the trainees were given a small project to develop, with a questionnaire, regarding the suitability of the AguiaJ and the usages of the images. Resultant data revealed positive impacts on training for both suitability and visualization of the images used. It was concluded that AguiaJ was usable for interactive lecturing and exercising, and teaching OOP worked effectively with the domain of image manipulation [19].

Issues: While AguiaJ supports an instant change between source code and object illustration, it is hard to map them together, because the editor window and the visualization window are not seen simultaneously. The user must switch between the windows from one to another. Like BlueJ, AguiaJ exercises OO concepts, without starting the ‘main’ method. AguiaJ is open source running on top of the Eclipse IDE. Users first download and install the Eclipse IDE and then install AguiJ as a plugin.

2.5 JIVE

Description: JIVE (Java Interactive Visualization Environment) is an interactive program execution environment, supporting the visualizations of both the runtime state and the call history

of a Java program. Its approach is to facilitate program understanding and to enhance the comprehension of the runtime execution of object-oriented programs in Java, through displaying runtime object structures, providing object states in multiple views, visualizing the history of program execution with sequence diagrams, supporting forward and backward program execution, and producing clear drawings of the object structure and method-invocation sequence [21, 22].

A complete run-time state of a program is visualized through a contour (object) diagram, showing an object structure with method activations in object contexts. The contour diagram comprises a set of rectangles (contours) with bindings of parameters and variables. A contour diagram is visualized in both a detailed view and a compact view. The detailed view demonstrates the highest level of detail about object's complete run-time states, including an object's context, variables defined in the context, and values for the variables. A history of program execution is visualized, using a time sequence diagram. In the visualization, object contours are represented as contexts, and active methods are shown as rectangles with vertical lines. When a user selects an object context or a method in the sequence diagram, JIVE jumps to the contour diagram for the corresponding state.

Evaluation: Through the experiences of using the extended notation of the contour model in years, Jayaraman and Baltus found the technique of using the contour (object) diagrams were appropriate for object-oriented programs [23]. It also clarified the diagrams were useful for debugging, particularly to make a clear difference between a user's imagined structure and an actual structure created. The time sequence diagrams generated in JIVE also have proven effective in explaining the behavior of design patterns and program structures [22].

Other experiments on several programs were conducted to carry out multiple runs of each program, to merge state diagrams, and to produce a composite diagram. The experiments found that both single and composites state diagrams provided good insight into the program and helped uncover subtle flaws in the implementation [24].

Issues: Despite the effectiveness of the sequence diagrams, JAVE makes the history of the diagrams very complicated, with a large program providing a comprehensive diagram. It was suggested to modularize the history information into multiple diagrams and relate them to the corresponding compact contour diagrams [22]. Jumping from one program state to another in the sequence diagram became inefficient, because the interaction module only supports stepping in a sequence from the beginning to the end. In addition, it is more likely to have difficulty writing and understanding the code, because the visualizations of a Java program behavior with object and sequence diagrams are done with or without the Java source code.

2.6 jGRASP

Description: jGRASP is a lightweight IDE for program visualizations [31, 32], which allows interactions with its dynamic viewers to support understanding of data structures [36]. Three ways to interact with the viewers are via debugger, workbench, and text-based interaction tab. Using the debugger, users can select a line of the source code to stop program execution and start debugging from the line by clicking the ‘down arrow’ button of Threads and continue checking the values of variables or references for each line. Using the workbench viewer, users can see the drawing of data structure on the basis of references and values. Users also can create instances of a class, using menus on the UML class diagram or source code editing windows. The text-based

interaction tab allows users to enter expressions and statements that can be evaluated and executed, without compiling and running an entire program.

jGRASP supports structure identifier viewer to analyze source code, detect arbitrary data structures, and visualize them. It displays internal workings of data structures as elements are removed or added, which might be useful in finding bugs in algorithm and data structure implementations [33]. jGRASP also allows for standard program execution using a canvas window to step through code line by line [37]. The canvas is displayed in a separate window from the source code. Program output is displayed at the bottom of the initial window. Execution of program in the canvas can be paused, and elements from the debugger or work bench can be dragged into the canvas window. This process transforms elements into visualization, a frame with the name of an object and its variables, and the values of these variables.

Evaluation: An data structure visualization system in jGRASP has been evaluated in the light of recognizing data structures using structure identifier, conducting code understanding experiments, and analyzing feature utilization data from end users [33]. In testing structure identifier against textbook data structure examples, the results of testing textbook examples revealed structure identifier worked correctly to display 82% of the examples directly, 92% by selecting a field, and 97% by configuring the viewer [34]. For the code understanding experiments using jGRASP data structure viewers, it was found students could detect and correct logical errors more accurately using the viewers than the traditional methods of visual debugging for singly linked lists [35]. Other experiments to investigate if students would code faster and with higher accuracy, using data structure visualization on certain programming tasks, also have shown students in the experimental group consistently performed better than a control group, and

the viewers were helpful during the code development [36]. From the viewer utilization data collected from January 1, 2012 to May 13, 2014, it was analyzed that viewer usage alone was high, while debugger usage alone was relatively low. Viewer usage as a fraction of debugger usage was the highest when the debugger was heavily used. Hundreds of thousands of viewers were opened by end users each year, and the debugger and viewer usage had been gradually increased during the period.

The efficacy of canvas viewer was also evaluated through controlled experiments and user questionnaires to investigate the impacts of viewers in terms of student performance on programming tasks [37]. The results of the experiments revealed positive effect of the viewers that students used the viewers did code faster and had fewer errors in program development, and found more errors and faster in existing code than students not using the viewers. The results of questionnaires with faculty members across the country and students at Auburn University also showed that canvas visualization could positively affect student learning and was a useful aid in learning.

3 JavelinaCode

This chapter presents a web-based educational programming environment, JavelinaCode, on the basis of its system overview, unique design principles, implementation, and interface. JavelinaCode is designed for teaching object-oriented programming in Java. It aims to enhance student programmers' programming skills and to help them understand object-oriented design concepts. It provides integrated static and dynamic visualizations: the static state of a Java program in a UML class diagram and the dynamic run-time state of the program execution.

JavelinaCode provides integrated static and dynamic visualization of Java programs at line level and a full overview of a project under development. This interactive hybrid visualization approach is expected to help students better understand current code and choose the right design alternatives. With the synchronized multi-view real time visualization with source code, JavelinaCode is expected to reduce student programmers' cognitive workload in Java programming and to enhance comprehension of object-oriented programming and design concepts. The research hypothesis explored in this paper is whether an interactive static and dynamic visualization that incorporates structural and behavioral views supports OOP comprehension in an OOP development environment.

3.1 System Overview

Figure 1 gives the system overview of JavelinaCode. A student accesses JavelinaCode through a front end web browser. Java source code created by the student is sent to a back end server.

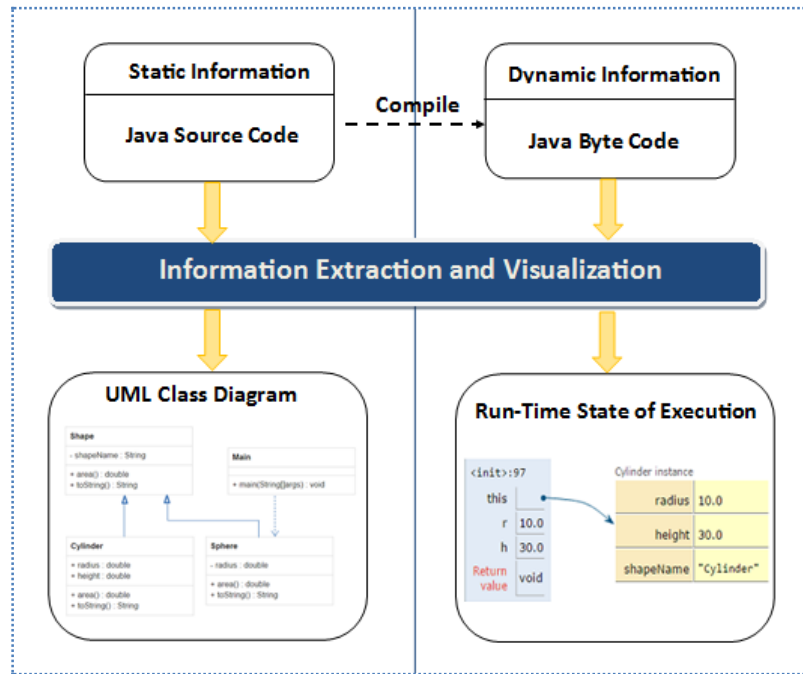


Figure 1. JavelinaCode system overview

From the source code, static structural information, such as instance/class variables and their types, methods and parameters, and relationships among classes, is extracted and visualized in a UML class diagram. From the Java bytecode translated by the Java compiler on the server, dynamic functional information, such as values of instance/class variables, is extracted and dynamic run-time state of the program is visualized.

3.2 Design Principles

For the best practice of OOP and OOD, JavelinaCode is designed with these principles: 1) Easy to Access, 2) Easy to Use, 3) Easy to Understand, 3) Source Code and User Centered, 4) Static Visualization of Structural Information of a Program, 5) Dynamic Visualization of Functional Information of Data, 6) Static and Dynamic Visualization together, 7) Synchronized Multi-View with Source Code, and 8) Structural and Functional Feedback in real Time.

JavelinaCode is platform-independent. Students use a web browser to develop and run a Java program with no required software or plug-in installation. They can program anywhere and anytime, using their laptops, desktops, tablets, and mobile phones. The project files created by a student are saved on the cloud storage. The student needs no memory system to keep and manage the data. The student is freed from concern of continuous version changes and evolutions to the Java language, IDEs, plug-ins, and operating systems. This provides a great deal of accessibility and usability to the program development environment.

JavelinaCode is both source code and user centered. When a student writes a line of code, the corresponding structural information of the program is dynamically linked with the two sets of UML class diagrams, and the functional information of data is synchronized in the run-time state visualization. Real time feedback of the current line of the code is immediately given to the student. The links are highlighted when the user clicks the 'Back' or 'Forward' key after completing the line, such as variable declaration and/or initialization, method declaration, or expression. This will greatly help the student establish the mental model of program execution [17].

3.3 Implementation

JavelinaCode is being implemented with a front end, written in HTML5, CSS3, and jQuery, and a back end, written in PHP. Ace, an embedded open-source code editor, is fully integrated into the environment [25]. To generate a UML class diagram for the static information of the Java source code, PlantUML, an open-source tool that converts textual code description to draw UML diagrams, is integrated into the system [26]. For the run-time state visualization of program execution, the Java Visualizer by David Pritchard and Will Gwozdz [27], based on the Online Python Tutor [28] by Philip Guo, is fully integrated into the interface. The Java Visualizer reads Java source code as input, traces the Java bytecode data (objects, methods, and variables) using Java debugger, and outputs the trace in JASON format for the front end visualization. Information of the abstract design of the Java source code is extracted to draw the UML class diagram.

Platform: JavelinaCode is being developed on the AWS (Amazon Web Services) cloud computing platform on the back end virtual server, running the Ubuntu 14.04 LTS operating system, Apache 2 HTTP server, and MySQL database server with php5 and Java 8 installed. Also, PhpMyAdmin is used to handle the administration of the MySQL and to interact with its databases for managing users and project files.

Figure 2 shows the infrastructure of the JavelinaCode that employs the services provided by AWS [29]. Users access JavelinaCode web IDE through a front end web browser. This web IDE is typically structured into three logical tiers: the first tier - web browser, the second tier - an application server, and the third tier - a database server. On the AWS infrastructure, an EC2 (Elastic Compute Cloud) instance of Ubuntu 14.04 LTS virtual server and a DB instance of MySQL as a database server are created and used in the cloud. The web and application tiers run

on the EC2 instances, and the database tier runs on the DB instances. The load balancer distributes traffic evenly among the EC2 instances.

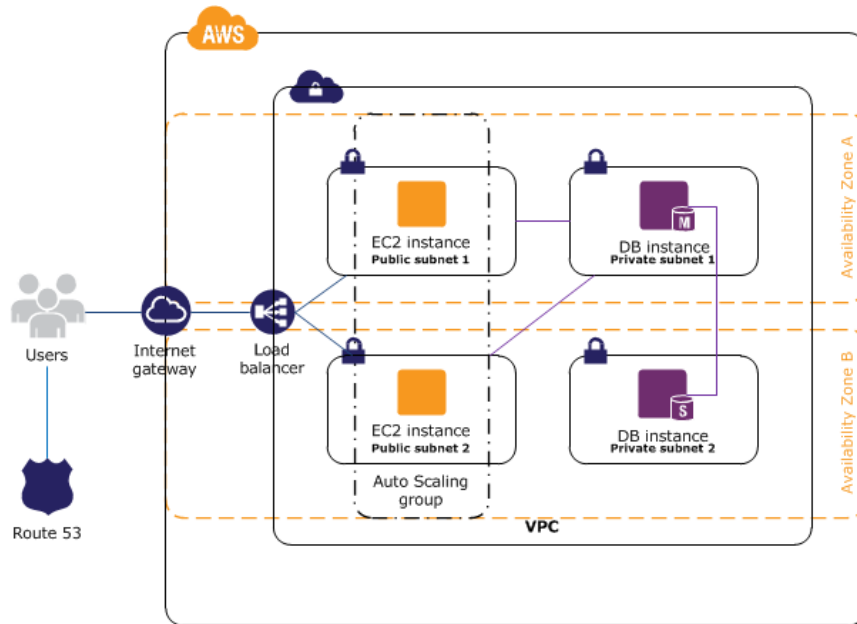


Figure 2. JavelinaCode infrastructure on AWS cloud platform

The Auto Scaling group maintains the EC2 instances and handles any traffic loads. It spans multiple Availability Zones to protect a potential failure of another Availability Zone. AWS Route 53 provides reliable routing of four domains: <http://javelinacode.com/>, <http://javelinacode.org/>, <http://www.javelinacode.com/>, and <http://www.javelinacode.org/>.

Cloud computing: There are some benefits of running JavelinaCode on the AWS cloud computing. The system is reliable, secure, and scalable. Since data is stored and retrieved from multiple virtual servers on the cloud, even if one of the servers crashes unexpectedly, the system won't go down. On the AWS global infrastructure, the system is secure with the foundation services of computing, storage, database, and network. The elastic scalability helps us match resources to the demands of our own. We choose the resources we need to scale our web


application, up or down, based on our demand. It is also flexible to run any software we want. We choose the operating system, programming language, software, and other web application platform to build and run our web application.

Web-based: The biggest benefit of using web-based IDE is users simply use a web browser to run JavelinaCode, with no required software and plugin installation or configuration on a local computer. When students learn programming in Java, they need to know how to edit, compile, debug, and run on a Java programming environment, which gets involved with the JDK (Java Development Kit), an editor, and an input/output console. As indicated [4], many novice Java programmers encounter difficulties in using the software tools, including installing the JDK on their machine and modifying the system environment variables to complete the JDK installation. They need to understand and use the tools and to compile, debug, and run Java programs in the environment. Using JavelinaCode IDE, students need not know how to install tools or modify environment variables. They can program anywhere and anytime, not only with their laptops or desktops, but also their tablets or mobile phones. This provides a great deal of easy accessibility to an environment for program development.

Target users: The target users of JavelinaCode are students, learning how to program in Java and how to design an OO system, and instructors, teaching Java programming and designing a system. However, not only the students and instructors can benefit from this system, but also professional developers can benefit in tracing and understanding their code, especially for debugging.

3.4 User Interface

The user interface of JavelinaCode is presented in Figure 3. The user interface comprises four main components: static UML diagram areas (a) and (b), an editor area (b), a dynamic run-time state visualization area (d), a terminal area (e), and an input/output console area (d). The editor area displays the active Java code a student user is working on, and by selecting a tab, the user can create multiple Java files and add them into a project. When a new class is added, the default code, representing the basic structure of a class, is generated for the user to start immediately changing the existing code. For each line of code, its corresponding class is highlighted in the compact class diagram in (b) and the functional information of data is synchronized in the run-time state visualization in (d). Three sets of UML diagrams are generated: (a) one for the active Java program in the editor, (b) one for the whole project, and (g) one for a detailed diagram containing all the information related to the current project.

When an enlarged icon  in area (b) is clicked, a detailed UML diagram for the project is illustrated in a separate window (g). The detailed UML diagram in the new window shows all classes created in the project. Each class is outlined as a rectangle containing three sections to represent class name, attributes, and methods. The diagram also visualizes all relationships among classes, such as association, inheritance, and interface. This is to give student users a better understanding of a program's design, and it will become helpful, especially, when the program gets larger with more classes added to the project.

The corresponding Java class to an active class in the editor window is highlighted in the compact UML diagram (b) to dynamically synchronize it with the source code and the run-time state of the program execution (d).

(a) View Class Diagram: Shows a simple class diagram for 'Main' with a method '+main(String[] args):void'.

(b) Class Diagram: Shows a hierarchy where 'Shape' is the superclass for 'Rectangle', 'Cylinder', and 'Sphere'. 'Main' is a separate class.

(c) Source Code: Shows the Java code for 'Main.java'.

```

1 // PolyShape Project
2
3
4 public class Main
5 {
6     public static void main (String[] args)
7     {
8
9         Rectangle deck = new Rectangle(20, 35);
10        System.out.println (deck + " and its area is " + deck.area());
11
12        Sphere bigBall = new Sphere(15);
13        System.out.println (bigBall + " and its area is " + bigBall.area());
14
15        Cylinder tank = new Cylinder (10, 30);
16        System.out.println (tank + " and its area is " + tank.area());
17    }
18 }
19
20
21
22
23
24
25
26
27
28
29
30
31

```

(d) Visualize Program Execution: Shows the runtime state. 'main:67' contains 'deck' and 'bigBall'. 'Rectangle instance' has length: 20.0, width: 35.0, shapeName: "Rectangle", testVariable: 100. 'Sphere instance' has radius: 15.0, shapeName: "Sphere", testVariable: 100.

(e) Terminal: Shows the command-line execution of 'javac Main.java' and 'java Main', followed by the program's output.

```

Welcome to JC Web Console!
JC$cd /var/www/html/ProjectDir/sample@gmail.com
JC$cd '/var/www/html/ProjectDir/sample@gmail.com/PolyShape(2015-12-26)'
JC$javac Main.java
java Main
JC$java Main
Rectangle of length 20.0 and width 35.0 and its area is 700.0
Sphere of radius 15.0 and its area is 2827.4333882308138 and its area is 2827.4333882308138
Cylinder of radius 10.0 height 30.0 and its area is 9424.77796076938 and its area is 9424.77796076938
JC$

```

(f) Console: Shows the program's output.

```

Output
Rectangle of length 20.0 and width 35.0 and its area is 700.0
Sphere of radius 15.0 and its area is 2827.4333882308138 and its area is 2827.4333882308138

```

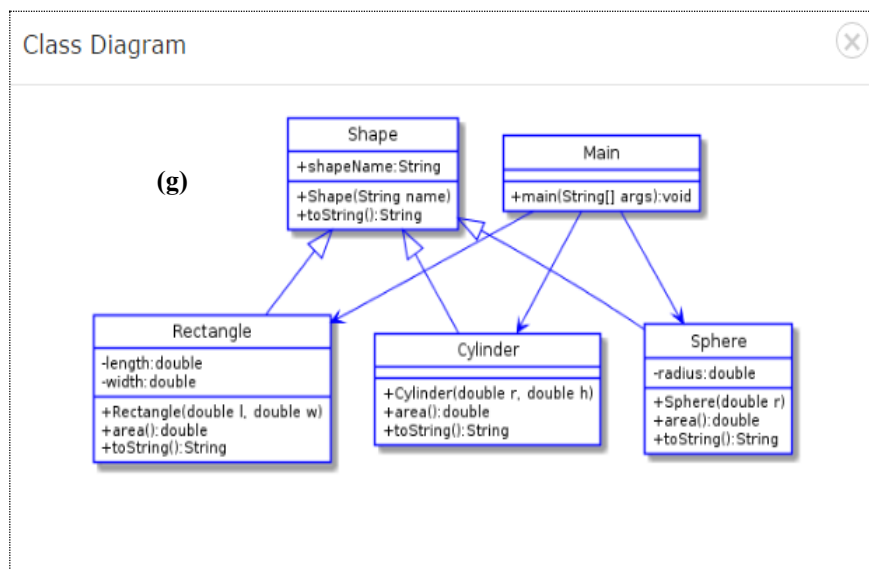


Figure 3. User interface of JavelinaCode

The terminal area (e) is for users' interactive input and output operations, and the console area (f) is to get input directly from the user if the user runs 'Visualize Program Execution' without compiling the project. The console accepts 'stdin' property containing all the standard input fed for in memory compilation and used for run time visualization of program execution.

3.5 Modeling Example

The modeling example of JavelinaCode is presented in Figure 4. In this example, it is clearly noted the UML notation and run-time state are instantly changed when a line of code changes. The running example involves five classes having inheritance and polymorphism relationships: Shape as a parent (super) class and Rectangle, Cylinder and Sphere as children (sub) classes.

As illustrated in (a) of Figure 4, after the Rectangle, Cylinder, and Sphere classes are defined with inheritance and an object of Rectangle is created in the testing Main class, its corresponding UML class diagram, with the Main highlighted in yellow on the left, indicates the inheritance relationship between Shape and Rectangle and association between Main and Rectangle. Highlighting Main in yellow means Main is an active Java class in the editor window. The values of the rectangle object are assigned to its instance variables, which are also visualized in the run-time state on the right.

As shown in (b) of Figure 4, when a new Sphere class is added and its object is created in the editor, the UML diagram instantly changes, showing the new inheritance relationship between Shape and Sphere and the association between Main and Sphere. The values of the instance variables of the Sphere object are also illustrated in the dynamic visualization.

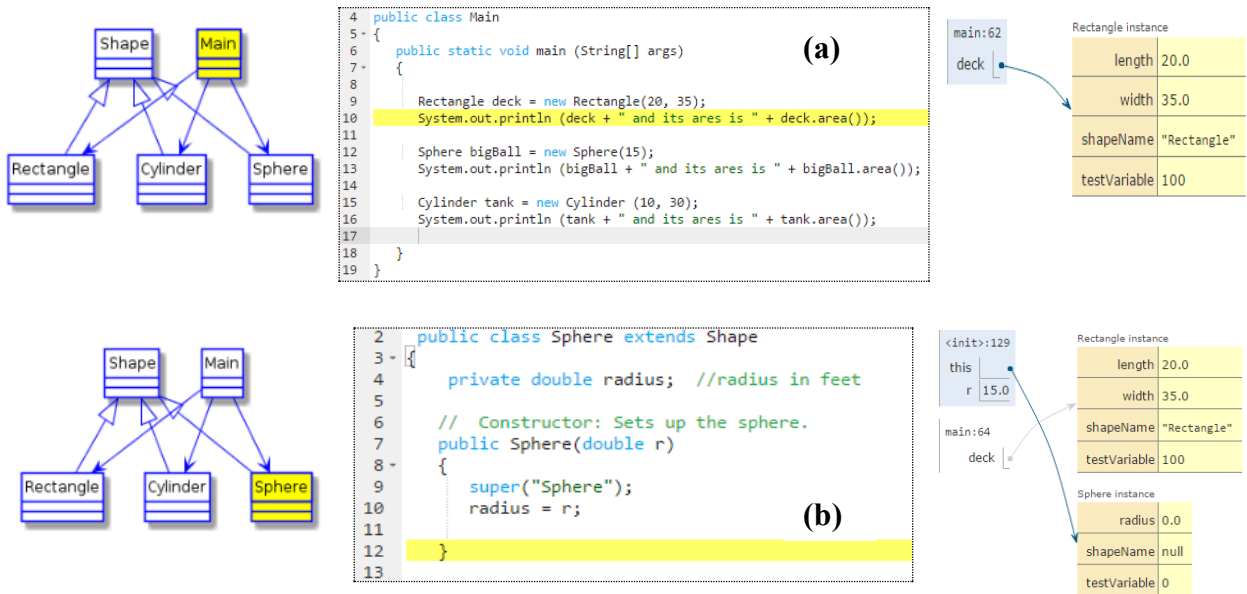


Figure 4. Modeling example linking source code with static and dynamic visualization

3.6 Mechanics

JavelinaCode provides a complete environment for programming tasks in Java. This section gives all aspects of the task, including creating, editing, compiling, executing, and visualizing programs within the environment.

3.6.1 Getting Started

LogIn: To get started, a user must access one of the web URLs, <http://javelinacode.org>, <http://javelinacode.ocom>, <http://www.javelinacode.org>, and <http://www.javelinacode.ocom>, and complete a sign up form with an email ID and password (first time user). After signing up, the user's credentials are saved in a database and the user can log in to the system (Figure 5).



Figure 5. Login window

Under this ID, project files are saved on the server side, when a Java project is created with multiple classes, and log files of the users' behavioral patterns in programming will be recorded for analytics.

3.6.2 Editing with Ace Editor

Ace, an embedded open-source code editor, is fully integrated into the environment [14]. The Ace editor is easily embeddable in any web application built on JavaScript, and it comes with a lot of advantage:

- Syntax highlighting makes it easy for users to differentiate the key words, variables, and methods, which makes the code more readable and easier to understand.
- The look and feel of the editor can be adjusted, as it supports over 20 themes.
- It also supports automatic indentation, which can help users to maintain the code.

- It provides the facility to handle large files, almost up to four million lines of code. This a very important feature, as novice programmers add their Java classes in the same file, rather than creating different files.
- Block highlighting makes it easy to understand the scoping issues, as the editor automatically highlights the matching brackets when clicked on either the start or the end.
- Users can drag and drop the source code from other sources, which makes it easy for them to test the code.

A class just created is given basic elements of a Java class, including a class name, a private instance variable, and a public method. From the default code representing the basic structure of the class created, users can instantly edit the code and make changes to the code.

3.6.3 Creating, Opening, and Deleting a Project

The file menu contains multiple options, including creating a new project, opening an existing project, deleting a project, and signing out.

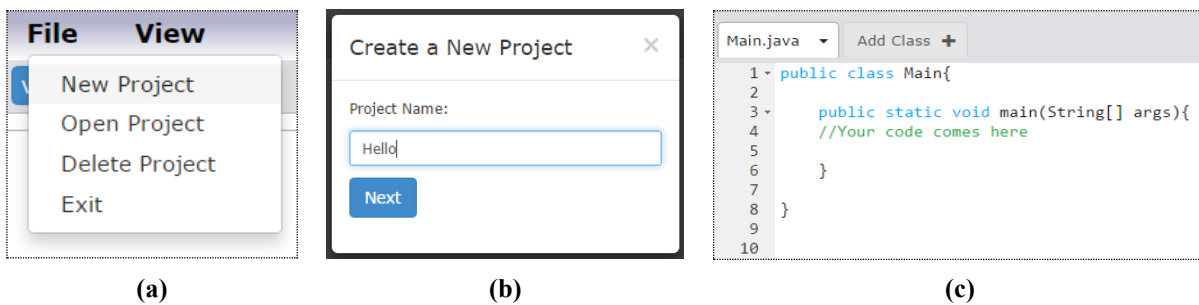


Figure 6. Creating a new project

Creating a new project: To create a new project, users can select the ‘New Project’ option from the ‘File’ on the top menu bar. This prompts the user to enter the project name. Once the user enters the project name, a new project is created with Main.java (see (a), (b), and (c) of Figure 6).

When a new project is created, the `createProject` method in `ideIntegratedWithAce.js` is triggered, and it validates its project modal and creates an instance of the Ace editor for Main.java. This method internally is chained with a few other method calls, such as `createProjectFolder` method, used to build a unique project name for the user, based on the time stamp. This call is followed by `createClassUI` method in which an editor instance is initiated for each JAVA class, and the initial properties, such as theme, mode, and height of the editor, are set.

Opening an existing project: When ‘Open Project’ option is chosen from the FILE menu, `openProject` method is invoked, so the user can select a project listed and open the project (see (a), (b), and (c) of Figure 7). This connects to the server and fetches the list of all the projects saved for the user from the server. Once the project is chosen from the list, all the JAVA classes associated with the project are listed. After the project is chosen, the project name and list of classes of the project are sent to open Java files to the editor window.

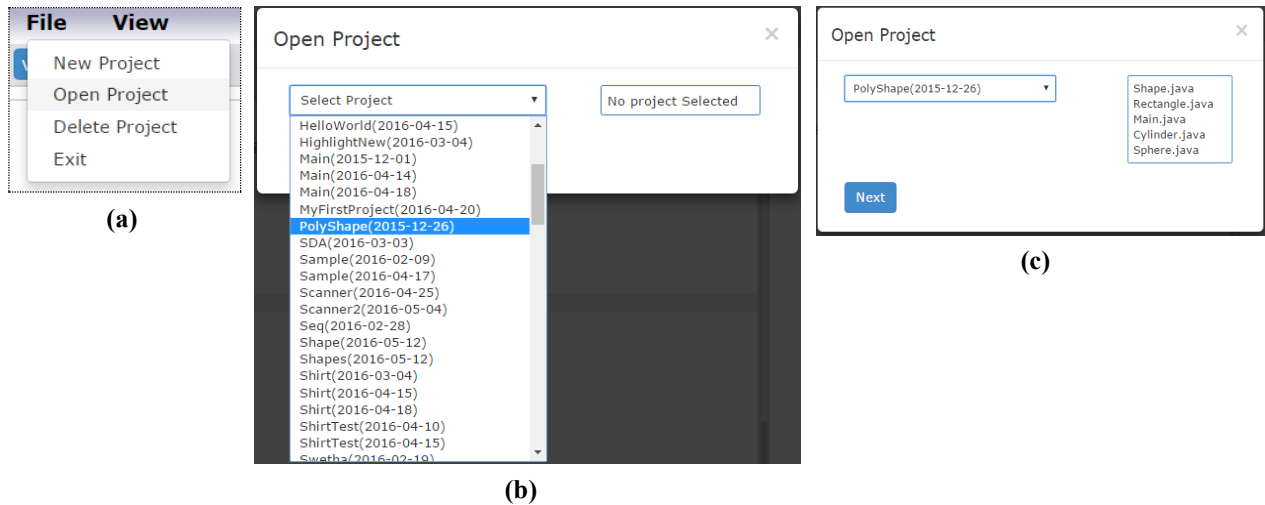


Figure 7. Opening an existing project

Deleting a project: When the ‘Delete Project’ option from the FILE menu is selected, all the projects he user has created are listed. When a project is chosen to be deleted, the user is prompted for a confirmation. Once the ‘OK’ button is clicked, the project files are removed from the server folder that belongs to the user ((a), (b), and (c) of Figure 8).

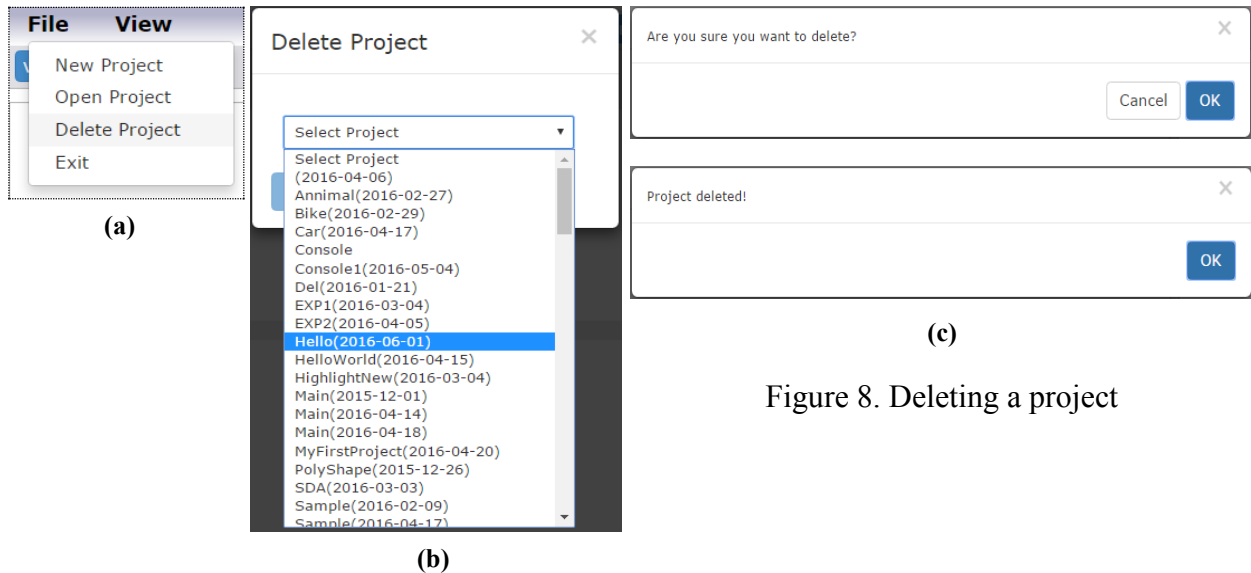


Figure 8. Deleting a project

3.6.4 Adding, Closing, Renaming, and Deleting a Class

The Ace editor can display multiple Java classes, using bootstrap tabbing. Using these tabs, a user can add, close, rename, and delete a Java file. By clicking ‘Add Class’ tab, the user is prompted to enter the class name (Figure 9). After the name is entered, it is validated, and an instance of the editor is created for the class. After the user interface is created for this new class, the Java file is created on the server, with the class name, and saves the content when ‘Save’ button is clicked.

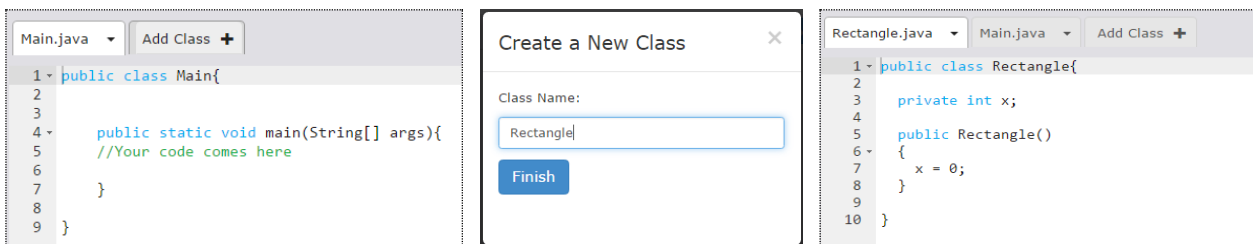


Figure 9. Adding a class

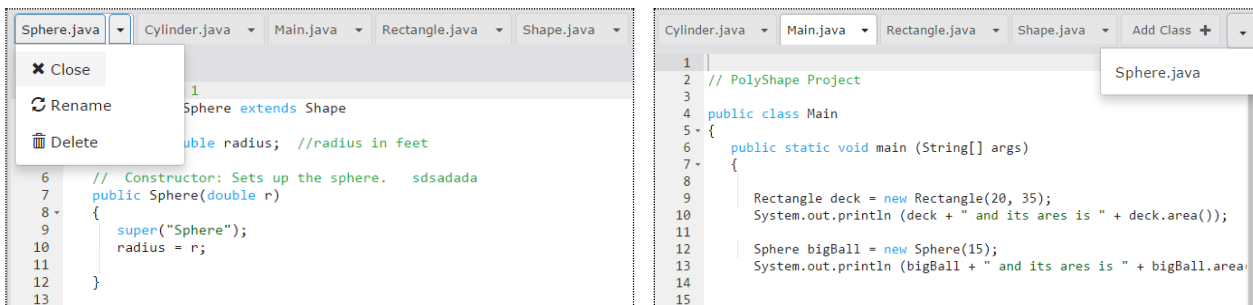


Figure 10. Closing a class

When the user closes a file, the tab for the Java file is closed, and the name of the closed file is listed with a small arrow on the right corner of the editor (Figure 10). This is seen only when the user hovers on the arrow. When the user renames a file, the user is prompted to enter a new file name (Figure 11). When a new name is given, the tab name is changed, and the same is done on the server. When ‘Delete’ is selected, the user is asked for a confirmation (Figure 12). When

the user confirms to delete the file, that tab is deleted from the user interface, and the file is also deleted from the server.

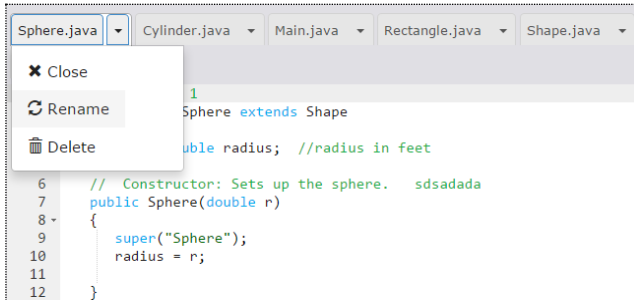


Figure 11. Renaming a class

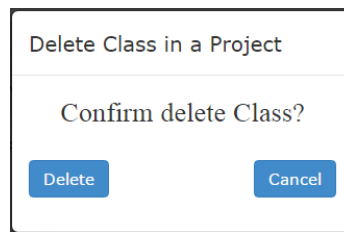
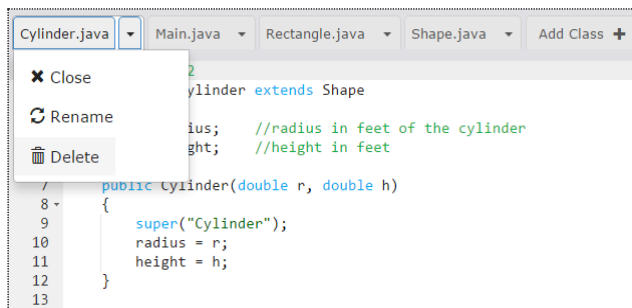
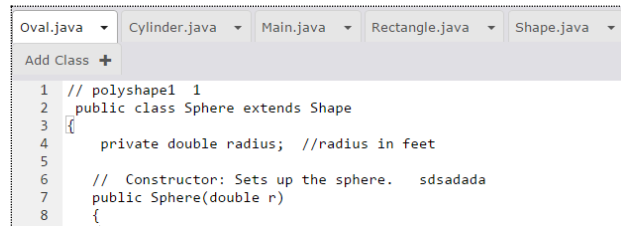
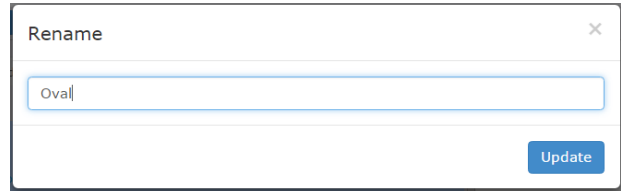


Figure 12. Deleting a class

3.6.5 Saving, Compiling, and Running a Project

After editing the class, the whole project can be saved, compiled, and executed. Clicking the ‘Save’ button triggers the method that collects the content of all the Java classes in the project and saves them in the project folder under the user. Clicking the ‘Compile and Run’ button compiles and executes the project. This involves a complete dependency and association analysis, and the compilation of all classes in the project is done. If errors are detected by the compiler, error messages are displayed in a pop-up window, so the user can fix the error in the

appropriate manner. The output of the results is displayed in the ‘Terminal’ window at the bottom of the screen (Figure 13).

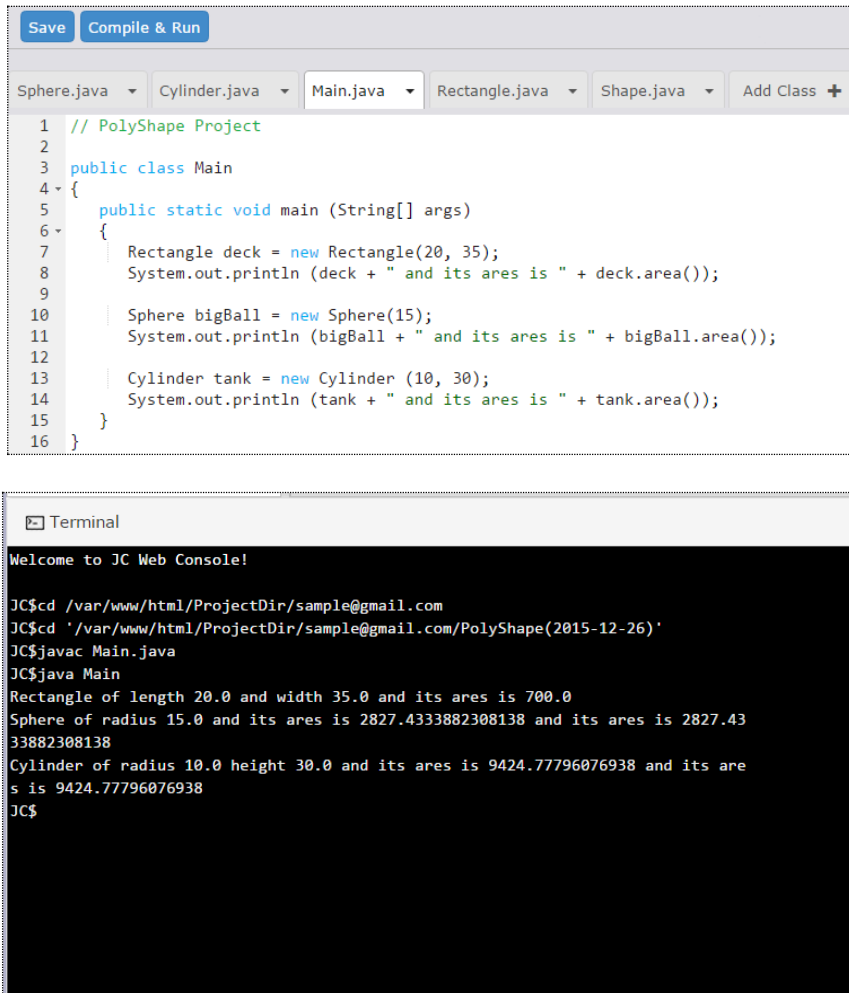


Figure 13. Output of the project in the terminal window

3.6.6 Viewing UML Class Diagrams

There are three types of UML class diagrams supported by JavelinaCode: an individual class diagram, a compact class diagram, and a detailed class diagram.

An individual UML class diagram, as presented in Figure 14, is created from the active Java class currently on the editor window. From the Java source code, its textual input is generated by the expressions defined in Chapter 4. Then, the input is sent to the Plant UML API, which returns the image of the class diagram. The diagram is represented as a box that contains three sections for the name of the class, the attributes of the class with their data types and access modifiers (+, -), and the list of methods, with return type and parameters, with which the class can operate.

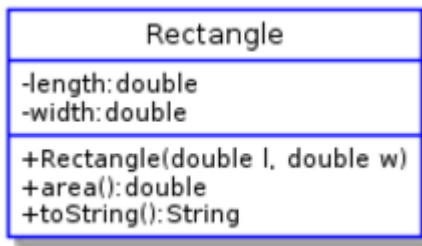


Figure 14. Example of an individual class diagram

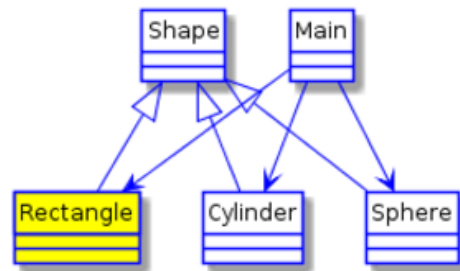



Figure 15. Example of a compact class diagram

A compact class diagram, as presented in Figure 15, is generated from the project. The diagram includes all classes for the project and the relationships among them, including inheritance and associations. The plant UML textual input described in Chapter 5 is generated from the Java classes in the project. When this text is given to the Plant UML server as an input, it generates an image, which includes the relationship between all the classes in that project.

It is easily noticed, in the example, there are five classes (five boxes: each box represents a class), including Main, Shape as a parent (super) class of Rectangle, Cylinder, and Sphere classes, and Main acts as a client to create objects of the children classes to communicate with. Rectangle class is highlighted in yellow, which means a user is editing and working on that class

in the editor window. When the user switches to another class in the editor window, that switched class is active in the editor window and highlighted in the compact class diagram.

A detailed class diagram for the same project is presented in Figure 16. This diagram is displayed in a separate pop-up window when the user clicks the ‘Enlarge’ button  on the left side of the initial interface or selects the ‘View Class Diagram’ button on top left. This button is only enabled after visualizing the project. On click, the full details from the JSON output are collected and sent to the Plant UML server to get a class diagram image.

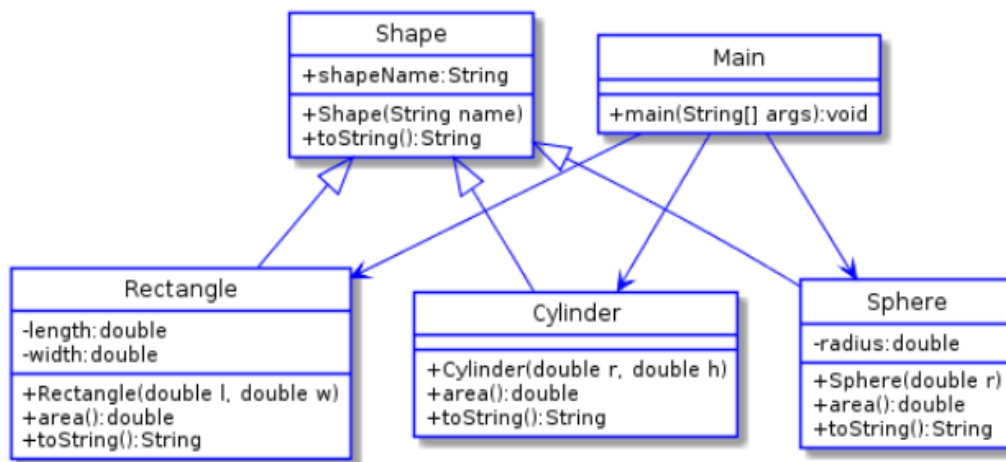


Figure 16. Example of a detailed class diagram

The diagram shows the classes of the project, their relationships, including inheritance and association, the methods and attributes of the classes. Each attribute and method has its own detailed information, such as a data type, an access modifier, a return type, and parameters. The diagram provides an overview of the system, so users can build a conceptual model of the system.

3.6.7 Visualizing Program Execution

By clicking the ‘Visualize Program Execution’ button, users can begin monitoring the run-time state of program execution line by line. In the source code display, a highlighted yellow line bar indicates the line just executed. When the yellow bar moves forward or backward by clicking one of the buttons, Back or Forward, in Fig 17, the representation of objects and their references at the current execution point is visualized. The example shows objects (instances of Rectangle and Sphere) and stack frames at the current execution point, with the stack growing downward when a new object is created. Each frame shows the name of the instance and a list of instance variables and values.

When the button is clicked, the system connects to the Java Visualizer API installed on the server, which triggers each tab in the editor, so the Ace API can generate the DOM elements for each line of the code in all the Java classes. This is required to ensure the highlight in the code is synchronized with the run-time visualization.

3.7 Anticipated Benefits

For the best practice of OOP and OOD, JavelinaCode is designed with these principles: 1) Easy to Access, 2) Easy to Use, 3) Easy to Understand, 3) Source Code and User Centered, 4) Static Visualization of Structural Information of a Program, 5) Dynamic Visualization of Functional Information of Data, 6) Static and Dynamic Visualization together, 7) Synchronized Multi-View with Source Code, and 8) Structural and Functional Feedback in real Time.

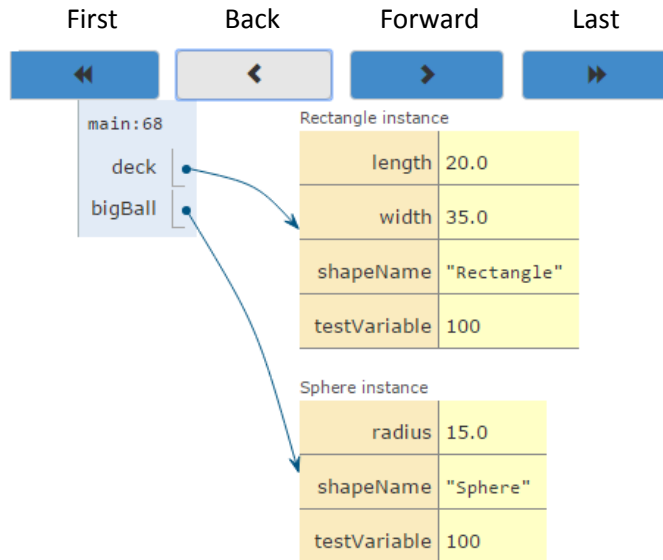


Figure 17. Run time state of program execution

JavelinaCode is platform-independent. Students use a web browser to develop and run a Java program, with no required software or plug-in installation. They can program anywhere and anytime, using their laptops, desktops, tablets, or mobile phones. The project files created by a student are saved on cloud storage. A student needs no memory system to keep and manage data. Students are free from concern of continuous version changes and evolutions to the Java language, IDEs, plug-ins, and operating systems. This provides a great deal of accessibility and usability to the program development environment. JavelinaCode will be further developed as mobile applications that runs on both iOS and Android.

JavelinaCode is both source code and user centered. When a student writes a line of code, the corresponding structural information of the program is dynamically linked with the two sets of UML class diagrams, and functional information of data is synchronized in the run-time state visualization. Real time feedback of the current line of the code will be immediately given to the

student. The links will be highlighted, when the user hits the 'Enter' key, after completing the line, such as variable declaration and/or initialization, method declaration, or expression. This will greatly help students establish the mental model of program execution [12]. Figure 3 illustrates the modeling example of linking and highlighting a line of code with its static and dynamic changes in visualization.

The main contributions of JavelinaCode are: a) Students use a web browser to run a Java program, with no required software and plugin installations or configuration on a local computer. They can program anywhere and anytime, using their laptops, desktops, tablets, or mobile phones; b) Students need no memory system to keep and manage project files. They are free from concern of continuous version changes and evolutions of the Java language, IDEs, plug-ins, and operating systems; c) JavelinaCode will help students establish the mental model of program execution, with the source code of a program; d) With the synchronized multi-view real time visualization with source code, JavelinaCode is expected to reduce students' cognitive workload in Java programming and to enhance comprehension of the OO concepts, such as inheritance, polymorphism, and OO design.

4 Synchronized Static and Dynamic Visualization

This chapter describes two open source visualization tools based upon how these tools are utilized to transfer a textual input to a static or dynamic visualization. As a similar approach was adapted to merge static information with dynamic information in generating visualizations [41, 42, 43], in JavelinaCode, Plant UML [26] is deployed for a static state of a Java program in a UML class diagram and Java Visualizer [27] is used for a dynamic run-time state of the program execution.

4.1 Plant UML

Plant UML is a web-based open-source visualization tool that allows users easily to create UML diagrams from a plain textual description [26]. Various UML diagrams, such as a sequence diagram, a use case diagram, a class diagram, a state diagram, and an object diagram, can be generated using an intuitive textual language.

Table 1. Symbols used in Plant UML to draw a class diagram

OO concept	Symbol or keyword	OO concept	Symbol or keyword
Extension	< --	Composition	*--
Aggregation	o--	Field or method	:
Static field or method	{static}	Abstract method	{abstract}
Abstract class	abstract or abstract class	Interface	interface

In order to draw a UML class diagram, Plant UML defines notations, symbols, and keywords for relations between classes, labels on relations, static fields and methods, abstract classes, and interfaces as presented in Table 1. In drawing diagrams, Graphviz/Dot is used to compute node positions. Once Graphviz/Dot generates a simplified Scalable Vector Graphics (SVG) output from the users' textual input, the SGV data is parsed and the drawings of the diagrams are completed. Figure 18 illustrates an example of a class diagram generated by Plant UML based on its textual input with the symbols and keywords.

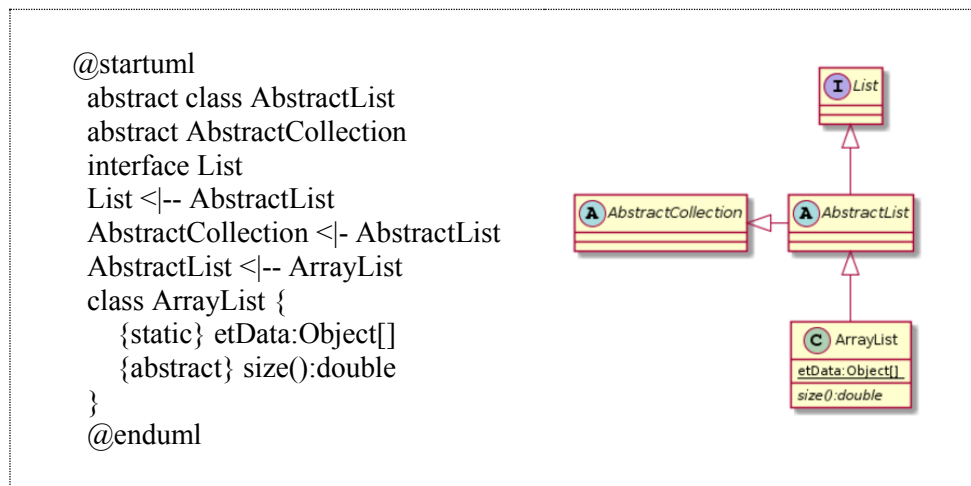


Figure 18. Textual input and its class diagram by Plant UML

4.2 Java Visualizer

Java Visualizer is a web-based program visualization tool [27], which illustrates the dynamic run-time state of a Java program by stepping forwards and backwards through program execution. It is based on 'Online Python Tutor,' a Python visualizer [6, 28], whose purpose is to help programmers learn programming better and to understand what happens when each line of source code executes.

Java Visualizer readapts and uses the Javascript frontend from the Python visualizer and replaces the backend with Java jail that runs in a sandbox. The backend installation consists of safeexec, a safe execution environment. The safeexec provides a general-purpose sandbox environment which safely executes user programs and prevents any malicious users from causing troubles or mistakes that can damage a server. The Java jail serves as a chroot (changed root) for executing Java programs, and TracePrinter, a Java package, is used to print the traces of Java programs as the results in JSON format as they execute.

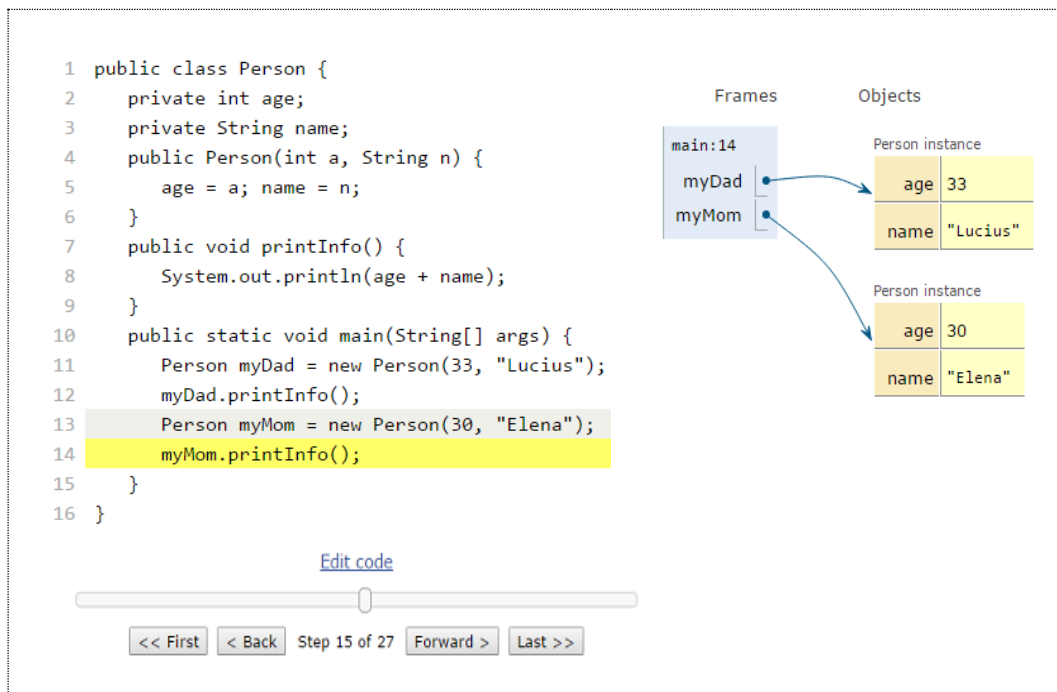


Figure 19. Visualization of Java program execution by Java Visualizer

A Java feature example of a class by Java Visualizer is visually represented in Figure 19. In the source code display, a highlighted grey line bar indicates the line that has just been executed (line # 13 in this example), and a yellow line bar indicates the next line to be executed (line # 14). When the gray bar, along with the yellow bar, moves forward or backward by clicking one of the buttons (First, Back, Forward, Last), the representation of objects and their references at

the current execution point is visualized. Although this example involves three classes, the visualizer deals with them as a single main class with multiple inner classes and the highlighted bars move forward and backward between classes in the class when objects are used.

4.3 Static and Dynamic Visualization in JavelinaCode

JavelinaCode is a web-based educational programming environment [40, 51] that runs in a web browser. It is the only web-based IDE that supports the synchronized visualization of both static and dynamic aspects of Java source code. Plant UML and Java Visualizer are customized and integrated into the JavelinaCode interface to help students better understand the structure of a Java program and the behavior and interactions of objects. A static aspect of the source code is visualized, using a customized Plant UML class diagram, and a dynamic aspect of the program execution is visualized, using a customized Java Visualizer. In both cases, program execution happens in memory. All Java files made from the editor for a project are merged into a single class file, which will serve as a main class. Other classes in the project will be inner classes for the main class.

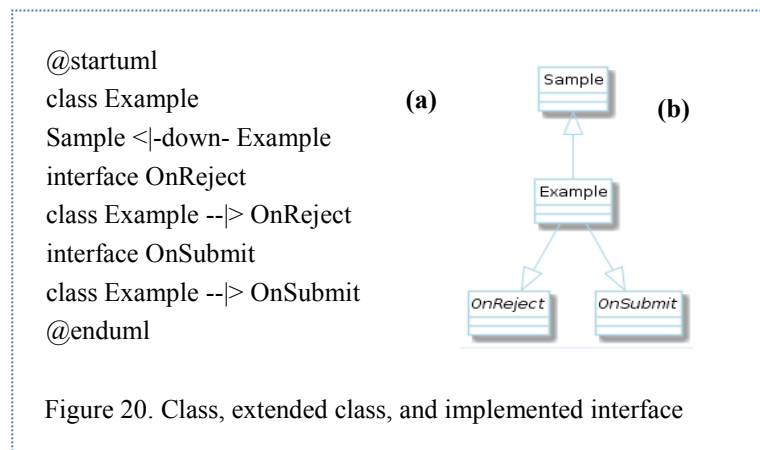
4.3.1 Customized Plant UML for Static Visualization

To draw a UML class diagram and to integrate it into JavelinaCode, textual input from the single Java file is produced with the four different regular expressions defined below and used to detect classes, extended classes, implemented interfaces, dependencies, methods, and variables. With the customized version of Plant UML in JavelinaCode, the colors of relations, borders, and backgrounds are also changed in the UML class diagram to match the ones used in the dynamic

visualization, and the circles of classes and interface are hidden to follow standard UML notation.

Class, Extended Class, and Implemented Interface

$[\backslash w.]^* *(:public|final) *(abstract) *(:public|final) *(class|interface|enum|@interface)$
 $+([\backslash w. < >,]^*?)(?: +extends+([\backslash w. < >,]^*?))?(?: +implements +([\backslash w. < >,]^*?)) *\\{ \quad (1)$



The regular expression (1) is defined and used to detect classes, extended classes, and implemented interfaces. Since Java matcher returns the pattern matches, all the values can be found in groups using this regular expression. The groups give us the important values, like class name, parent class name (if class declaration has the ‘extends’ keyword), and interface name (if class declaration has the ‘implements’ keyword).

The first keyword when defining a Java class is always public, which can be used to find the first line of the class, and then check whether it is an abstract class or not. If a class is extending another class, the ‘extends’ keyword is searched. The word followed by ‘extends’ keyword is the extended class. If a class implements an interface a keyword ‘implements’ is searched. Since a

class can implement multiple interfaces, all the words after the ‘implements’ keyword separated by a comma are found. All the comments are removed from the source code, or else, they can cause issues while using the regular expression.

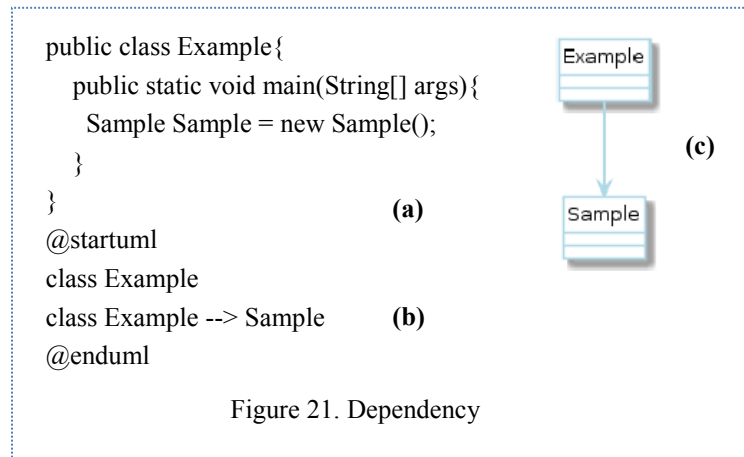


Figure 21. Dependency

If we assume the first line of a Java source code is ‘public class Example extends Sample implements OnSubmit, OnReject {’, this code is passed to the Java pattern matcher, then the matcher will provide the details, like ‘Example’ as a class name, ‘Sample’ as an extended class, and ‘OnSubmit and OnReject’ as implemented interfaces. Therefore, a Plant UML textual input for the code and its corresponding UML class diagram generated by the Plant UML are as shown in Figure 20.

Dependency

$$(new)(\s+)([^\s]+)(\s*)(\s*(\s*\s*))(:) \quad (2)$$

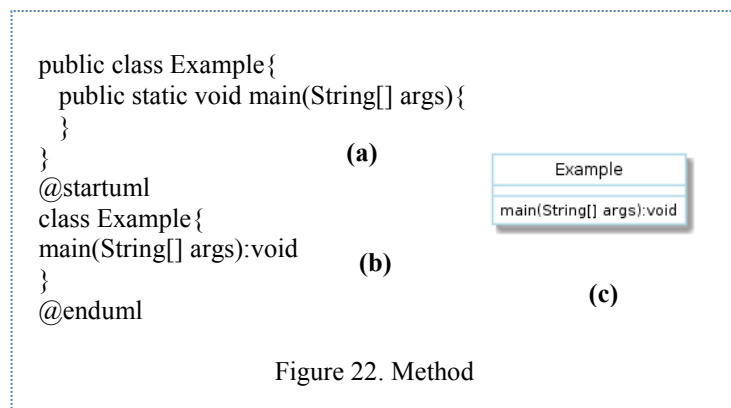
The regular expression (2) is defined and used to look for dependencies among classes. It is assumed that there is a dependency on a class if an object creation pattern happens with a ‘new’ keyword in the class. For instance, if we have the source code as in (a) of Figure 21, its Plant

UML textual input for the code and UML class diagram generated by Plant UML are as shown in (b) and (c) respectively of Figure 4.

Method

$$([\w\<|\>|\[\]]+)+\s+(\w+) *([\^]) *([\])$$
 (3)

The regular expression (3) is defined and used to find methods in a class. A method generally contains an access modifier followed by the return type, method name, and arguments. The first value in the regular expression pattern gives us a word which is an access modifier, the second value is the return type of the method followed by a space. Finally, the values between the parentheses are the arguments. This expression provides information about a return type of method and a name of the method, checks whether the method has arguments, and extracts the argument if the method has them. If we have the source code as in (a) of Figure 22, its textual input for the code and UML class diagram are as shown in (b) and (c) of Figure 22.



Variables

Private variables: private\s+(\w*)\s+([\w,]*)\s*;

Public variables: `public\\s+(\\w*)\\s+([\\w,]*)\\s*`;

Protected variables: `protected\\s+(\\w*)\\s+([\\w,]*)\\s*`; (4)

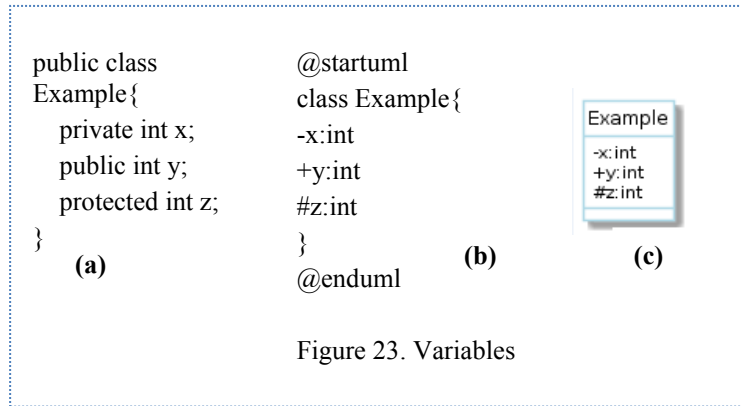


Figure 23. Variables

Regular expressions (4) are defined and used to find three different types of variables. A variable syntax generally contains an access modifier and a data type followed by the name of the variable. By using this expression, the data type and access modifier of the variable are detected. Private, public, and protected variables are denoted with '-', '+', and '#', respectively. If we have the source code as in (a) of Figure 23, its textual input for the code and UML class diagram generated by Plant UML are as shown in (b) and (c), respectively, of Figure 23.

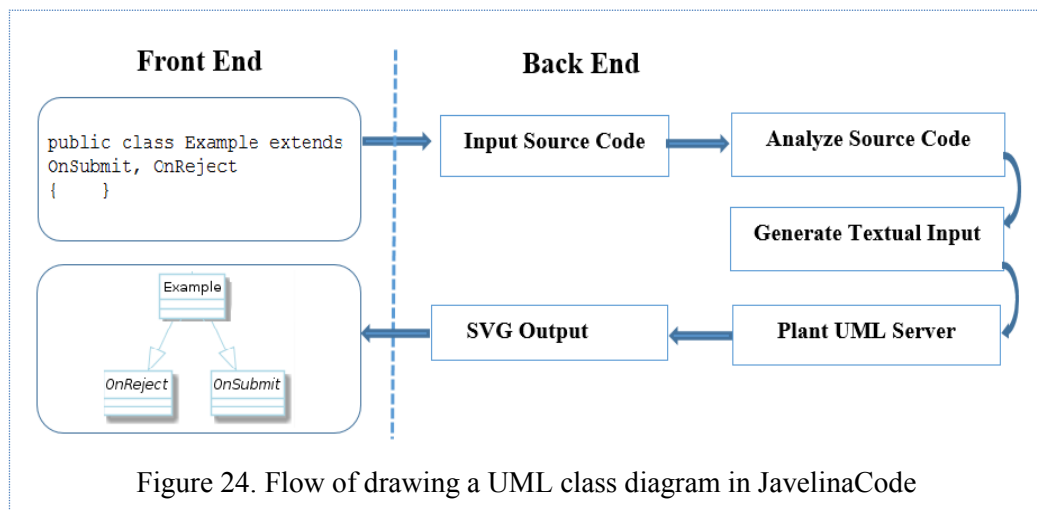


Figure 24. Flow of drawing a UML class diagram in JavelinaCode

Figure 24 demonstrates the flow of drawing a UML class diagram in JavelinaCode. A single Java source file from the frontend is processed by the expressions defined above to produce textual input that is sent to the Plant UML server to produce a Scalable Vector Graphics (SVG) output representation. For drawing diagrams, Graphviz/Dot is used to compute node positions. Once Graphviz/Dot creates a simplified SVG output representation, the SVG data is parsed and the drawings of the diagrams are completed.

4.3.2 Customized Java Visualizer for Dynamic Visualization

The original Java Visualizer is customized and integrated to the JavelinaCode interface to better support understanding of the behavior and interactions of objects. To avoid any confusion of using two colored bars, only one highlighted bar in yellow is used to indicate the line that has just been executed. As shown in Figure 8, in the editor window of JavelinaCode, there are tabs for multiple Java class files created for a project and one active class is open. The color of the tab of the active class in white that has a current line of the code being executed is distinguished from others.

The running example involves five classes having inheritance and polymorphism relationships: Shape as a parent (super) class and Sphere, Rectangle, and Cylinder as children (sub) classes. After executing line 4 in (a) of Figure 25, when a new object of Sphere is created with the instant values of the object assigned, the yellow bar jumps to the constructor of the Sphere class which is open in the active editor window in (b) of Figure 25. When the constructor calls the super class's constructor in line 8, again the highlighted bar will jump to Shape class

which is a parent of Sphere. The values of the instance variables of each object are accordingly illustrated in the visualization area.

(a)

(b)

Figure 25. Dynamic visualization of Java program using customized Java Visualizer

Through one step forward or backward of program execution by line, the behavior and interactions of the objects are much more easily captured and understood. Using this runtime visualization, students can clarify the concepts of tracing different classes, parameters and return values of method calls, and values of variables, and referencing multiple identifiers of the same object.

Figure 26 demonstrates the flow of generating a run-time visualization of a Java program. A single Java source file from the frontend is taken to the backend server and compiled to bytecode in memory. An instance of the Java Virtual Machine (JVM) is launched, the bytecode is loaded to the JVM, and the program is executed under supervision of the Java Debug Interface (JDI), which records the program's run time state for every executed line.

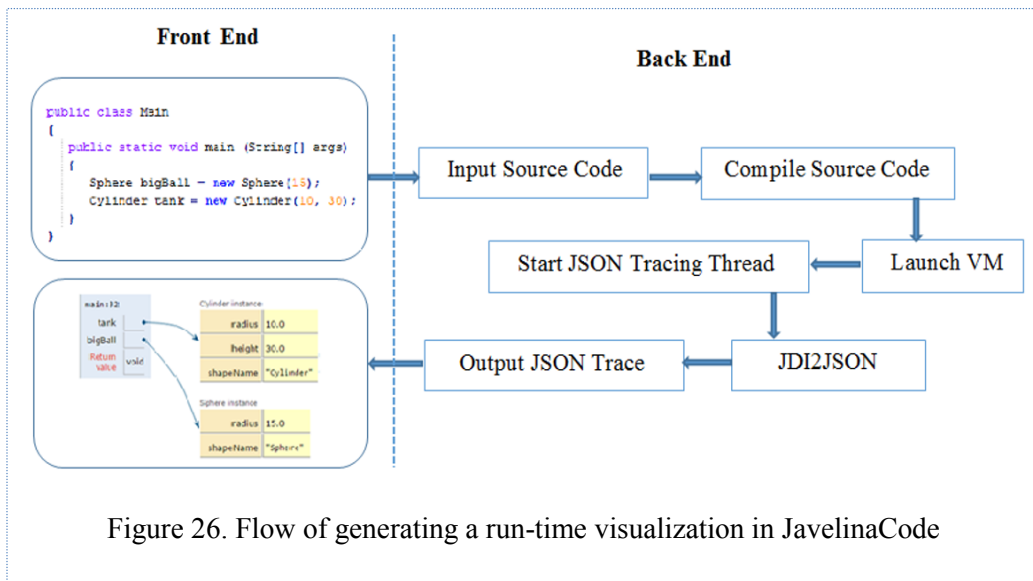


Figure 26. Flow of generating a run-time visualization in JavelinaCode

A JSON Tracing Thread is used for the event handling loop, and then JDI2JSON converts everything to a text output JSON trace. The output JSON trace contains the dynamic run time state of all the objects, methods, and variables of the Java program, which is converted to a visualization generated by Javascript 3D library in the frontend.

Environment

On the AWS cloud server, Java Visualizer is installed on the Ubuntu operating system with a safeexec environment along with Java jail that acts as Chroot. To execute a Java program and

collect the trace of its execution, the Trace program of Java platform debugger architecture is used. Then the trace runs the Java program and generates an output of its execution.

Using `EventRequest`, a notification of an event such as `BreakPointRequest` can be requested. Whenever an event occurs, an event set is placed on an event queue and then the events are managed by `EventRequestManager`. There are different kinds of event sets, which are enabled like `ThreadStart`, `ThreadEnd`, `methodEntry`, and `methodExit`. `EventRequest` is a parent class of `StepRequest`. When a step request occurs in the Java virtual machine, a notification is requested. All the step requests are also placed in the event queue and its trace is printed. Java Debug Interface (JDI) is used to get access to the virtual machine. JDI provides a control of the virtual machine state including information of classes, arrays, primitive data types, instances of the classes, etc.

Using `ToolsProvider`, an instance of Java compiler is obtained, and using that object Java files are compiled. If the compiler generates errors, error messages will be sent to an error listener. In the application of `JavelinaCode`, `RAMClassFileManger` is used as a file manager, which gives the compiler an instance of `RAMClassFile` so that the compiler can write the byte code. Upon the successful compilation, the byte code can be retrieved from the `RAMClassFile`.

To communicate between debugger and the target Virtual machine (VM), a connector is needed. Using a launching connector, an instance of target virtual machine is obtained and can be launched, and `MethodEntryRequest`, `MethodExitRequest`, `ThreadDeathRequest` and `ExceptionRequest` are created. Once a request is created and processed, an event set is placed on

the event queue. All the events on the EventQueue convert the execution point into JSONObject and all the JSONObjects are added to a JSONArray for trace output.

Chroot and Safeexec

Chroot acts as a virtual root folder to an application to recover a system in case the system becomes unbootable. Chroot installs another operating system inside the host system so that applications are not able to access the files in the root hierarchy. This is called as a jail or sandbox environment, which is needed for any security breach. If an application is installed in a sandbox environment, applications outside of the system can't access to the base system. Safeexec is a sandbox environment to execute programs safely inside a virtual root environment. It sets limits on CPU time and memory usage since there are multiple users simultaneously working on the same system and the CPU must be efficiently used.

The virtual root folder contains all the folders of the host system. It has a duplicate copy of the original system. In JavelinaCode, a virtual root is installed with Java jail, which has traceprinter files that execute the program in memory and print the JSON trace. The virtual root also contains its own Java installation since Java programs need to be executed on the virtual root environment not on the host system. The few important Java files which perform the operation are included: InMemory that is the main class of the application that takes the Java source code as input and generates output JSON trace, RAM Tools that uses a default file manager object and executes Java files and generates the bytecode, VMCommander that loads the byte code into VM memory and starts execution, JSONTracingThread that converts all the events from Event Queue to the JSON object and this operation works in a loop, JDI2JSON

which all the event information from is converted into the required JSON format, Compile2Bytes that converts the Java source to byte code with the help of a compiler object, RAMJavaFile that simulates the file object of Java source code in memory, RAMClassFile that simulates the file object of a compiled Java source code in memory, and RAMClassFileManager that simulates the collection of RAMClassFile in memory.

4.3.3 Case Study: yo-yo effect with Synchronized Static and Dynamic Visualization

A modeling example of a synchronized static and dynamic visualization in JavelinaCode and its detailed UML class diagram are graphically presented in Figure 27. The example simulates the yo-yo effect that causes problems and results in a data flow anomaly from method overriding and polymorphism with an overridden method ‘bounced’ up and down among levels of inheritance hierarchy [44]. In the presence of dynamic binding, a runtime visualization technique presented in JavelinaCode shows all possible executions by lines.

The user interface consists of three main components: a static UML class diagram area (compact class diagram (a) and detailed class diagram (b)), an editor area (c), and a dynamic runtime state visualization area (c). The editor area displays the active Java code a user is currently working on, and, by selecting a tab, the user is able to create multiple Java files and to add them into a project. When a forward or backward button is clicked, the line that has just been executed is highlighted in yellow in the editor window, the active Java class is synchronized in yellow in the class diagram area, and the functional information of the line is synchronized in the run-time state visualization. Two sets of UML diagrams are generated: (a) one for the active Java class in the editor and (b) the other for the whole project.

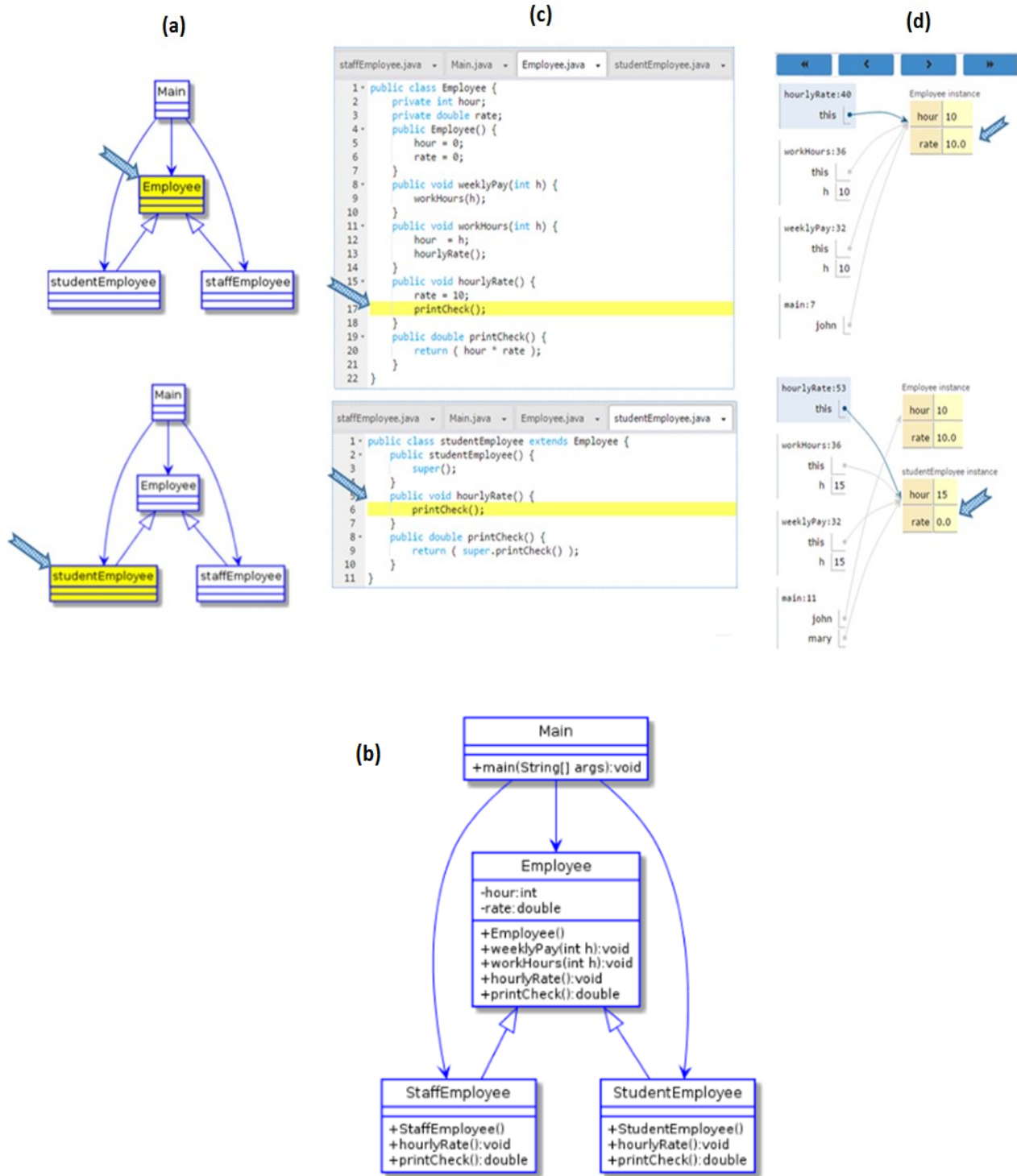


Figure 27. Modeling example of yo-yo effect

Considering the inheritance between the 'Employee' and 'studentEmployee' classes that have an overridden method 'hourlyRate()', due to the fact that the 'hourlyRate()' in the 'studentEmployee' sub-class does not have a rate assigned for an instance of that class, the student's hourly rate (mary) is '0.0' resulting in a '\$0.0' payment despite having 15 hours of work. As the line that has just been executed is highlighted in the center editor window, its corresponding class in the static UML class diagram on the left is synchronously highlighted and its dynamic run time information is also visually represented on the right.

Using this technique, users can easily trace and detect changes of the values of instances through stepping forward and backward by clicking buttons as shown in (d) of Figure 27 (top right).

5 Comparative Analysis of Educational Programming Environmental Tools

To investigate the usability of a software system, a comparison test can be used in conjunction with a validation test [45]. The objective of the comparison test is to compare designs and functionalities between competitive software products to establish which design is easier to use or learn and to better understand the advantages and disadvantages of different designs. In this study, a comparative analysis was conducted between JavelinaCode and other educational programming environment tools, including BlueJ, Jeliot 3, AguiáJ, JIVE, and jGRASP. Each of these tools was evaluated on the basis of time constraints for download and installation, complexity of the download process and tool's interface, and the provision of static and dynamic visualizations.

5.1 Comparison of Download and Install Time

This section reports a comparison analysis conducted to measure and compare the time required to download and install educational programming environmental tools including the best Java IDEs for Java programmers in 2014 [30], e.g., Eclipse and NetBeans.

JavelinaCode was compared against other tools listed above in terms of time and complexity constraints. These factors are measured and compared to determine how students will be able to efficiently use their time and critical thinking skills for writing code rather than being impeded by setting up the environments. Table 2 presents the details of the comparison of time to install

and download the environmental tools and other fundamental characteristics of each tool, which includes the version of the tool used for comparisons, whether or not the tool was stand-alone or a plugin for another tool, and the size of the tool in megabytes (MB) both before and after installation. The process of downloading and installing was done on Mac OS X Yosemite version 10.10.1 under a stable network connection on the campus of Texas A&M University-Kingsville. The specifications of the computer system used for jGRASP are OS X Yosemite 10.10.5, 2.5 GHz Intel Core i5, 4 GB 1333 MHz DDR3.

Table 2. Comparison of download and install time of programming environmental tools

	Version	OS supports	Plugin	File size (Before/After installation)	Download Time (mm:ss)	Install Time (mm:ss)
NetBeans	8.0.2	Windows, Linux, Mac OS	No	254.7 MB/ 764MB	00:49	9:54
Eclipse	4.4	Windows, Linux, Mac OS	No	173.4 MB/ 221.5 MB	00:34	N/A
BlueJ	3.1.5	Windows, Linux, Mac OS	No	169.8 MB/ 338.6 MB	00:32	00:07
Jeliot 3	3.7.2	Windows, Linux, Mac OS	No	1.6 MB/ 2.1MB	00:02	00:02
CoffeeDregs	N/A	Windows, Mac OS	Yes (NetBeans)	8 MB / 17.9 MB	00:04	0 (Portable)
AguaJ	1.1	Windows, Linux, Mac OS	Yes (Eclipse)	N/A	00:01 (Via menu in Eclipse)	0:10
JIVE	N/A	Windows 7 or later, Mac OS X or later, Linux	Yes (Eclipse)	238.1 MB	00:10 (Via menu in Eclipse)	
jGRASP	1.9.27	Windows, Linux, Mac OS	No	5/10.6 MB	00:01	00:02
JDK	8	Windows, Linux, Mac OS, Solaris	N/A	227.07 MB	00:41	01:10

While these tools must be downloaded and installed as either a stand-alone program, or as a plugin for Eclipse (AguaJ, JIVE) or NetBeans (CoffeeDregs), this is not an issue for

JavelinaCode since it is web-based. From a user's perspective, there is no worry about hardware capability, version changes of operating systems, nor data loss. They are freed from concerns about continuous version changes of the Java language, IDEs, and operating systems.

5.2 Comparison of Quality of User Interface and Aspects of Visualization

Additional comparison tests were conducted to identify any difficulties that student programmers encounter to start programming under these environments, and to compare aspects of static and dynamic visualization provided by the tools. The tests are to present the differences between JavelinaCode and the tools listed above to demonstrate how JavelinaCode improves students' understanding of OOP and OOD concepts.

Each tool was compared on the quality of user interface in terms of fundamental design and the means of handling two Java projects, PolyShape and yo-yo problem projects. The Polyshape project is considered as a relatively easy project while the yo-yo problem project is a relatively difficult one. The PolyShpe project has five Java classes including a Shape, Rectangle, Cylinder, Sphere and Main class. It introduces some fundamental class hierarchy with the Shape as a parent class, others as children, and Main as a client. It also introduces a polymorphic behavior with a overridden method, `area()`, to calculate an area of each shape. The yo-yo problem project creates a data anomaly that occurs when its execution bounces up and down the class hierarchy due to unintentional values assigned in an overridden method. The project was simulated with three classes, an Employee as a parent along with a StaffEmployee and StudentEMployee that inherit variables and methods from the Employee. The `hourlyRate()` method is a overridden

method that fails to set a proper value for hourly rate for objects of both StaffEmployee and StudentEmployee, resulting in a '0' payment for those objects.

BlueJ

BlueJ's (Version 3.1.7, released in February 2016) user interface was found to be straightforward and intuitive to build a new project, write Java classes for a project, and compile the project. However, running a project was not integrated with the other features and it was required a counterintuitive process of right-clicking the Main class and selecting the main method. The program source code and output of the programs are also displayed in a separate window rather than integrated on the initial window with the Java files. Testing the PolyShape project showed the use of the primary window as a class diagram. Java classes added to the project were connected based on relationships of inheritance and association (Figure 28).

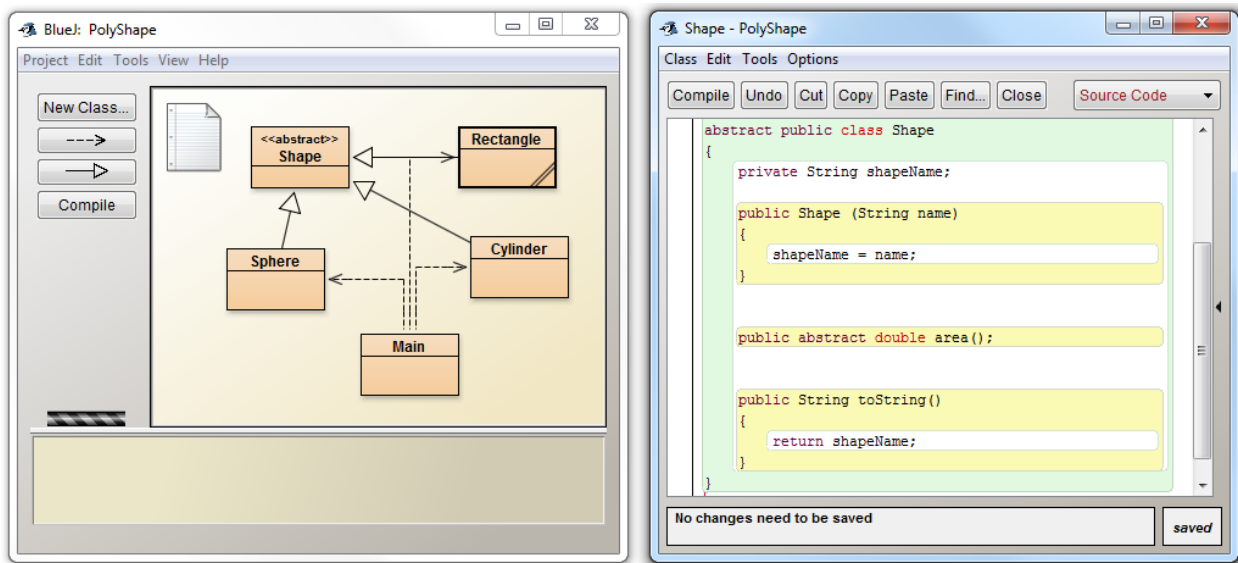


Figure 28. Screenshot of BlueJ

The main issue is that the automatically generated relationships in BlueJ were not correct and had to be corrected manually. BlueJ relies on the static visualization only, thus it seems to be very difficult to detect the source of errors with the yo-yo problem project. By setting a breakpoint at a meaningful line in the Main class, users must step through the remaining lines using the debugger to identify the problem. The debugger window shows threads, call sequences, and variables as separated text fields. This provides a dynamic approach in analyzing the source code but no run time visualization technique is employed, which makes stepping through lines of code more difficult to read and follow.

Jeliot 3

In Jeliot 3 (Version 3.7.2, released in March 2014), Java source code is displayed on the left hand side of the window and program execution is animated line by line in the theater, the animation frame, on the right hand side of the window to show how values are assigned to variables and how output is displayed to the appropriate window. The buttons to compile and execute the program and manage the animation are listed at the bottom of the window. (Figure 29).

Jeliot 3 does not support adding multiple files into a single project. All classes are specified in a single file, limiting students' understanding of how classes are distinct from one another. Execution of the PolyShape shows that the dynamic visualization, the theater, is split into four sections for methods, constants, instances and arrays, and expression evaluations. As values are specified in method calls for initializing objects of a class, those values are transferred over as

variables of the object. Different values from method calls, constants, and object variables are collected into the expression evaluation section to construct output statements.

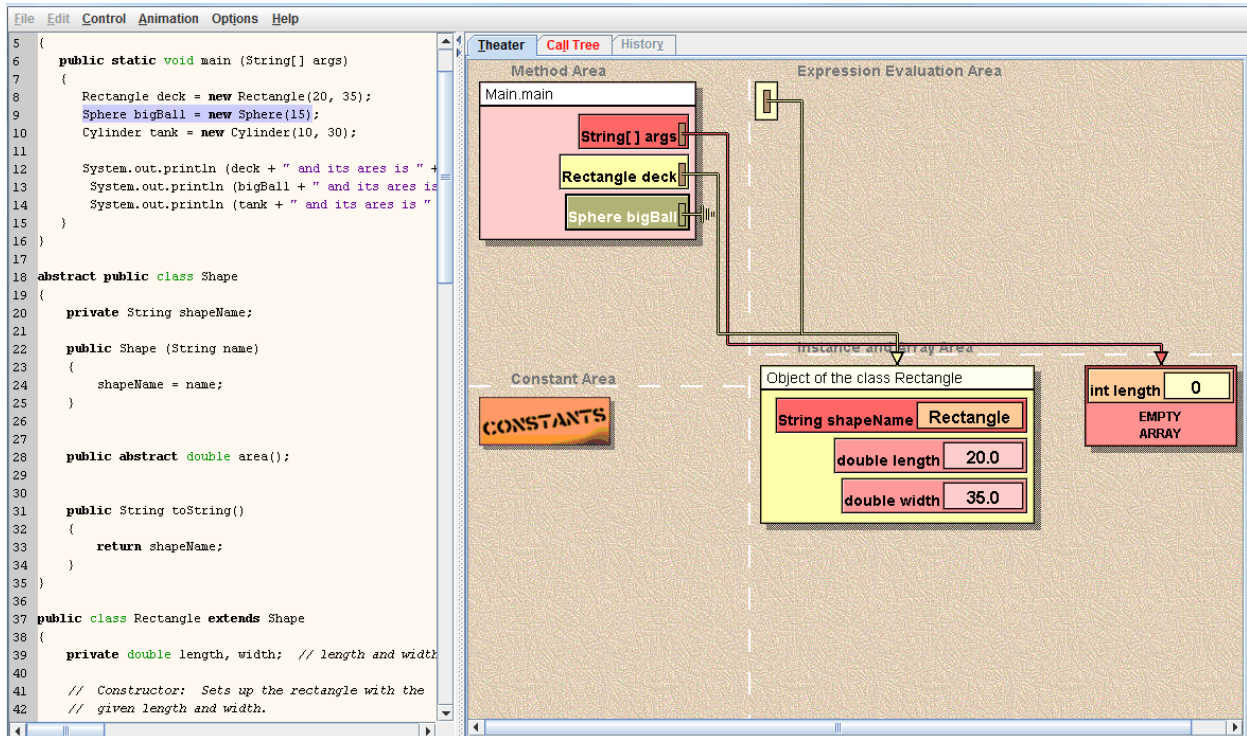


Figure 29. Screenshot of Jeliot 3 with a Theater

While running the yo-yo problem project, it is assumed that users can readily identify the likely location of program error, and those lines of code can be analyzed using the theater and stepping through each line. Then users can determine the problem based on the value of zero being passed as rate between sections of the theater. Due to the lack of a meaningful static visualization, this method of assessing the problem is dependent on the users' assumptions about the program's execution and a trial and error method.

jGRASP

jGRASP (Version 2.0.2_01, released in March 2016) allows line by line program execution through the use of the debugger and the canvas window. The canvas is displayed in a separate window from the source code and actually covers the initial window, resulting in a clustered viewing experience for users. Program output is displayed at the bottom of the initial window.

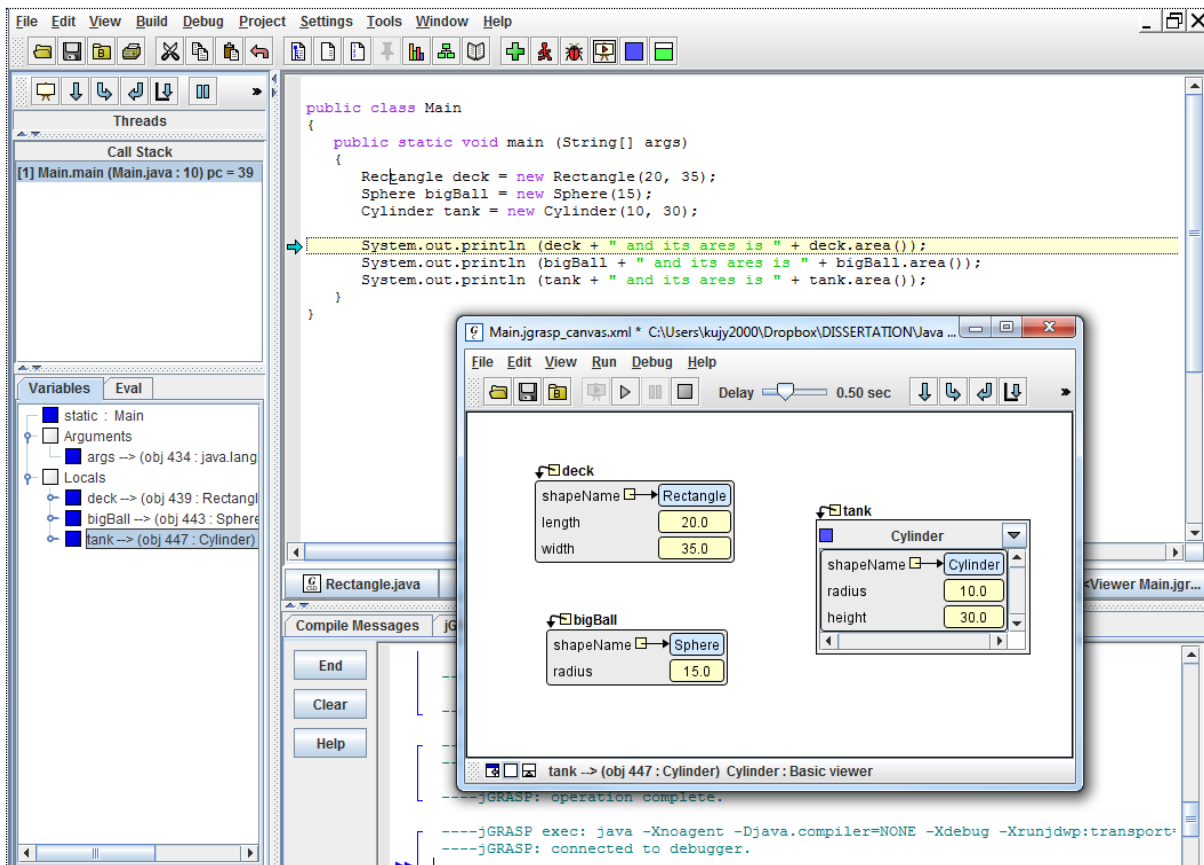


Figure 30. Screenshot of jGRASP with a Canvas window

Execution of the PloyShape project in the canvas can be paused and elements from the debugger or work bench can be dragged into the canvas window. This converts the elements to the visualization as a frame with the name of the object and its instance variables as well as the values of the variables as shown in Figure 30. Users need to drag elements from the debug frame to the canvas to check the information.

For running the yo-yo problem project, three objects created can be observed near the end of the program's execution. By dragging each of these objects into the canvas, users can observe that the object of Employee has a rate of nonzero while the objects of StaffEmployee and StudentEmployee have a rate of zero.

AguaJ

AguaJ (Version 1.1, released in October 2013) is an Eclipse plug-in designed to emphasize class instantiation. It embodies novel interaction metaphors to illustrate object-oriented programming concepts with first class representations [19]. This is made possible by introducing a graphical environment for creating and controlling classes and objects interactively [18]. From what was observed, there are windows for presenting an overview of the classes involved, holding instantiated objects of each class, and inputting a line of code for the objects (Figure 31).

Classes and objects are not bound to the runtime environment of the program. Objects are freely created and used, regardless of the original program's intent, using either the appropriate button in the interface or inputting a line of constructor code. Users can make visible the fields, private fields, and operations (variables and methods) for each object. In the context of PolyShape, various instances of Rectangle, Sphere, and Cylinder can be created independent of the original Main class. Through the implementation of multiple classes, users would be provided with an incredibly useful and flexible static visualization of code.

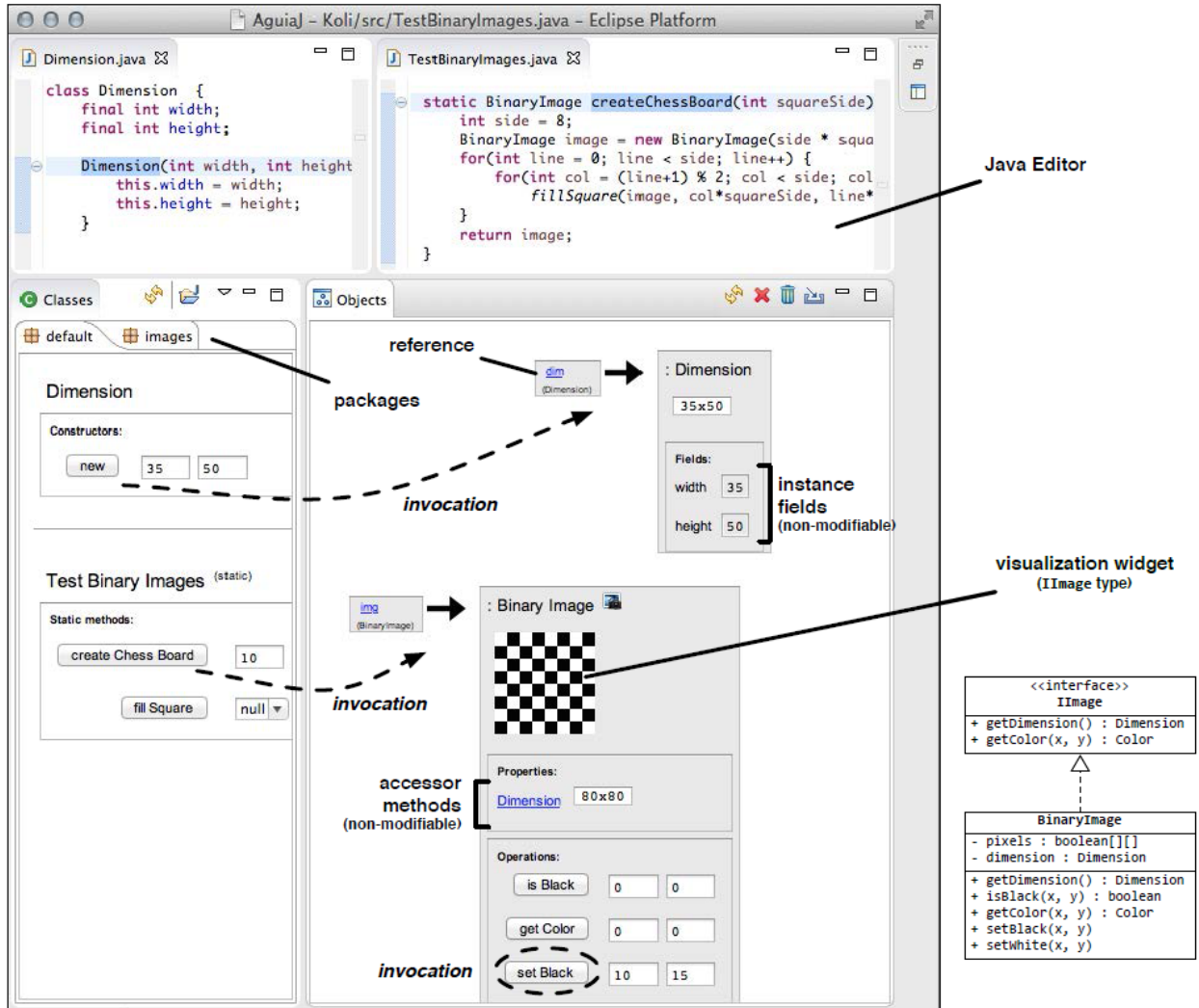


Figure 31. Screenshot of AguiAJ (image source from [19])

However, restrictions in AguiAJ became more noticeable while testing the yo-yo project code. One of the most notable problems is that inherited methods cannot be used as input to a particular object. In this context, the methods inherited from the `Employee` class by `StudentEmployee` and `StaffEmployee` cannot be used. Additionally, method calls within methods will not be executed. Since classes and objects generated in AguiAJ are not dependent on a particular runtime environment, this is not an effective tool for detecting runtime errors. At best, if users are testing lines of code from the current runtime environment, this would require

stepping back and forth between the AguiasJ interface and the standard Java interface for Eclipse. To solve this problem, users would have to know the issue is present in the `hourlyRate()` method, execute this method on each of the three classes, and observe the value of rate after execution.

JIVE

JIVE system provides a novel approach to the runtime visualization and analysis of OOP [22]. To use JIVE in Eclipse, users must properly configure the debugger. To enable the debugger for a project, users need to create a launch configuration for the project, edit the configuration to specify that JIVE must be used for debugging, and open the JIVE window perspective for checking the visualization of the program execution. The project must be run at least once before configurations can be modified to include JIVE. This process seems unnecessarily complicated compared against the other tools.

The JIVE interface itself includes two UML diagrams (Figure 32), an object and sequence diagram, and a contour model. Each is given its own set of buttons for stepping through the code by lines. This is extraneous as users stay in the same location of the code on both diagrams along with the source code. Given the use of an object and sequence diagram in JIVE, as users step through the code using the diagrams, the execution of code can be observed based on instantiations and method calls. However, to trace the values of the instance variables of an object, users must click the 'Contour Model' tab to switch over another panel and select the name of the variables.

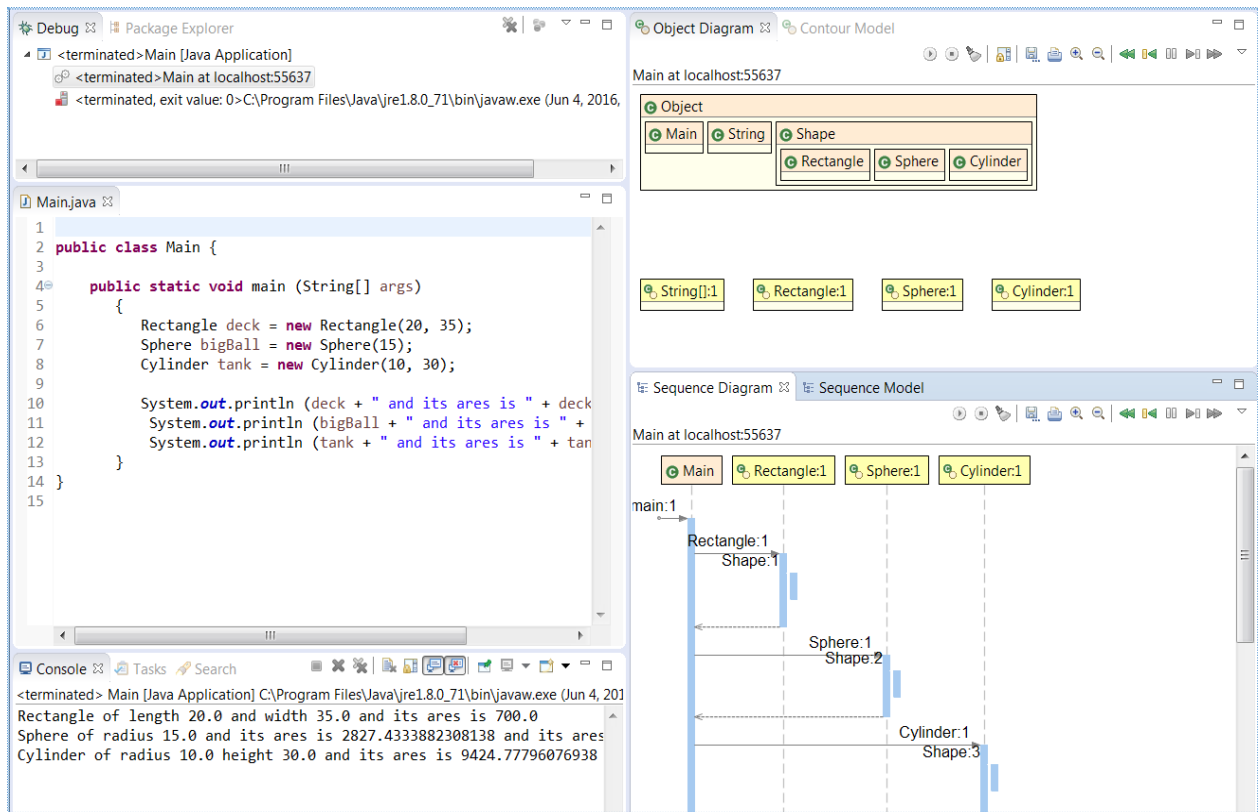


Figure 32. Screenshot of JIVE with object and sequence diagrams

Considering the design space and view of the interface, it should be noted that windows can become cluttered with larger projects. This was already becoming noticeable in the PolyShape project, which creates only six Polygon objects. Users may jump to specific points in a particular time line of the sequence diagram, allowing for more flexibility in parts of the code to focus on.

The biggest issue with JIVE while testing the yo-yo problem is that output is not generated alongside a line by line step through of the code. Similarly, the default setting for the class diagram does not provide enough visual evidence of the expected output of the program. In addition, observing the values of variables within a particular object requires setting the class diagram to objects with tables. Once the class diagram has been modified as such, users can

identify the issue when stepping through the methods for each class and noting that the value of rate is never set for objects of StaffEmployee or StudentEmployee.

JavelinaCode

JavelinaCode interface presents a static visualization of the code as a class diagram on the left hand side, the source code in the center, and a dynamic visualization on the right hand side. During line by line execution, the current line of code being executed and its encompassing class are highlighted in the source code and class diagram, respectively. Output is displayed at the bottom of the window.

The screenshot displays the JavelinaCode interface for a project named "PolyShape(2015-12-26)". It is divided into three main sections:

- Left Panel (Class Diagram):** Shows a class hierarchy. A "Main" class is at the top, with a method "+main(String[] args):void". Below it are three classes: "Rectangle", "Cylinder", and "Sphere". Arrows indicate that "Rectangle", "Cylinder", and "Sphere" all inherit from "Shape".
- Center Panel (Source Code):** Displays the source code for "Main.java". The code is as follows:

```
1 // PolyShape Project
2
3
4 public class Main
5 {
6     public static void main (String[] args)
7     {
8
9         Rectangle deck = new Rectangle(20, 35);
10        System.out.println (deck + " and its ares is " + deck.area());
11
12        Sphere bigBall = new Sphere(15);
13        System.out.println (bigBall + " and its ares is " + bigBall.area());
14
15
16        Cylinder tank = new Cylinder (10, 30);
17        System.out.println (tank + " and its ares is " + tank.area());
18    }
19
20 }
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

The line containing the creation of the "tank" object (lines 16-17) is highlighted in yellow.
- Right Panel (Dynamic Visualization):** Shows the state of the program at the current execution point. It includes a stack of frames for "main:71" with variables "deck", "bigBall", "tank", and "Return value" (void). Below this, three instance objects are shown with their attributes:
 - Rectangle instance:** length 20.0, width 35.0, shapeName "Rectangle", testVariable 100.
 - Sphere instance:** radius 15.0, shapeName "Sphere", testVariable 100.
 - Cylinder instance:** radius 10.0, height 30.0, shapeName "Cylinder", testVariable 100.Arrows point from the "deck", "bigBall", and "tank" variables in the stack to their respective instance data tables.

Figure 33. Screenshot of JavelinaCode with static and dynamic visualization

For the PolyShape project, during execution, the dynamic visualization is useful in observing the assignment of values to variables of each object. These objects are kept relatively organized in the window and can be easily referenced at the end of program execution. By stepping through each line of code in the project for the yo-yo problem, it can be observed that rate was not properly set for objects of StudentEmployee or StaffEmployee. This is easily identified by the dynamic visualization showing a value of zero being assigned to rate for instances of these classes as illustrated in Figure 33. Users are also able to analyze the class diagram to determine that a line of code is missing in the hourlyRate() method of StudentEmployee and StaffEmployee to assign some value to rate.

In summary, the results of the comparative analysis reveal that JavelinaCode provides a more effective means for understanding OO concepts than those considered. This is due to a simplified user interface and a better integrated set of both static and dynamic visualizations along with source code. Through the use of static and dynamic visualizations, student users can easily identify OO concepts such as polymorphic behavior, inheritance, and other OO paradigms. Static visualizations provide them a reference when identifying the inheritance between classes in class diagrams, and detecting problems such as the yo-yo problem. Dynamic visualizations aid in reinforcing how each line of code is associated with the construction of class instances and actions, including effectively showing polymorphic behavior at runtime. Combining both static and dynamic visualizations enables users to transition between identifying both the structure and behavior of object-oriented programs.

JavelinaCode excels in the simplicity of its interface. This includes reducing clutter, such as the issue found in JIVE, and keeping both the code and visualizations to a single window, such

as the issue found in BlueJ and jGRASP. Additionally, the set up process for JavelinaCode is reduced to navigating to a web page and registering an account as opposed to downloading and installing new software, allowing students to immediately begin programming.

6 Evaluation

6.1 Introduction

What do we mean by “usable” and what makes a software system usable? Rubin and Chrisnell [45] discussed the basic concepts of usability of a software product and the six aspects of usability. If a product is to be usable, it must be useful, efficient, effective, satisfying, learnable, and accessible.

Usefulness is to concern the degree to which attributes of a software product enables users to accomplish their specific goals and to assess the user’s willingness to use the product. Efficiency is the matter of time to be measured that the user’s goal can be achieved accurately to ensure its performance. Effectiveness is more likely to refer to the extent to which a software product does behave in a way that users expect and does provide easiness that users can do what they intend. A benchmark to test efficiency could be “95 percent of all users will be able to load the software in 10 minutes” while a benchmark to test the effectiveness could be “95 percent of all users will be able to load the software correctly on their first attempt” [45]. Learnability refers to the user’s ability to utilize a software system after a certain period of time and training and to re-utilize the system after a certain period of inactivity. It is to measure ease of learning that how fast users can achieve tasks without previous experience about the system. Satisfaction is regarding the user’s perceptions, feelings, and opinions of using a system. It is usually measured from written and/or

oral questioning about the system. Accessibility is the matter of how easily access to the system needed to perform tasks.

International Organization for Standardization (ISO) also defined that usability is extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use: effectiveness - accuracy and completeness with which users achieve specified goals, efficiency - resources expended in relation to the accuracy and completeness with which users achieve goals, and satisfaction - freedom from discomfort, and positive attitudes towards the use of the product [49].

To evaluate JavalinaCode, the aspects of effectiveness, efficiency, and satisfaction were adopted, since usability testing is a good research method in experimental study to evaluate the degree to which a software system meets specific criteria.

6.2 Validation

In order to evaluate the educational effectiveness, efficiency, and satisfaction of JavelinaCode, both quantitative and qualitative experiments were carefully designed and conducted. The quantitative and qualitative evaluation approach was adopted based on the suggestions by Rubin and Chisnell [45] and by the ITiCSE working group [46], which is one of six groups of approaches to assess educational systems.

This study aimed at evaluating the impact of using JavleinaCode with static and dynamic visualizations. The quantitative evaluation measures the results of data on performance from a group of users using JavelinaCode supporting both static and dynamic representation of source

code and a group of users using a standard IDE, NetBeans, with only the source code without using any visual aspects of the code. The resultant data is statistically analyzed. The qualitative study is designed to assess whether using JavelinaCode demonstrates the enhanced usability. An effective interface should be intuitive and user friendly rather than clustered and convoluted. The survey questionnaire was formed to measure whether the JavelinaCode environment does help students make their OOP learning easier and to alleviate the difficulties in programming. This is to measure if the usability of the JavelinaCode interface will contribute to similar unnecessary complexities and the more user friendly interface will improve student's intuitive and reasoning skills. The qualitative data collected from the questionnaire can be also used to guide an analysis that produces a quantitative output.

6.3 Quantitative Evaluation

Effectiveness could be measured in terms of accuracy and completeness with which users can achieve certain goals and another indicator of effectiveness could be the quality of a solution [48]. Efficiency is the relation between accuracy and completeness, and the resources expended to achieve them. Task completion time and learning time could be good indicators of measuring efficiency [48]. In this study, the correctness of solving programming problems is used as the quality of solution and is the primary indicator of the effectiveness of the JavelinaCode system. The completion time to solving a programming problem is used as a primary indicator of measuring efficiency.

Two controlled experiments were conducted to test hypotheses formulated in the following section. The study would be a controlled experiment that separates students into a controlled

group and an experimental group to do a comparative analysis of their performance statistical data on solving problems.

6.3.1 Hypotheses

The evaluation study may claim that the availability of both static and dynamic visualizations in JavelinaCode will reduce the amount of time taken to solve Java programming comprehension test questions and to increase the correctness of solutions. Accordingly, the following null and alternative hypotheses can be formulated:

- H_{10} : Having both static and dynamic visualizations available in JavelinaCode does not impact the correctness of solving problems.
- H_{20} : Having both static and dynamic visualizations available in JavelinaCode does not impact the time for solving programming problems.
- H_1 : Having both static and dynamic visualizations available in JavelinaCode increases the correctness of solving programming problems.
- H_2 : Having both static and dynamic visualizations available in JavelinaCode reduces the time for solving programming problems.

Two dependent variables, the time subjects spent to answer questions and the correctness of the answers to the questions, are used to statistically analyze the results of the collected data. The same hypotheses are used in both experiments.

6.3.2 Questionnaire

For the first session of both experiments, five Java classes including Main in Figure 34 are used and the following questionnaire is formed for tasks on program tracing and understanding. The Java classes simulate a PloyShape project that introduces some fundamental inheritance hierarchy with a Polygon, Rectangle, Sphere, and Cylinder class. The later three classes inherit variables and methods from their parent class and have an overridden method that calculates the area for each. This is considered as a relatively easy project to understand.

1. What is the parent (super) class of the class P?
 - a. C b. M c. R d. S
2. How many child (sub) classes does class S have?
 - a. 1 b. 2 c. 3 d. 4
3. Can an instance (object) of class R be an instance (object) of class S?
 - a. Yes b. No
4. Can an instance (object) of class S be an instance (object) of class C?
 - a. Yes b. No
5. Which method does demonstrate polymorphic behavior?
 - a. a b. sl c. gr d. sh
6. What is output by the statement at line # 9 marked // Problem 1 in Main.java?
 - a. R: l is 2.0 and w is 3.0 b. P: l is 2.0 and w is 3.0
 - c. C: l is 2.0 and w is 3.0 d. S: l is 2.0 and w is 3.0
7. What is the output by the statement at line # 11 marked // Problem 2 in Main. java?
 - a. Its a is 1.0 b. Its a is 6.0 c. Its a is 2.0 d. Its a is 3.0
8. What is the output by the statement at line # 13 marked // Problem 3 in Main. java?
 - a. R: r is 1.0 b. C: r is 1.0 c. P: r is 1.0 d. S: r is 1.0

9. What is the output by the statement at line # 15 marked // Problem 4 in Main. java?
 - a. Its a is 2.0
 - b. Its a is 3.0
 - c. Its a is 1.0
 - d. Its a is 6.0
10. What is the output by the statement at line # 17 marked // Problem 5 in Main. java?
 - a. P: r is 1.0
 - b. C: r is 1.0 and h is 3.0
 - c. R: r is 2.0 and h is 3.0
 - d. S: r is 1.0 and h is 3.0
11. What is the output by the statement at line #19 marked // Problem 6 in Main. java?
 - a. Its a is 1.0
 - b. Its a is 2.0
 - c. Its a is 6.0
 - d. Its a is 3.0

For the second session of both experiments, four Java classes including Main in Figure 35 were used and the following questionnaire is formed on program understanding and tracing.

1. What is the output by the statement at line # 7 marked // Problem 1 in Main.java?
 - a. John's payment is 10.0
 - b. John's payment is 100.0
 - c. John's payment is 0.0
 - d. John's payment is 150.0
2. What is the output by the statement at line # 11 marked // Problem 2 in Main.java?
 - a. Mary's payment is 10.0
 - b. Mary's payment is 15.0
 - c. Mary's payment is 150.0
 - d. Mary's payment is 0.0
3. What is the output by the statement at line # 15 marked // Problem 3 in Main.java?
 - a. Henry's payment is 20.0
 - b. Henry's payment is 200.0
 - c. Henry's payment is 0.0
 - d. Henry's payment is 10.0

Three Java classes (Employee.java, studentEmployee.java, staffEmployee.java) simulate the yo-yo effect that causes problems and results in a data flow anomaly from method overriding and polymorphism with an overridden method 'bounced' up and down among levels of inheritance hierarchy [44]. Considering the inheritance between the 'Employee' and

‘studentEmployee’ classes that have an overridden method ‘hourlyRate()’, due to the fact that the ‘hourlyRate’ method in the ‘studentEmployee’ sub-class, it does not have a rate assigned for an instance of that class, two student’s hourly rate (mary and henry) would be a ‘0.0’ resulting in ‘\$0.0’ payments despite having 15 (mary) and 20 (henry) hours of work.

```

1 - public class Main {
2
3 -     public static void main(String[] args){
4
5         R d = new R(2, 3);
6         P b = new P(1);
7         C t = new C(1, 3);
8
9         System.out.println(d.o());           // Problem 1
10
11        System.out.println("Its a is " + d.a()); // Problem 2
12
13        System.out.println( b.o());           // Problem 3
14
15        System.out.println("Its a is " + b.a()); // Problem 4
16
17        System.out.println( t.o());           // Problem 5
18
19        System.out.println("Its a is " + t.a()); // Problem 6
20
21    }
22
23 }

```

```

1 - abstract class S {
2
3     private String n;
4
5 -    public S (String na) {
6         n = na;
7     }
8
9     public abstract double a();
10
11    public String o() {
12        return n;
13    }
14
15 }

```

```

1 - public class R extends S {
2
3     private double l, w;
4
5 -    public R (double le, double wi) {
6         super("R");
7         l = le;
8         w = wi;
9     }
10
11    public double a() {
12        return (l * w);
13    }
14
15    public void sl(double le) {
16        l = le;
17    }
18
19    public void sw(double wi) {
20        w = wi;
21    }
22
23    public double gl() {
24        return ( l );
25    }
26
27    public double gw() {
28        return ( w );
29    }
30
31    public String o() {
32        return ( super.o() + ": l is " + l + " and w is " + w );
33    }
34
35 }

```

```

1 - public class C extends S {
2
3     private double r, h;
4
5 -    public C(double ra, double he) {
6         super("C");
7         r = ra;
8         h = he;
9     }
10
11    public double a() {
12        return ( r * r * h );
13    }
14
15    public void sh(double he) {
16        h = he;
17    }
18
19    public double gh() {
20        return ( h );
21    }
22
23    public String o() {
24        return ( super.o()+ ": r is " + r + " and h is " + h );
25    }
26
27 }

```

```

1 - public class P extends S {
2
3     private double r;
4
5 -    public P (double ra) {
6         super("P");
7         r = ra;
8     }
9
10    public double a() {
11        return ( r * r );
12    }
13
14    public void sr(double ra) {
15        r = ra;
16    }
17
18    public double gr() {
19        return ( r );
20    }
21
22    public String o() {
23        return ( super.o() + ": r is " + r );
24    }
25
26 }

```

Figure 34. Java classes used for Session 1 in both Experiments 1 & 2

```

1 public class Main {
2
3     public static void main(String[] args) {
4
5         Employee john = new Employee();
6         john.weeklyPay(10);
7         System.out.println(john.printCheck());    // Problem 1
8
9         studentEmployee mary = new studentEmployee();
10        mary.weeklyPay(15);
11        System.out.println(mary.printCheck());    // Problem 2
12
13        staffEmployee henry = new staffEmployee();
14        henry.weeklyPay(20);
15        System.out.println(henry.printCheck());    // Problem 3
16
17    }
18
19 }

```

```

1 public class studentEmployee extends Employee {
2
3     public studentEmployee() {
4         super();
5     }
6
7     public void hourlyRate() {
8         printCheck();
9     }
10
11    public double printCheck() {
12        return ( super.printCheck() );
13    }
14
15 }
16
17 }

```

Figure 35. Java classes used for Session 2 in both Experiments 1 & 2

```

1 public class Employee {
2
3     private int hour;
4     private double rate;
5
6     public Employee() {
7         hour = 0;
8         rate = 0;
9     }
10
11    public void weeklyPay(int h) {
12        workHours(h);
13    }
14
15    public void workHours(int h) {
16        hour = h;
17        hourlyRate();
18    }
19
20    public void hourlyRate() {
21        rate = 10;
22        printCheck();
23    }
24
25    public double printCheck() {
26        return ( hour * rate );
27    }
28
29 }

```

```

1 public class staffEmployee extends Employee {
2
3     public staffEmployee() {
4         super();
5     }
6
7     public void hourlyRate() {
8         printCheck();
9     }
10
11    public double printCheck() {
12        return ( super.printCheck() );
13    }
14
15 }

```

This is considered as a relatively difficult project to understand and it serves as an example of the issues students encounter when transitioning to practical programming applications. To record a response time and answer accurately for each question, the questionnaire was presented in a series of web pages. Each page contains a single question. Whenever a question is answered by selecting one of the choices (radio buttons) and clicking on ‘Next’ button, the response time and response to that question were saved in a database. A response time is calculated as the time elapsed from when the current question is loaded until the student submits a response by clicking the ‘Next’ button for a next question.

6.3.3 Data Analysis

The experiments were designed in such a way that each observation (question) in one population (a controlled group) is matched with an observation in other population (an experimental group). The matching is conducted by using the same set of questions for each group. Thus, it is logical to compare the difference for both groups for each question. To statistically verify whether both static and dynamic visualizations provided in JavelinaCode have impact on the response time and correctness to answer questions, the null hypotheses were tested using Student's t-test, which was used to analyze the data since the experiments were designed to have matched samples in two groups. In a t-test, differences among the means of both response time and correctness between two populations were studied. The null hypotheses are that the group means for all responses and correctness are the same ($\mu_d = 0$), which is labeled with $\mu_d = \mu_1 - \mu_2$. To test the null hypothesis about μ_d , the following test statics was used:

$$t = \frac{\bar{x}_d - \mu_d}{\left(\frac{s_d}{\sqrt{n}}\right)}$$

where \bar{x}_d is a sample mean, s_d is a sample standard deviation, and n is the number of differences.

To validate that the t-test can be used, the Kolmogorov-Smirnov (K-S) test was applied to verify normal distribution in the sample. As shown in the tables 4, 5, 6, and 7, p-values of the K-S test are greater than a value 0.05, which is what we are looking for and is significant that the

sample is normal. Due to a small sample size (observation), the two-sample K-S test was applied for the yo-yo problem project.

6.3.4 Experiment 1

The hypotheses for this experiment were that students would respond faster and with higher accuracy for programming tracing and understanding using both static and dynamic visualizations in JavelinaCode.

6.3.4.1 Participants

Sixteen lower-division computer science major students enrolled in Data Structures and Algorithms at Texas A&M University-Kingsville participated in this experiment (see Table 3). The student participants are considered as novices without much experience in JAVA except for taking Object-Oriented Software Engineering in Java as a prerequisite of the current course. To treat the participants in accordance with the “Ethical Principles of Psychologists and Code of Conductor” [47], they were given a small amount of extra credits for their participation toward their final grade in their course.

Table 3. No. of subjects participated in both experiments

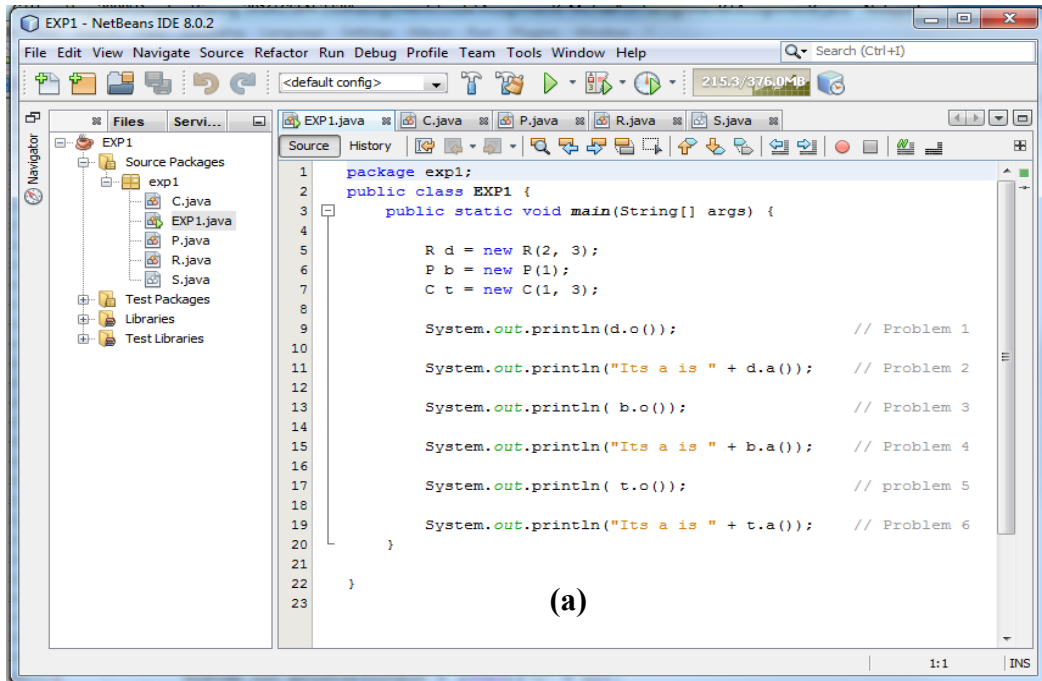
	No. of participants		
	Experiment 1	Experiment 2	Total
	Undergraduates	Graduates	
Controlled group	6	33	39
Experimental group	10	42	52
Total	16	75	91

6.3.4.2 Method and Procedure

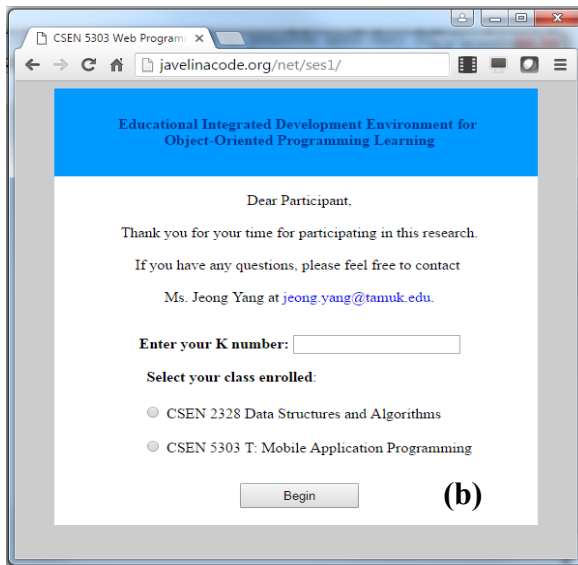
Student participants were divided into two groups. One controlled group was given Java projects (PolyShape project with Java classes in Figure 34 for Session 1 and yo-yo problem project with Java classes in Figure 35 for Session 2) in plain text with NetBeans IDE 8.0.2 while the other experimental group was given the same Java projects with two aspects of static UML class diagrams and dynamic run time visualization of program execution in JavelinaCode. To equally balance two groups for reliable results, the selection of the participants was based on their cumulated grade for the course. The experiment was held in one of the computer classrooms located in the College of Engineering at Texas A&M University-Kingsville. There were two one-day sessions (one session each day) for each group. Each session lasted approximately one hour.

For the controlled group:

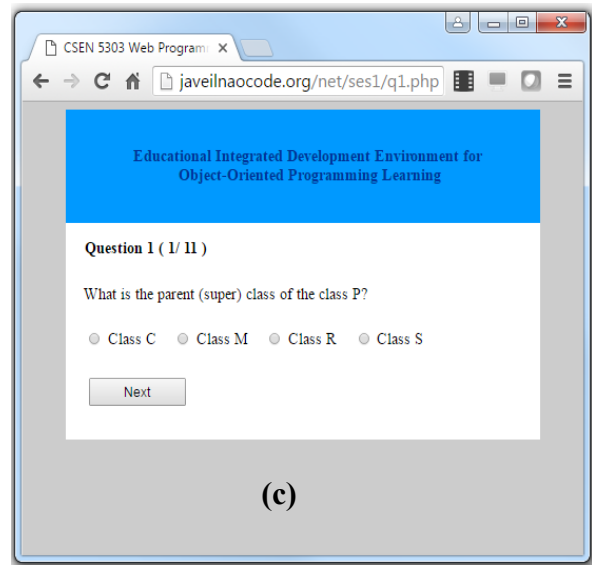
- Session 1 – Day 1
 - The student participants were given a short instruction about the experiment.
 - They were given a consent form to sign that they agreed the participation was voluntary and unpaid.
 - They were instructed to log into the course Blackboard, download a prepared zip file of the PolyShape project, unzip it, and open the project in the NetBeans IDE.
 - They were also instructed to close the ‘Output’ window and not to run the project because the questions are related to the output results by the statements executed.
- Two graduate assistants were assigned to an instructor to observe the experiment.



(a)



(b)



(c)

Figure 36. Screenshots of (a) PolyShape project in NetBeans IDE, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question

- They were instructed to take their time as much as possible until they fully understood the code and be ready to answer questions listed in section 6.3.2 related to the PloyShape project.

- To begin answering the questions, the participants were instructed to load a specific URL and to do their best to answer each question correctly.

Figure 36 gives the screenshots of (a) the PloyShape project opened in NetBeans IDE, (b) the first web page loaded to give an ID and selection of the classes enrolled and begin answering questions, and (c) a web page loaded for the first question.

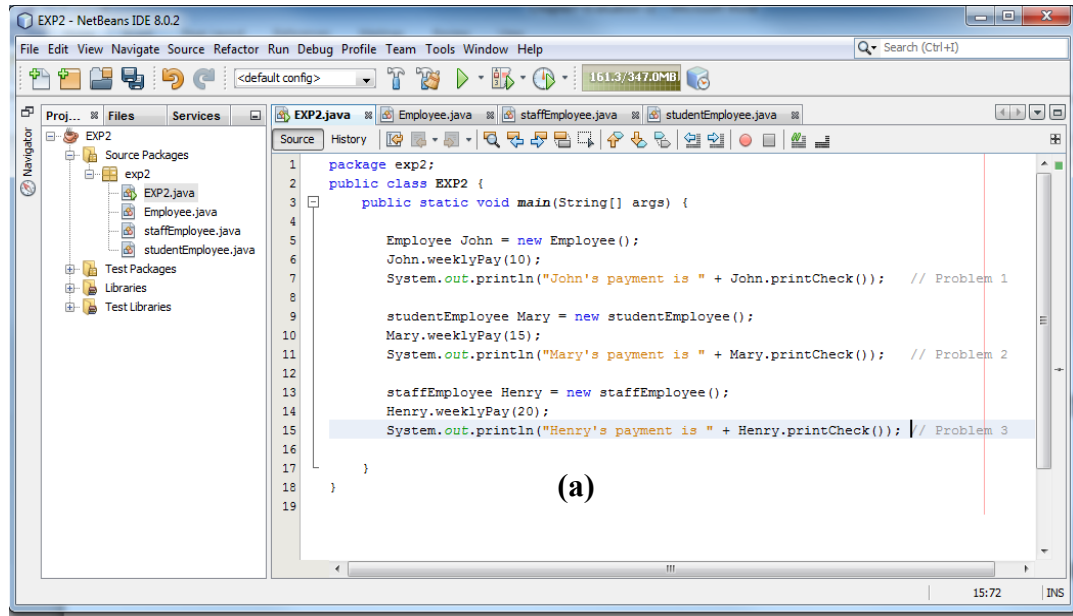
- Session 2 – Day 2

- The student participants were instructed to log into the course Blackboard, download a prepared zip file of the yo-yo problem project, unzip it, and open the project in the NetBeans IDE.
- They were also instructed to close the ‘Output’ window and not to run the project because many of the questions asked the output results by the statements executed. Two graduate assistants were assigned to an instructor to observe the experiment.
- They were instructed to take their time as much as possible until they fully understood the code and be ready to answer questions listed section 7.3.2 related to the yo-yo problem project.
- To begin answering the questions, participants were instructed to load a specific URL and to do their best to answer each question correctly.

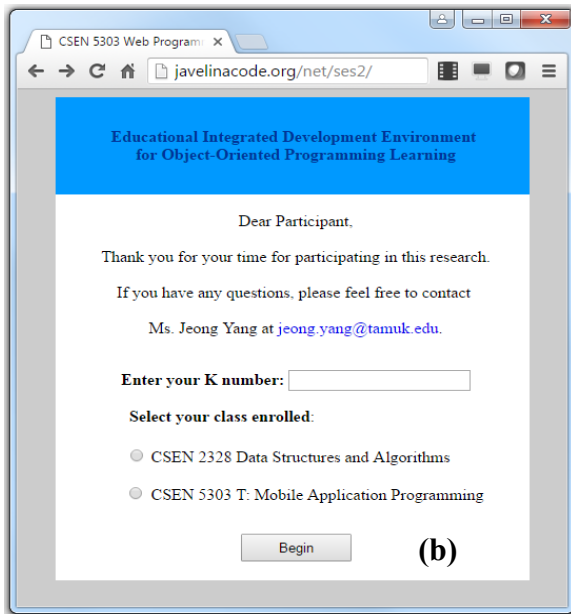
Figure 37 gives the screenshots of (a) the yo-yo problem project opened in NetBeans IDE, (b) the first web page loaded to give an ID and selection of classes enrolled and begin answering questions, and (c) a web page loaded for the first question.

For the experimental group:

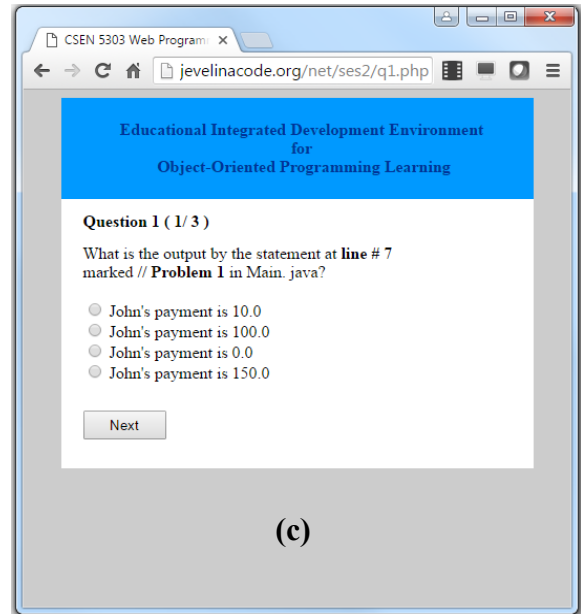
- Session 1 – Day 1



(a)



(b)



(c)

Figure 37. Screenshots of (a) yo-yo problem project in NetBeans IDE, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question

- The student participants were given a short instruction about the experiment.
- They were given a consent form to sign that they agreed the participation was voluntary and unpaid.

- They were instructed to open the Chrome browser to access JavelinaCode (<http://javelinacode.org>) and to log into the system using their assigned id and password.
- Pre session activity was provided for the participants to get familiarize with JavelinaCode system in terms of creating a project, adding Java files into the project, editing source code, closing or deleting a file from the project, renaming a file, compiling and running the project, checking the compact and detailed UML class diagrams, and visualizing the run time state of program execution.

The figure consists of three screenshots of a Java IDE showing code for different classes:

- Shirt.java:**

```

1 public class Shirt{
2
3     private int shade;
4
5     public Shirt (int s)
6     {
7         shade = s0;
8     }
9
10    public void wash () {
11        shade--;
12    }
13
14    public int getShade() {
15        return shade;
16    }
17
18 }
19

```
- Main.java:**

```

1 public class Main{
2
3     public static void main(String[] args){
4
5         Tshirt ts = new Tshirts(4,5);
6         System.out.println(ts.getShade());
7
8         Shirt s2 = new TShirts(10, 4);
9         s2.wash();
10        s2.wash();
11        System.out.println(s2.getShade());
12
13    }
14
15 }
16

```
- Tshirt.java:**

```

1 public class Tshirt extends Shirt {
2
3     private int mult;
4
5     public Tshirt(int s, int m) {
6         super(s);
7         mult = m;
8     }
9
10    public void wash() {
11        for (int i = 0; i < mult; i++ )
12            super.wash();
13    }
14
15 }
16

```

Figure 38. Java classes used to familiarize with JavelinaCode system

- The instructor led the class to show how to use menus and how to do related activities listed above and two graduate assistants were assigned to the instructor to observe

and help the participants. The Java classes involved in this activity are represented in Figure 38.

- After the pre session is over, the student participants were instructed to open the prepared Java project (project named EXP1 for the same PolyShape project) and close the output console window.
- They were instructed to take their time as much as possible until they fully understood the code and be ready to answer questions listed section 7.3.2. This time the visualization of UML class diagrams and program execution to the code were provided from JavelinaCode.
- To begin answering the questions, the participants were instructed to load a specific URL and to do their best to answer each question correctly.

Figure 39 gives the screenshots of (a) the PloyShape project opened in JavelinaCode with two aspects of static and dynamic visualizations, (b) the first web paged loaded to give an ID and selection of classes enrolled and begin answering questions, and (c) a web page loaded for the first question.

- Session 2 – Day 2
 - Student participants were instructed to open the Chrome browser to access JavelinaCode and to log into the system using their assigned id and password.
 - They were instructed to open the prepared Java project (project named EXP2 for the yo-yo problem project) and close the output console window.
 - They were instructed to take their time as much as possible until they fully understood the code and be ready to answer questions listed section 6.3.2. The

visualization of UML class diagrams and program execution to the code were provided from JavelinaCode.

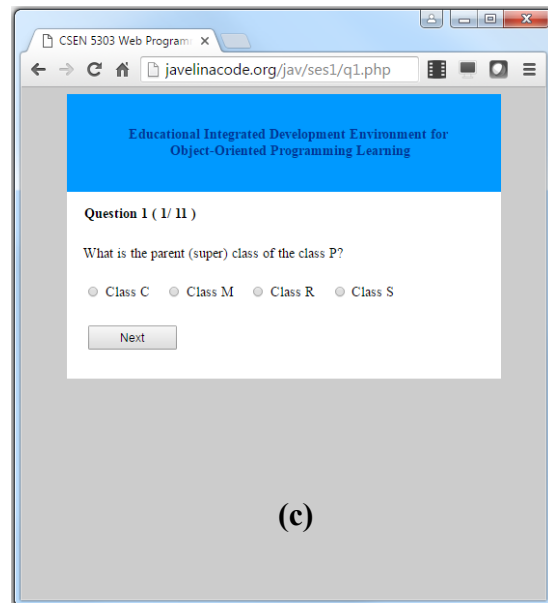
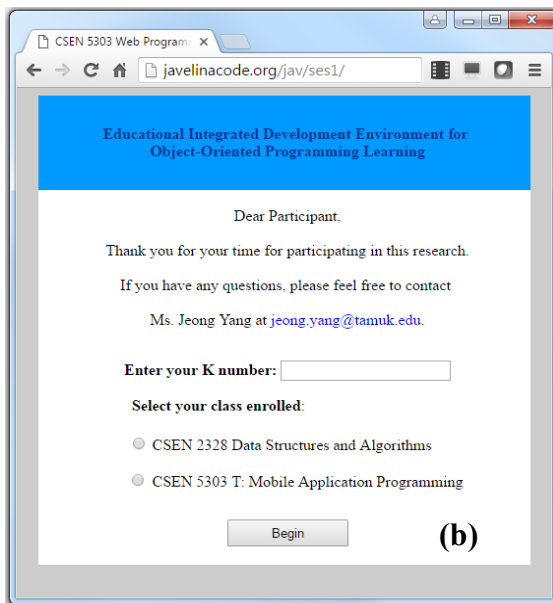
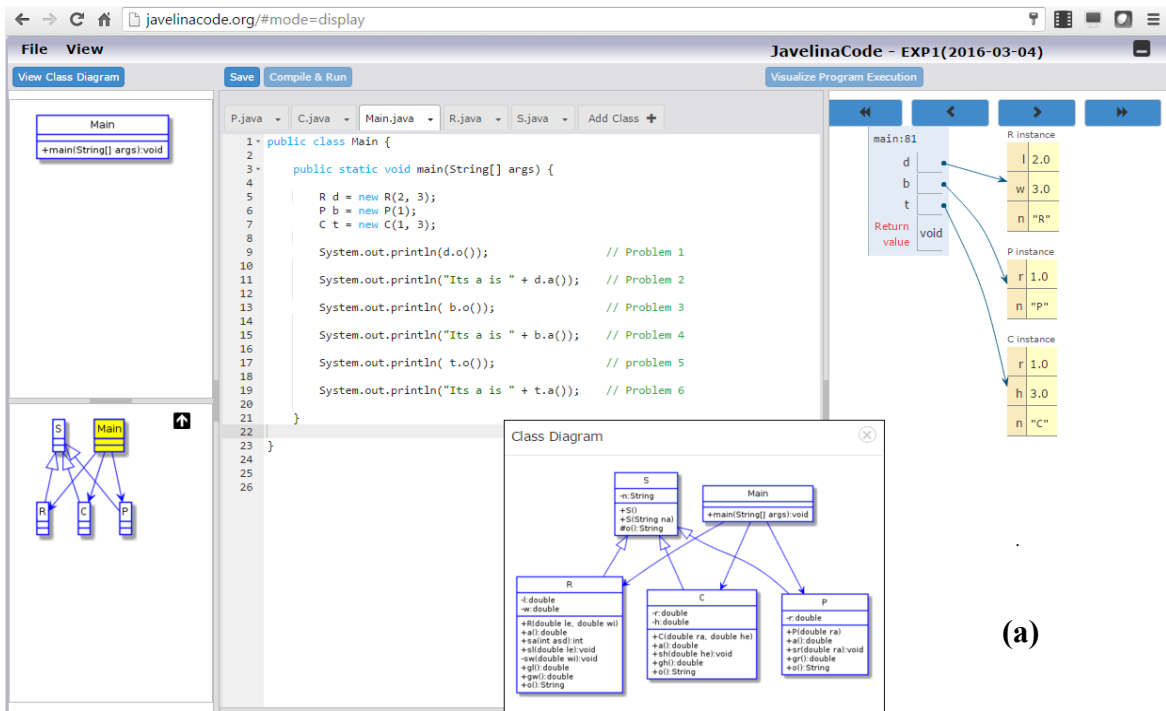


Figure 39. Screenshots of (a) PolyShape project in JavelinaCode, (b) Web Page loaded for ID and Class selection, and (c) Web Page for the first question

(a)

(b)

(c)

(d)

Figure 40. Screenshots of (a) yo-yo problem project in JavelinaCode, (b) Web Page for ID and class selection, (c) Web Page for the first question, and (d) sample page for usability questions

- To begin answering the questions, the participants were instructed to load a specific URL and to do their best to answer each question correctly
- Figure 40 gives the screenshots of (a) the yo-yo problem project opened in JavelinaCode with two aspects of static and dynamic visualizations, (b) the first web page loaded to give an ID and selection of classes enrolled and begin answering questions, (c) a web page loaded for the first question, and (d) a sample web page for usability questions.
- After the source code related questions were answered, a series of visualization and usability related questions described in Tables 8, 9, and 10 in section 6.4.1 were asked to gather data on how satisfied the participants were with JavelinaCode.

6.3.4.3 Results

Response Time

The average response time is the total average time taken to respond each question. Figure 41 shows the comparison of the average response times taken to answer questions related to the PloyShape project by both groups (group 1: the controlled group with source code in plain text using NetBeans IDE and group 2: the experimental group with the same code along with visualizations using JavelinaCode). While the average response times in group 2 for six questions (q1, q2, q4, q6, q8, q10) are less than the ones in group 1, as shown in Table 4, the statistical analysis reveals that there is no significant difference between two groups in terms of the response time.

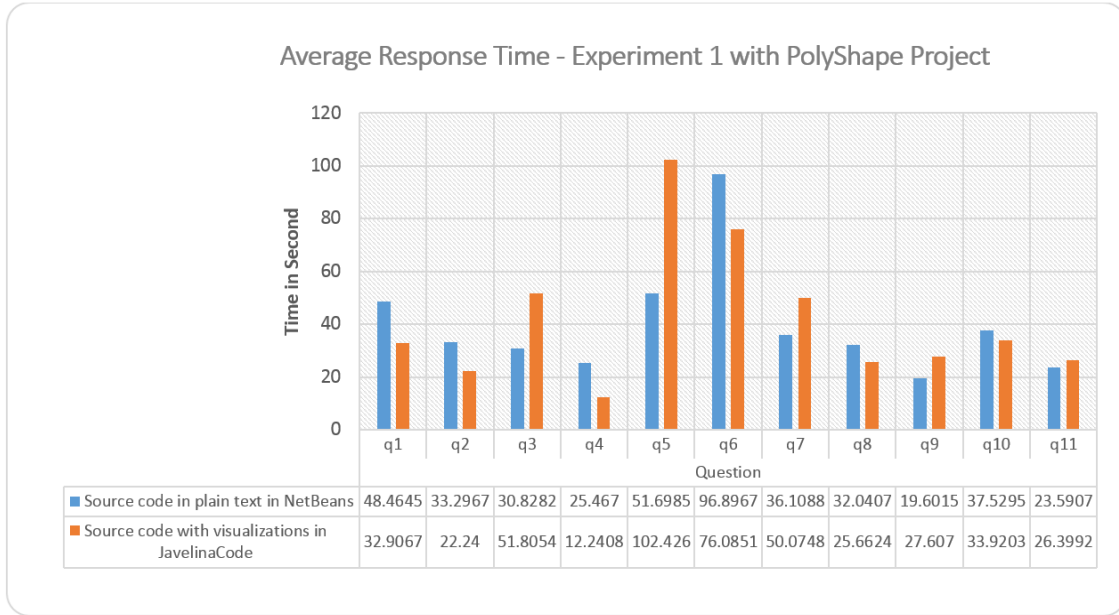


Figure 41. Comparison of average response time with PolyShape project

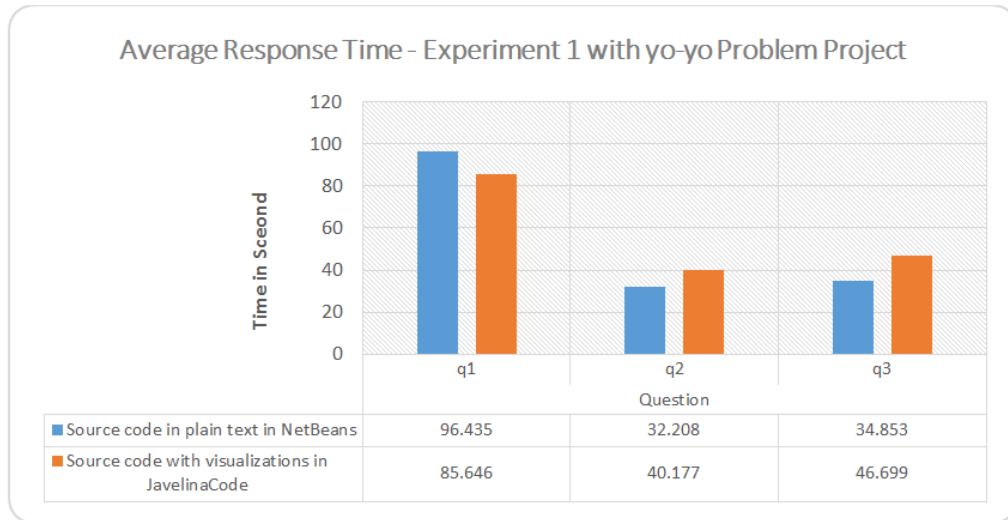


Figure 42. Comparison of average response time with yo-yo problem project

Table 4. Statistical evaluation of response time in Experiment 1

		Mean (Time in Second)	Standard Deviation	Observation	K-S	t	p
Session 1 with PolyShape project	G1	39.593	21.340	11	0.14	-0.378	0.357
	G2	41.942	26.676		0.27		
Session 2 with yo-yo problem project	G1	54.499	36.342	3	0.58	-0.430	0.354
	G2	57.507	24.586				

G1 denotes 'Controlled Group' and G2 denotes 'Experimental Group'.

Figure 42 shows the comparison of the average response time taken to answer each question related to the yo-yo problem project by both groups. As shown in Table 4, the statistical analysis again reveals that there is no significant difference between two groups in terms of the response time.

Correctness

The correctness is the total percentage of correct responses to each question. Figure 43 shows the comparison of the correctness to answer questions related to the PloyShape project by both groups. While the correctness for 9 questions (81.81%: q1, q2, q3, q4, q6, q7, q9, q10, q11) in group 2 are equal to or higher than the ones in group 1, as shown in Table 5, the statistical analysis reveals that there is a significant difference between two groups in terms of the correctness. The result of t-test rejects the first null hypothesis and accepts the alternative hypothesis, which means that the correctness is statistically significantly increased by the availability of both visualizations in JavelinaCode as the p-value is with 0.025.

Figure 44 shows the comparison of the correctness to answer each question related to the yo-yo problem project by both groups. Although the correctness for all three questions (100%: q1, q2, q3) in group 2 are equal to or higher than the ones in group 1, as shown in Table 5, the statistical analysis reveals that there is no significant difference between two groups in terms of the correctness. This is due to higher variances with a relatively smaller sample size, (observation 3).

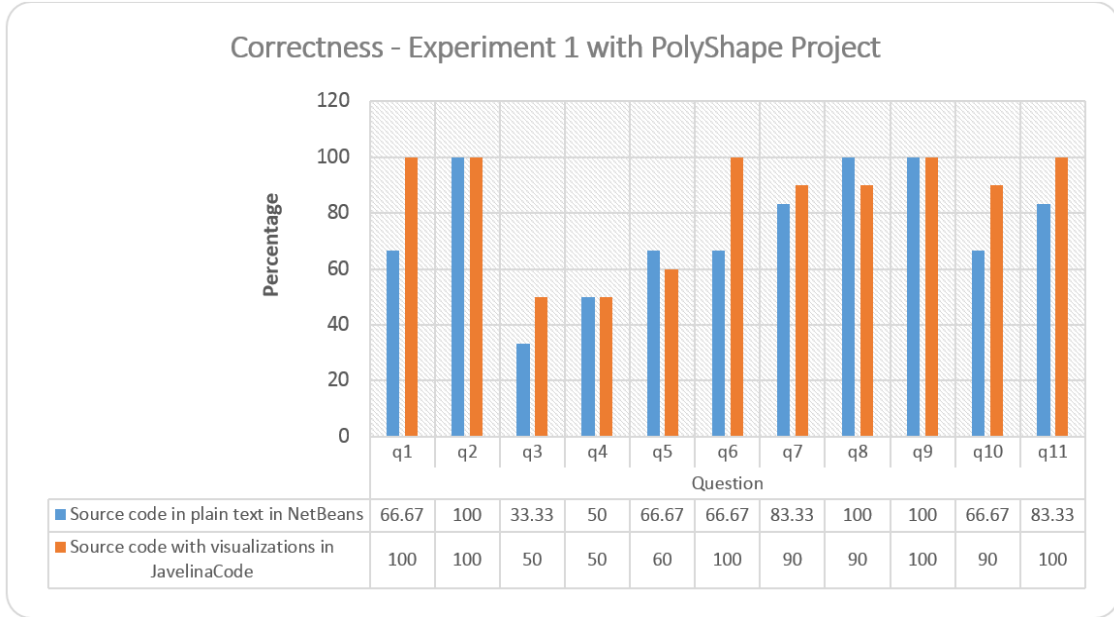


Figure 43. Comparison of correctness with PolyShape project

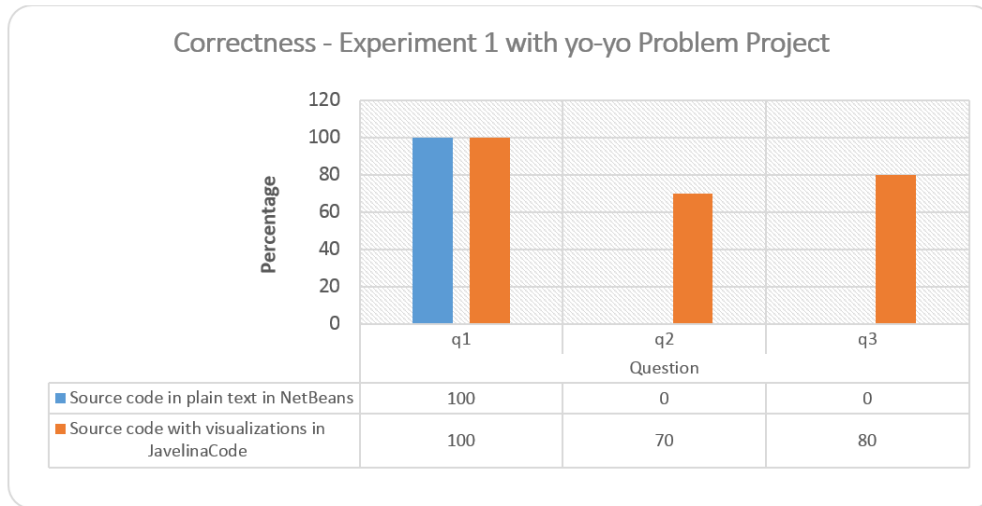


Figure 44. Comparison of correctness with yo-yo problem project

Table 5. Statistical evaluation of correctness in Experiment 1

		Mean (Percentage)	Standard Deviation	Observation	K-S	t	p
Session 1 with PolyShape project	G1	74.243	21.556	11	0.62	-2.232	0.025
	G2	84.545	20.671		0.15		
Session 2 with yo-yo problem project	G1	33.333	57.735	3	0.518	-1.987	0.093
	G2	83.333	15.275				

G1 denotes 'Controlled Group' and G2 denotes 'Experimental Group'.

6.3.5 Experiment 2

The same hypotheses were applied for this experiment that students would respond faster and with higher accuracy for program tracing and understanding using both static and dynamic visualizations in JavelinaCode.

6.3.5.1 Participants

Seventy five graduate level computer science major students enrolled in Mobile Application Programming (Android platform) at Texas A&M University-Kingsville participated in this experiment. The student participants are considered as relative experts in the experimental task.

6.3.5.2 Method and Procedure

Student participants were divided into two groups. One controlled group was given Java projects in plain text with NetBeans IDE while the other experimental group was given the same Java projects with two aspects of static UML class diagrams and dynamic run time visualization of program execution in JavelinaCode. To equally balance two groups for reliable results, the selection of the participants was based on their cumulated grade for the course. The experimental method and procedure for both controlled and experimental groups were the same described in section 6.3.4.2 for Experiment 1. Both the amount of time taken to answer each question and the correctness of the solution were recorded. To record responses and response times accurately, the questions were presented in a series of web pages.

6.3.5.3 Results

Response Time

Figure 45 shows the comparison of the average response times taken to answer questions related to the PloyShape project by both groups. While the average response times in group 2 for five questions (q1, q2, q8, q10, and q11) are less than the ones in group 1, as shown in Table 6, the statistical analysis reveals that there is no significant difference between two groups in terms of the response time.

Figure 46 shows the comparison of the average response time taken to answer questions related to the yo-yo problem project by both groups. As shown in Table 6, the statistical analysis reveals that there is a significant difference between two groups in terms of the response time. The result of t-test rejects the second null hypothesis, which means that the response time is statistically significantly increased by the availability of both visualizations in JavelinaCode as the p-value is with 0.012. This is a new finding of an opposite result of what is expected.

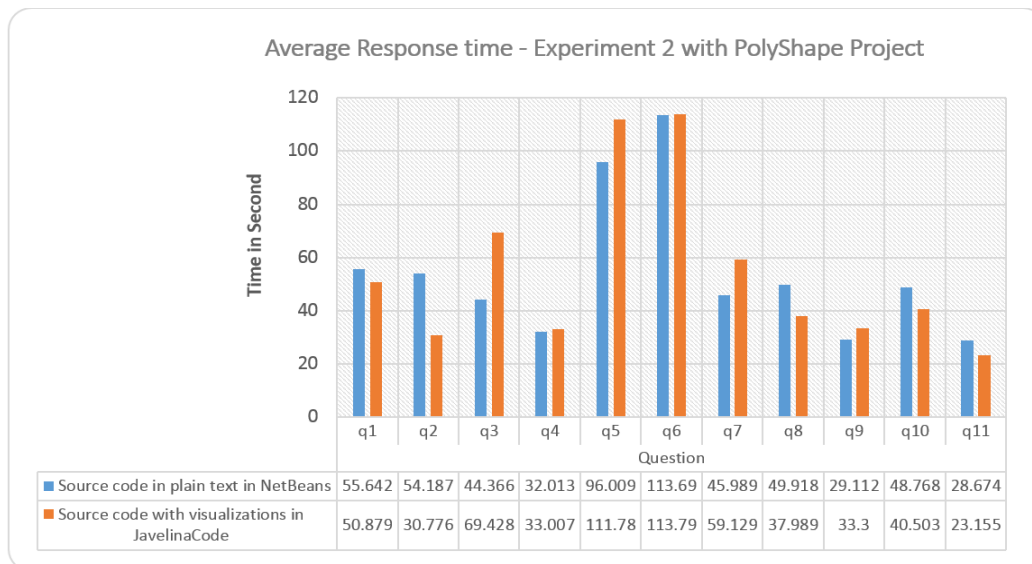


Figure 45. Comparison of average response time with PolyShape project

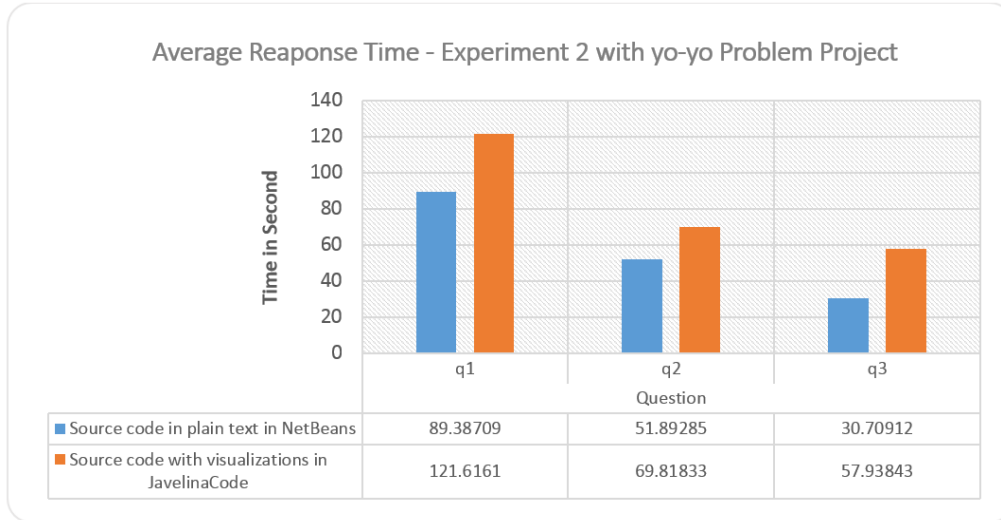


Figure 46. Comparison of average response time with yo-yo problem project

Table 6. Statistical evaluation of response time in Experiment 2

		Mean (Time in Second)	Standard Deviation	Observation	K-S	t	p
Session 1 with PolyShape project	G1	54.398	26.971	11	0.08	-0.118	0.454
	G2	54.886	31.587		0.46		
Session 2 with yo-yo problem project	G1	57.330	29.714	3	0.518	-6.155	0.012
	G2	83.124	33.860				

G1 denotes 'Controlled Group' and G2 denotes 'Experimental Group'.

Correctness

Figure 47 and 48 show the comparison of the correctness to answer questions related to the PloyShape project and the yo-yo problem project respectively by both groups. While the correctness for 8 questions (72.73%: q2, q3, q4, q5, q6, q7, q9, and q11) in group 2 are higher than the ones in group 1 for the session 1, the correctness for all three questions (100%: q1, q2, q3) in group 2 are also higher than the ones in group 1 for the session 2. As shown in Table 7, the statistical analysis reveals that there is a significant difference between the two groups in terms of the correctness for both sessions with two different projects. The results of

t-test reject the first null hypothesis and accept the alternative hypothesis, which means that the correctness is statistically significantly increased by the availability of both visualizations in JavelinaCode as the p-value is with 0.027 for the PloyShpe project and 0.006 for the yo-yo problem project.

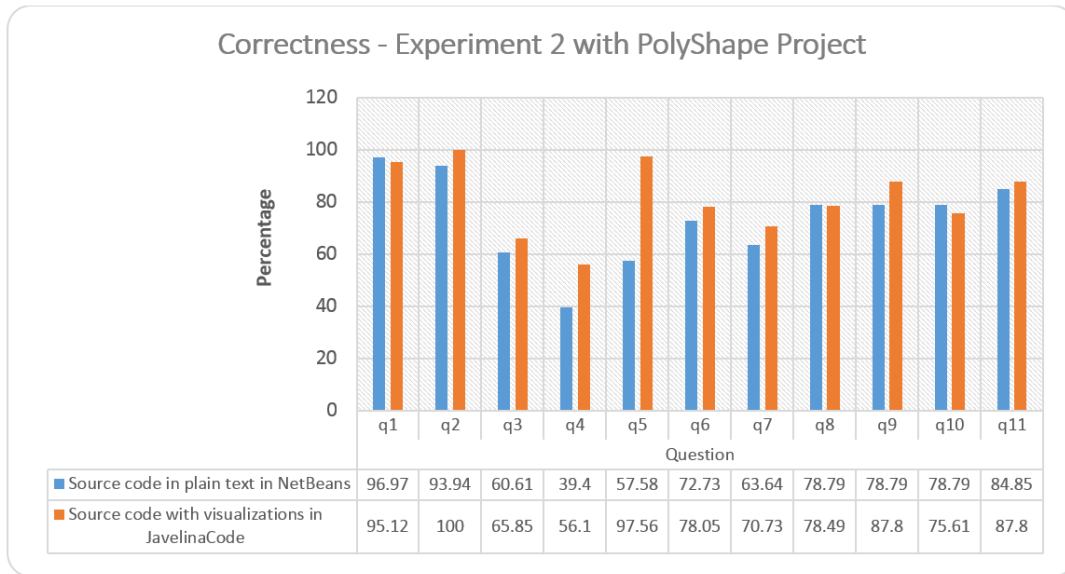


Figure 47. Comparison of correctness with PolyShape project

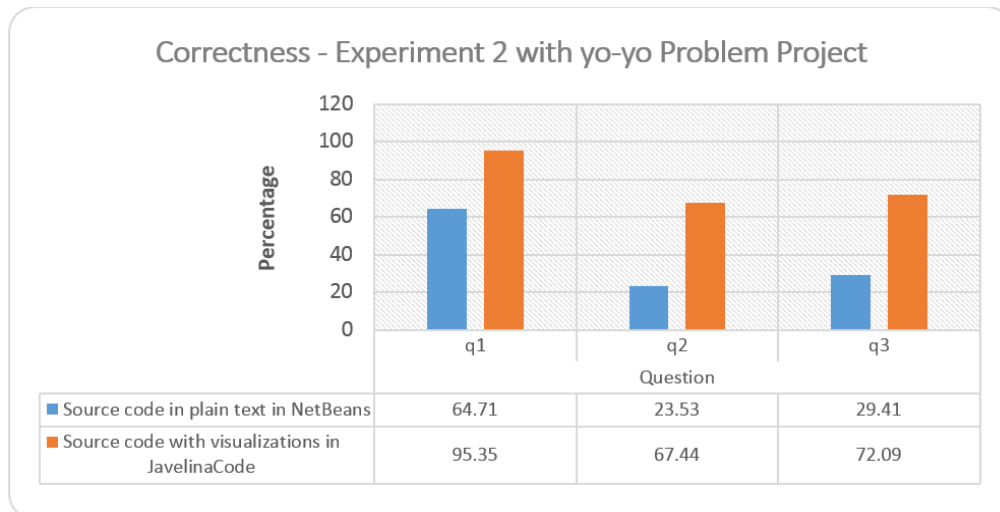


Figure 48. Comparison of correctness with yo-yo problem project

Table 7. Statistical evaluation of correctness in Experiment 2

		Mean (Percentage)	Standard Deviation	Observation	K-S	t	p
Session 1 with PolyShape project	G1	73.281	16.915	11	0.93	-2.190	0.027
	G2	81.192	13.877		0.99		
Session 2 with yo- yo problem project	G1	39.127	22.273	3	0.1	-9.231	0.006
	G2	78.293	14.953				

G1 denotes ‘Controlled Group’ and G2 denotes ‘Experimental Group’.

In summary, it was observed that students in the experimental group using two aspects of visualizations in JavelinaCode consistently performed better to correctly respond to questions on program understanding for both relatively easy and hard questions than the controlled group. Therefore, the statistical analysis of the experimental data supports the conclusion that having both static and dynamic visualizations available in JavelinaCode does positively impact the correctness of solving problems. The analysis also helps draw a conclusion that having both static and dynamic visualizations available in JavelinaCode does positively impact on increasing the correctness of solving problems, in particular, for relatively difficult questions (yo-yo problem project).

6.4 Qualitative Evaluation

The qualitative study is designed to assess whether using JavelinaCode with the synchronized static and dynamic visualization could contribute to its goals of meeting user’s needs and providing satisfaction. Satisfaction is regarded as the users’ subjective reaction, comfort and positive attitude with using a system. In this study, the questionnaires with the System Usability Scale (SUS) suggested by ISO standard 9241 [50] are used to gather data on how satisfied users were with JavelinaCode. The degree of disagreement or agreement on a 5 point rating scale is ranged from 1 ‘Strongly Disagree’ to 5 ‘Strongly Agree’.

6.4.1 Objectives and Questionnaire

For student users, the main objectives of the evaluation can be established in the following:

- a. Do the UML class diagrams in JavelinaCode support student’s understanding of object-oriented concepts?
- b. Does the run-time visualization in JavelinaCode support student’s understanding of object-oriented programming?
- c. Does the JavelinaCode system make learning of object-oriented program easier?
- d. Is the JavelinaCode system easy to use?
- e. Are students satisfied and comfortable using the JavelinaCode system?

Background related questions for student’s class status, major, and rating in Java were formed as shown in Table 8. As described in Tables 9 and 10, the questionnaires related to visualizations and usability of the JavelinaCode system were formed to measure attainment of the objectives addressed above.

Table 8. Background questions

Background questions	
1. What is your class status?	a. Freshman b. Sophomore c. Junior d. Senior e. Graduate
2. What is your major?	a. Computer Science b. Electrical Engineering c. Other
3. How do you rate yourself in Java?	a. Beginner b. Advanced beginner c. Competent d. Proficient e. Expert

Table 9. Visualization related questions

Associated objectives	UML class diagram related questions
-----------------------	-------------------------------------

a	<p>1. UML class diagrams helped me better understand overall structure of Java programs. a. 1 b. 2 c. 3 d. 4 e. 5</p> <p>2. UML class diagrams help me to better understand Object-Oriented design concepts. a. 1 b. 2 c. 3 d. 4 e. 5</p> <p>3. Please add any other comment on the “UML class diagrams”.</p>
	Run time visualization related questions
b	<p>4. I understand the “dynamic run time visualization of Java program execution”. a. 1 b. 2 c. 3 d. 4 e. 5</p>
b	<p>5. The “dynamic run time visualization of Java program execution” helps me correct and improve the quality of my program. a. 1 b. 2 c. 3 d. 4 e. 5</p>
	6. Please add any other comment on the “dynamic run time visualization of Java program execution”.
	Synchronized UML diagrams and run time visualization related questions
c	<p>7. Synchronized UML class diagram and run time visualization along with source code make it easier for me to comprehend Java program. a. 1 b. 2 c. 3 d. 4 e. 5</p>
c	<p>8. Synchronized UML class diagram and run time visualization along with source code alleviate the intimidation of Java programming. a. 1 b. 2 c. 3 d. 4 e. 5</p>
	<p>9. Which visualization aspect was more useful to comprehend the code? a. UML class diagrams b. Run time visualization c. Both</p>

Table 10. Usability related questions

Associated objectives	Usability related questions
d	1. JavelinaCode makes it easier for me to start writing Java programs. a. 1 b. 2 c. 3 d. 4 e. 5
e	2. The interface of JavelinaCode is user friendly. a. 1 b. 2 c. 3 d. 4 e. 5
d	3. JavelinaCode is easy to use. a. 1 b. 2 c. 3 d. 4 e. 5
e	4. I enjoy the time I spent using JavelianCode. a. 1 b. 2 c. 3 d. 4 e. 5
e	5. Working with JavelinaCode is satisfying. a. 1 b. 2 c. 3 d. 4 e. 5

d	6. The way that JavelinaCode is presented is clear and understandable. a. 1 b. 2 c. 3 d. 4 e. 5
e	7. I was comfortable in programming with JavelinaCode. a. 1 b. 2 c. 3 d. 4 e. 5
e	8. I would like to use JavelinaCode for the rest of the subject a. 1 b. 2 c. 3 d. 4 e. 5
e	9. The speed of using JavelinaCod is fair enough. a. 1 b. 2 c. 3 d. 4 e. 5
	10. Please specify any difficulties or problems you have encountered while using JavelinaCode? 11. Please specify key benefits of using JavelinaCode. 12. What do you think needs most important improvement on JavelinaCode, and why?

6.4.2 Participants

41 students (6 undergraduates and 35 graduates) out of 52 students from the experimental groups of both Experiment 1 and Experiment 2 participated in this evaluation. They experienced the JavelinaCode system through two sessions of both experiments for about one and half hours before taking the questionnaire. As shown in Figure 53, among the 41 respondents, the majority, 29 (71%) of respondents indicated that they were either an advanced beginner or competent in rating themselves in Java. All respondents were Computer Science majors: including thirty five graduates, four seniors, and two sophomores.

6.4.3 Method

In both Experiments 1 and 2, after the completion of the questions related to the source code in session 2, a series of visualization and usability related questions described in Tables 8, 9, and 10 were asked for the participants to rate the degree of disagreement or agreement on a 5 point rating scale and gather data on how satisfied the participants were with JavelinaCode.

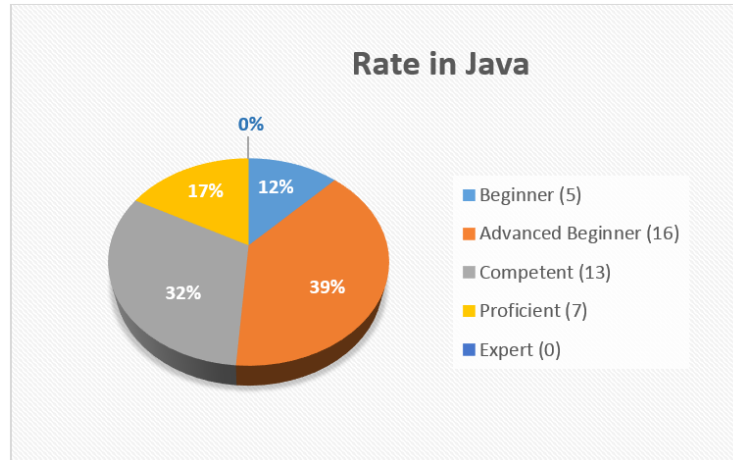


Figure 53. Rate students in Java

6.4.4 Results and Discussion for Aspect of Visualizations

Aspect of static visualization in UML class diagrams

Question 1: UML class diagrams help me better understand overall structure of Java programs.

For the UML class diagram related question #1, among the 41 respondents, 40 (98%) respondents indicated that they strongly agreed or agreed that the UML diagrams helped them better understand overall structure of Java programs as shown in Figure 54 (mean agreement rating = 4.44).

Question 2: UML class diagrams help me better understand Object-Oriented design concepts.

For the UML class diagram related question #2, among the 41 respondents, the majority, 37 (88%) of respondents indicated that they strongly agreed or agreed that UML class diagrams

helped them to better understand Object-Oriented design concepts as shown in Figure 55 (mean agreement rating = 4.39).

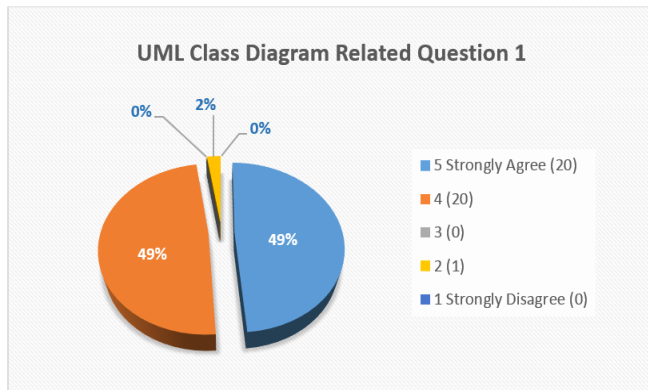


Figure 54. UML class diagram related question 1

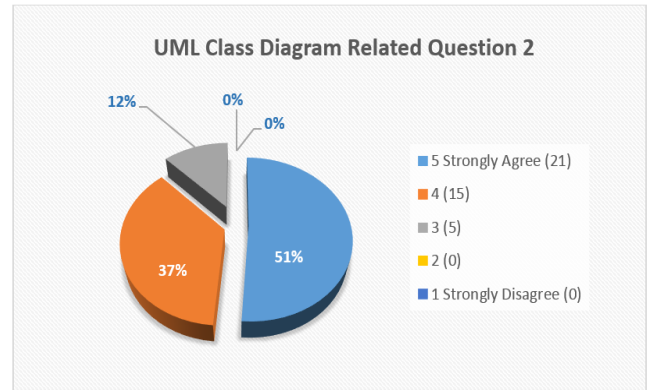


Figure 55. UML class diagram related question 2

Question 3: Please add any other comment on the “UML class diagrams”.

Overall, student respondents were satisfied with the aspect of static visualization in both compact and detailed UML class diagrams provided in JavelinaCode. As an illustration of this point, general responses include:

- *Those are good to understand.*
- *It helps in better understanding of code.*
- *The UML diagram is an easy way to understand how the execution works.*
- *Very good idea.*
- *With the UML diagram, I was quickly able to understand the structure of program,*
- *Useful and self-explanatory.*
- *They help to understand how the Java programs are structured.*
- *UML Diagrams helps us to understand the code structure in a simple and better way.*
- *It helps to understand the flow of the program.*
- *Very self-explanatory and great visualizations. Clean interface and very nice flow.*
- *Nice feature!*

Aspect of dynamic run time visualization in program execution

Question 1: I understand the “dynamic run time visualization of Java program execution”.

For the run time visualization related question #1, among the 41 respondents, 40 (97%) respondents indicated that they strongly agreed or agreed that I understood the “dynamic run time visualization of Java program execution provided in JavelinaCode as shown in Figure 56 (mean agreement rating = 4.61).

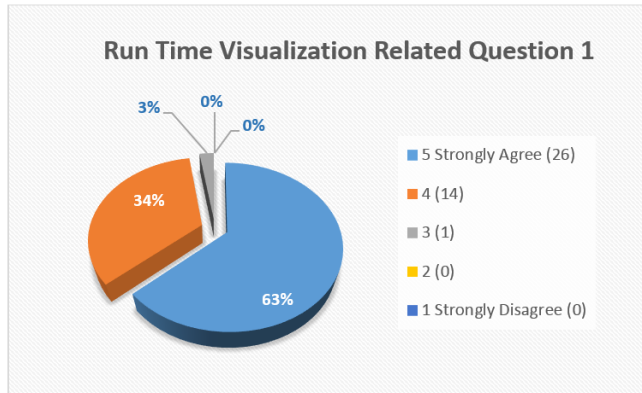


Figure 56. Run time visualization related question 1

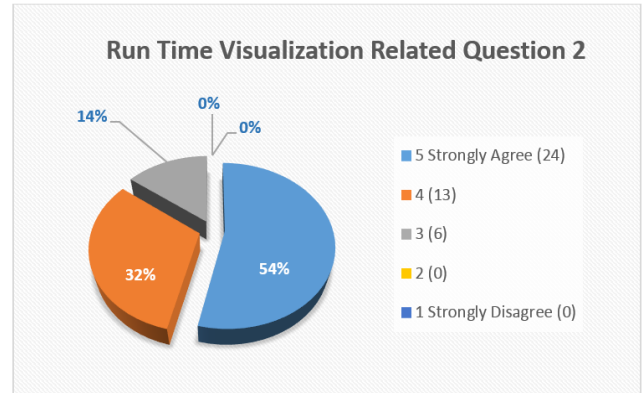


Figure 57. Run time visualization related question 2

Question 2: The “dynamic run time visualization of Java program execution” helps me correct and improve the quality of my program.

For the run time visualization related question #2, among the 41 respondents, the majority, 37 (86%) respondents indicated that they strongly agreed or agreed that the “dynamic run time visualization of Java program execution” helped them correct and improve the quality of their program as shown in Figure 57 (mean agreement rating = 4.39).

Question 3: Please add any other comment on the “dynamic run time visualization of Java program execution”.

Overall, student respondents were satisfied with the aspect of run time visualization of program execution provided in JavelinaCode. As an illustration of this point, general responses include:

- *Gives better understanding of how code executes and what is output at each step.*
- *Dynamic run time visualization of Java program execution helps in understanding the logic of program.*
- *With the dynamic run time visualization it was very to understand the structure of classes.*
- *Freshmen can easily get interest on java.*
- *Code comprehension become easy by dynamic run time visualization.*
- *Easy to understand.*
- *Excellent.*
- *Dynamic visualization helps to better comprehend the program.*

Two aspects of synchronized static and dynamic visualizations

Question 1: Synchronized UML class diagram and run time visualization along with source code make it easier for me to comprehend Java program.

For the two aspects of synchronized static and dynamic visualization related question #1, among the 41 respondents, 38 (93%) respondents indicated that they strongly agreed or agreed that both visualizations together helped them to make easier to Java programs (Figure 58).

Question 2: Synchronized UML class diagram and run time visualization along with source code alleviate the intimidation of Java programming.

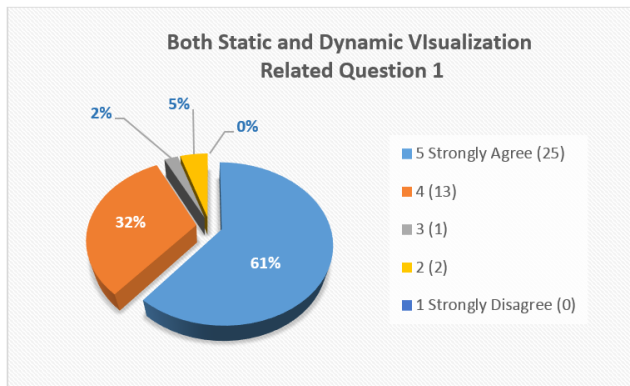


Figure 58. Both static and dynamic visualization related question 1

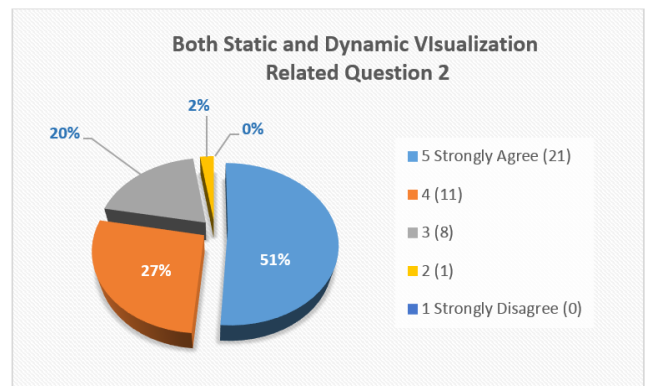


Figure 59. Both static and dynamic visualization related question 2

For the two aspects of synchronized static and dynamic visualization related question #2, among the 41 respondents, 33 (78%) respondents indicated that they strongly agreed or agreed that both visualizations together could alleviate the intimidation of Java programming (Figure 59).

Question 3: Which visualization aspect was more useful to comprehend the code?

For the two aspects of synchronized static and dynamic visualization related question #3, among the 41 respondents, the majority, 31 (76%) respondents indicated that both visualization

together was useful in comprehending code and 9 (22%) of them indicated that the dynamic run time visualization was useful (Figure 60).

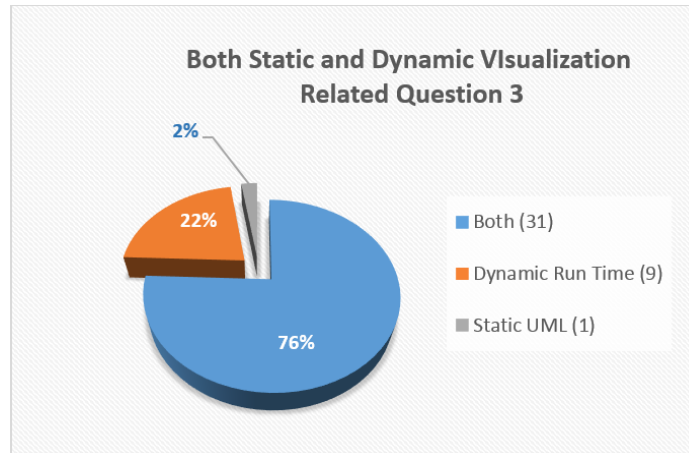


Figure 60. Both static and dynamic visualization related question 3

Figure 61 and Table 11 show the mean rating and percentage agreement for each of the visualization related questions. The mean ratings for all six questions ranged between 4.268 and 4.610, which can be considered as the respondents agreed highly for all questions. The respondents agreed most that the UML diagrams helped them to better understand overall structure of Java programs and they properly used and understood the “dynamic run time visualization of Java program execution provided in JavelinaCode.

6.4.5 Results and Discussion for Usability of the System

Figure 62 and Table 12 show the mean rating and percentage agreement for each of the usability related questions described in Table 10. The mean ratings for the first nine questions ranged between 4.415 and 4.683, which can be considered as the respondents mostly agreed for

all questions. Most respondents highly agreed that JavelinaCode is easy to use, they were comfortable in programming with it, and they'd like to use it for the rest of the subject.

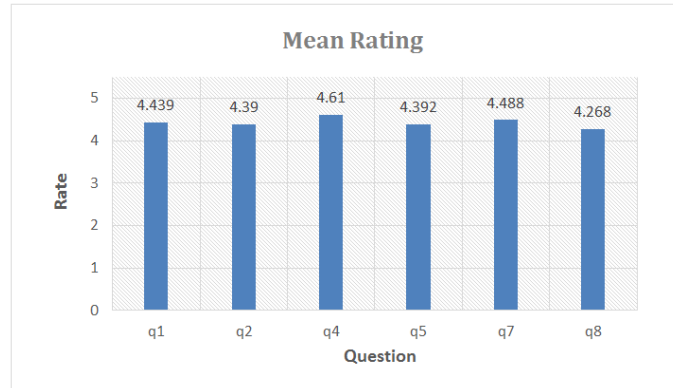


Figure 61. Mean rating for visualization related questions

Table 11. Mean rating and percent agreement for visualization related questions

Associated Objective	Question #	Strongly Disagree (1)	2	3	4	Strongly Agree(5)	Mean Rating	*Percent Agree
a	Question 1	0	1	0	20	20	4.439	97.56%
a	Question 2	0	0	5	15	21	4.390	87.80%
b	Question 4	0	0	1	14	26	4.610	97.56%
b	Question 5	0	0	6	13	22	4.392	85.37%
c	Question 7	0	2	1	13	25	4.488	92.68%
c	Question 8	0	1	8	11	21	4.268	78.05%

*Percent Agree (%) = Strongly Agree (5) and Agree (4) responses combined.

In the last three questions of the questionnaire, student participants were asked to provide additional suggestions or comments on the JavelinaCode system regarding any difficulties, key benefits, and needed improvements.

Question 10: Please specify any difficulties or problems you have encountered while using JavelinaCode?

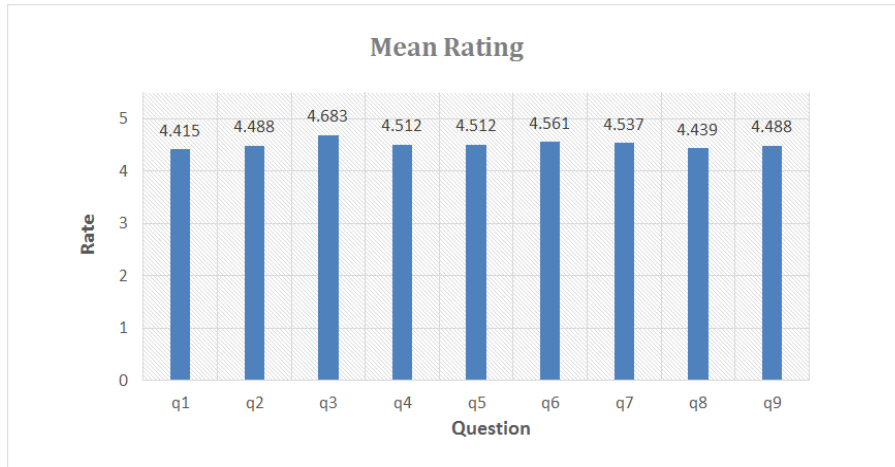


Figure 62. Mean rating for usability related questions

Table 12. Mean rating and percent agreement for usability related questions

Associated objective	Question #	Strongly Disagree (1)	2	3	4	Strongly Agree(5)	Mean Rating	*Percent Agree
d	Question 1	0	0	6	12	23	4.415	85.37%
e	Question 2	0	0	6	9	26	4.488	85.37%
d	Question 3	0	0	3	7	31	4.683	92.68%
e	Question 4	0	1	4	9	27	4.512	87.80%
e	Question 5	0	1	5	7	28	4.512	85.37%
d	Question 6	0	0	3	12	26	4.561	92.68%
e	Question 7	0	0	3	13	25	4.537	92.68%
e	Question 8	0	1	2	16	22	4.439	92.68%
e	Question 9	0	0	4	13	24	4.488	90.24%

*Percent Agree (%) = Strongly Agree (5) and Agree (4) responses combined.

For the question regarding any difficulties or problems using JavelinaCode, some student respondents have pointed out that the speed of visualizing the code was little slow and the error messages were not informatively displayed. 11 respondents indicated no difficulties or problems. As an illustration of this question, the following are general responses:

- *Errors during the execution of the program are not specific. Where the error has been made is not showing and its showing some piece of code which is not understood.*
- *While deleting a class that was created by mistake it faced a difficulty in removing it from the work-space.*
- *System take much time to visualize the code.*
- *Everything was good but to find out the specific errors occurred in program.*
- *The machines we used were a little slow.*

Question 11: Please specify key benefits of using JavelinaCode.

For the question specifying key benefits of using JavelianCdoe, 23 respondents positively commented. Overall, they were satisfied using JavelinaCode, which is simple, user friendly, understandable, and easy to use. They mentioned that both UML class diagrams and run time visualization techniques helped them understand the code better, the flow of program and the structure of inheritance and classes. Many of them also commented that one of the key features of using JavelinaCode is online with no installation of software, which they can use and benefit from it on any device to compile software and test programs. As an illustration of this question, the following are general responses:

- *UML class diagrams and visualization tool adds the beauty to JavelinaCode which makes understanding of the program easily.*
- *Simple and understandable.*
- *User friendly.*
- *The Key benefits of using the JavelinaCode is having the facility to work online and get a UML understanding of the code.*
- *Excellent to use.*
- *UML diagram.*
- *UML diagrams and visualization makes it better.*

- *Helps in understanding the code better through visualization.*
- *Good UI and easy to use.*
- *It makes me easier to understand the code and interaction between code and how it behaves.*
- *Easy and simple.*
- *Online.*
- *Would help to understand the java code step by step.*
- *We can learn java code perfectly .we can understand the java code.*
- *Clarity on java oops concepts.*
- *Dynamic visualization.*
- *CAN UNDERSTAND CLEARLY AND USER FRIENDLY.*
- *Can understand clearly.*
- *Dynamic Visualization and Run time helped me to understand the working of the code.*
- *UML Visualization makes it a simpler task in debugging code and to understand the program flow.*
- *It is simple and user friendly because of visualization I could easily get the outputs and other necessary stuff really easy instead of understanding each and every line of the code.*
- *It helps me to understand my code better, the flow of the program and structure of inheritance and classes.*
- *Using JavelinaCode helps in understanding the code flow. This in turn helps in writing better code.*
- *Software is online, no installation of software on my PC, online debugging is a plus. I can use this from any device to compile software and test small programs.*
- *Easy file creation, entire project is stored online, program tracer feature*
- *The UML Diagrams are very nice and helpful and the compiler setup is simple and not hard to use.*
- *I think the run time visualization is the best.*
- *It will be easier for professors to show us a program and will be helpful to explain to students how that program will run in a step-by-step process.*

- *Easier to follow along with the coding.*

Question 12: What do you think needs most important improvement on JavelinaCode, and why?

For the question asking any improvement on JavelinaCode, a few respondents commented that specifying the errors during editing programs and after executing the programs can be improved, and the execution speed can be also improved. The supporting of other programming languages was a good suggestion by two respondents. As an illustration of this question, the following are general responses:

- *Specifying the error during the typing of the program or at the end of execution such that it shows the exact line where the mistake has been done.*
- *It may helpful higher than previous, if we make it as more user friendly.*
- *Able to visualize proper flow of execution*
- *The LOOK & Feel can be better along with a user tutorial to understand the working of the IDE. The speed of the Execution can be improved!*
- *Responsiveness*
- *Execution speed can be improved*
- *Errors solving*
- *Run-time and Execution.*
- *Visualize Program can be improved by adding more details and it lags a little bit so can improve that also.*
- *It's perfect.*
- *Satisfied with the site.*
- *Support for more languages.*
- *Supporting other languages would be a very good enhancement. Also, you may add various coding problems to the repository and allow user to solve those and submit. Give*

a detailed statistics of the code considering the run time complexity and space complexity. Solution optimization techniques.

6.4.6 Results and Discussion for Associated Objectives

To assess whether using JavelinaCode with synchronized static and dynamic visualization could contribute to achieve its goal of meeting user's needs, evaluation results for objectives established in section 6.4.1 are analyzed. Table 13 demonstrates which objective is associated and addressed by which questions, and shows the mean rating and percent agreement for each objective from the combined data associated with.

Objective a: Do the UML class diagrams in JavelinaCode support student's understanding of object-oriented concepts?

This objective was addressed by the visualization related questions 1 and 2. 41 participants completed the questions and 92.68% of them strongly agreed or agreed that the UML class diagrams in JavelinaCode helped them understand object-oriented concepts (mean agreement rating = 4.415).

Objective b: Does the run-time visualization in JavelinaCode support students' understanding of object-oriented programming?

This objective was addressed by the visualization related questions 4 and 5. 91.46% of the participants strongly agreed or agreed that the run-time visualization in JavelinaCode supported their understanding of object-oriented programming (mean agreement rating = 4.501).

Table 13. Mean rating and percent agreement for associated objectives

Associated Objective	Questions	Strongly Disagree (1)	2	3	4	Strongly Agree(5)	Mean Rating	*Percent Agree
a	Visualization related Questions 1 & 2	0	1	5	35	41	4.415	92.68%
b	Visualization related Questions 4 & 5	0	0	7	27	48	4.501	91.46%
c	Visualization related Questions 7 & 8	0	3	9	24	46	4.378	87.5%
d	Usability related Questions 1, 3, & 6	0	0	12	31	80	4.553	91.74%
e	Usability related Questions 2, 4, 5, 7, 8, & 9	0	3	24	67	152	4.496	89.02%

*Percent Agree (%) = Strongly Agree (5) and Agree (4) responses combined.

Objective c: Does the JavelinaCode system make learning of object-oriented program easier?

This objective was addressed by the visualization related questions 7 and 8. 87.5% of the participants strongly agreed or agreed that JavelinaCode system made their learning of object-oriented program easier (mean agreement rating = 4.378).

Objective d: Is the JavelinaCode system easy to use?

This objective was addressed by the usability related questions 1, 3, and 6. 91.74% of the participants strongly agreed or agreed that JavelinaCode system was easy to use (mean agreement rating = 4.553).

Objective e: Are students satisfied and comfortable using the JavelinaCode system?

This objective was addressed by the usability related questions 2, 4, 5, 7, 8, and 9. 89.02% of the participants strongly agreed or agreed that they were satisfied and comfortable using JavelinaCode system (mean agreement rating = 4.496).

7 Conclusion and Future Work

The motivations of this research are that the difficulty for students to learn the Java programming language, a lack of programming environments for student programmers, and the difficulties of installing IDEs or plug-ins and setting up system environments.

While the Java language has been widely adopted as an introductory programming course in Computer Science, it has been known for being complex and difficult for students to learn due to its underlying Object-Oriented (OO) concepts and principles [19, 38]. Visualizations in various formats have been added to programming environments to help students better understand source code. Some only provide visual notations without source code, while others support a single aspect, structural or functional behavior, of the program. Despite all the efforts made, it is still difficult to match directly the source code to its visual notation due to separate windows for code and illustrations [7]. In addition, most IDEs including Eclipse and NetBeans provide static views of the program source code, but studies have pointed out that with only the static information, it is hard to understand the runtime behavior of the program, especially when polymorphism and dynamic binding are present in the code [7, 38]. It has been suggested to provide an approach which combines static and dynamic information together with the source code, which could reduce the burden of a complex mental simulation in the programmer's mind. Studies have also found that novice programmers often face difficulties in installing IDEs and Java Development

Kit (JDK) and in setting up and modifying system environment variables on their local machines [4, 39]. Downloading and installing software is a mandatory process to even start programming.

In order to deal with these issues, this research proposed an approach to provide source code along with static structural and dynamic behavioral visualizations in a web-based programming environment, JavelinaCode. The aim of this approach is to help student programmers better understand static structure and runtime behavior of a Java program, and to improve their ability to comprehend object-oriented programming concepts, thereby reducing their cognitive workload in Java programming through an effective development environment.

Using JavelinaCode, student programmers can write Java programs directly in a frontend web browser without any software or plug-in installation. They are provided with a view of the static state of a Java program in UML class diagrams and the dynamic run-time state of the program by stepping forward and backward through program execution. A case study simulating the yo-yo effect has revealed that our approach is useful, in particular, to easily trace and detect an object flow anomaly caused by method overriding and polymorphism. A preliminary comparison test result also has shown that through our web-based platform-independent environment, student programmers are freed from concern with continuous version changes and evolutions of the Java language, plug-ins, and operating systems.

To investigate the educational effectiveness of JavelinaCode with static and dynamic visualizations, both quantitative and qualitative evaluation experiments were designed. In the quantitative evaluation, two controlled experiments were conducted to test hypotheses on student performance on OO programming tasks. The null hypotheses were a) having both static and

dynamic visualizations available in JavelinaCode would not impact the correctness of solving problems, and b) having both static and dynamic visualizations available in JavelinaCode would not impact the time for solving programming problems. The goal of the experiments was to determine if students would be able to write Java program more accurately and in less time using the synchronized static and dynamic visualizations provided in JavelinaCode. The experiments were conducted using different levels of difficulty on program understanding, i.e., one for a relatively easy project and the other for a relatively hard project while both incorporate fundamental OO concepts such as inheritance and polymorphism. In the qualitative evaluation, the student feedback on the usability of JavelinaCode interface was gathered and evaluated by means of a debriefing usability related questionnaire. This was to investigate whether the JavelinaCode environment would help students make their OOP learning easier and help their understanding of OO concepts, and whether the interface would contribute to meeting its goals of providing satisfaction.

The results of the quantitative evaluation from the experiments support the conclusion that students in the experimental group using two aspects of visualizations in JavelinaCode consistently performed better to questions on program understanding than the controlled group. The application of t-tests rejects the first null hypothesis and accepts the alternative hypothesis meaning the correctness is statistically and significantly increased by the availability of both static and dynamic visualizations along with source code in JavelinaCode. This is evident that the p-values of 0.025 and 0.027 from Experiment 1 and Experiment 2 respectively are for the relatively easy project and 0.006 is for the relatively hard project considerably. All are lower than $\alpha = 0.05$. Therefore, the statistical analysis of the experimental data supports the conclusion that having

both static and dynamic visualizations in JavelinaCode does positively impact the correctness of solving problems.

In testing the second hypothesis regarding the response time, the statistical analysis from Experiment 2 reveals that, for the relatively hard project (yo-yo problem project), there is a significant difference between the controlled group and the experimental group. While the result of t-test does not accept the second alternative hypothesis, it rejects the second null hypothesis, which means that the response time is significantly increased by the availability of both visualizations in JavelinaCode with the p-values of 0.012 from Experiment 2. This is an interesting finding of how both visualizations did affect student understanding on program execution. Students took longer to answer, in particular, the relatively difficult questions using the visualizations supported in JavelinaCode, which led to higher accuracy for answering the questions correctly.

The results of the qualitative evaluation support the positive effect of JavelinaCode on helping students better understand OO concepts and meeting goals of providing comfortability and satisfaction. From the observations of visualization related questions during the qualitative evaluation, the mean ratings of all six questions remain high and consistently range between 4.268 and 4.610. With the percentage agreements ranging between 78.05% and 97.56%, the subjects strongly agreed or agreed that UML class diagrams helped them better understand overall structure of Java programs (mean agreement rating = 4.4439) and Object-Oriented design concepts (mean agreement rating = 4.390), they understood the “dynamic run time visualization of Java program execution” (mean agreement rating = 4.610), the “dynamic run time visualization of Java program execution” helped them improve the quality of the program (mean

agreement rating = 4.392), and the synchronized UML class diagram and run time visualization made it easier for them to comprehend Java program (mean agreement rating = 4.488) and alleviated the intimidation of Java programming (mean agreement rating = 4.268).

From the observations of the usability related questions, the mean ratings are high and consistently ranging between 4.415 and 4.683 along with high percentage of agreements (78.05% - 97.56%) throughout all questions. With percentage agreements ranging between 85.37% and 92.68%, the subjects strongly agreed or agreed that JavelinaCode was easy to use (mean agreement rating = 4.683), they enjoyed the time spent using JavelianCode (mean agreement rating = 4.512), working with JavelinaCode was satisfying (mean agreement rating = 4.512), the way that JavelinaCode presented was clear and understandable (mean agreement rating = 4.561), and they were comfortable in programming with JavelinaCode (mean agreement rating = 4.537).

For the associated objectives to access whether or not the visualizations in JavelinaCode could contribute to achieving its goal of meeting user's satisfaction, the mean ratings of all objectives remain high and consistently ranging between 4.378 and 4.553. With percentage agreements ranging between 87.5% and 92.68%, the subjects strongly agreed or agreed that the UML class diagrams in JavelinaCode helped them understand object-oriented concepts (mean agreement rating = 4.415). The run-time visualization in JvaelinaCode also supported their understanding of object-oriented programming (mean agreement rating = 4.501), made their learning of object-oriented program easier (mean agreement rating = 4.378), was easy to use (mean agreement rating = 4.553), and they were satisfied and comfortable using JavelinaCode system (mean agreement rating = 4.496).

For the questions regarding any difficulties or improvements in using JavelinaCode, a few subjects have pointed out that the speed of visualizing the code was a little slow and error messages were not informatively displayed. Therefore, specifying detailed error messages during editing and executing programs can be improved, and the execution speed of visualization can also be improved. However, while only several students expressed their concerns about the speed and error message displaying in JavelinaCode, thirty-three students did specify key benefits of using JavelinaCode. It was clearly observed that they were very satisfied using JavelinaCode - it is simple, user friendly, understandable, and easy to use. They mentioned that both UML class diagrams and run time visualization techniques helped them understand the code better, i.e., flow of program and the structure of inheritance and classes. Many of them also commented that one of the key features of using JavelinaCode is its online accessibility with no software installation, which can be used on any device.

7.1 Contributions and Benefits

This research has presented an approach to better support OOP learning in Java by providing synchronized static and dynamic visualizations of source code in a web-based programming environment, JavelinaCode. Following the best practices of OOP, its design principles are: providing easy access to the programming environment, creating a programming environment that is easy to use, making the environment source code and user centered, providing static visualization of structural information of a program and dynamic visualization of functional information of program execution synchronized along with the source code, and providing structural and functional feedback in real time. Through case studies and comparison tests, it has

been found that our approach is useful to uncover and clarify data flow anomalies and to offer a worry-free programming environment.

Therefore, the main benefits and contributions of this research are: a) Student programmers can use a web browser to program in Java without any software or plug-in installation or setting up or modifying system environment variables on their local machine. b) Student programmers do not need any kind of storage system to keep and manage their project files. They are freed from concern about continuous version changes of Java language, IDEs, and operating systems. c) Student's cognitive workload can be reduced and program comprehension can be enhanced with the synchronized visualization along with source code. d) Student's programming patterns can be saved in log files, their behavioral patterns in programming can be monitored and analyzed in the future.

7.2 Future Work

Based on the work and contributions of this research, important goals for future work of JavelinaCode are to extend its use and embed more functionality to the current version. Plans are to include a One-To-One (Many) Virtual Mentoring (VM) capability, a student learning analytics, a simplified run time visualization of program execution, and an optimized sequence diagram.

The Virtual Mentoring System (VMS) will consist of two main components, a screen sharing capability for presenting a student's code in real time to a course instructor or a teaching assistant (TA) and a list of commonly used coding segments. The student will be provided a mentoring session room to communicate with the instructor or TA who can instantly join the

room and connect to the code editor to help the student complete an assignment. The student will be also provided a list of code segments from which to make a selection. The selected segment can be imported to the student's code at the cursor's current position in the editor window. The objectives of VMS are to give students real-time assistance on coding and an easier means of implementing code segments that are unfamiliar to them. This serves to reduce the amount of workload dedicated to memorization and allows for greater focus on the intuitive elements of programming. The VMS will provide rapid feedback to students, improve coding skills, and enhance logical reasoning and critical thinking skills. The overall expectation is an increased completion and retention rate for Computer Science students.

Student's programming patterns can be saved in log files to measure, collect, analyze, and report of data about their behavioral patterns in programming, which can be monitored and analyzed. The purpose of student's learning analytics is to enhance student's programming, logical reasoning, and thinking skills through individualized mentoring based on individual learning speed and ability.

As indicated from the qualitative evaluation, one improvement that can be made to JavelinaCode is the speed of visualizing the run time program execution. The slow speed of the visualization is due to Java program execution in memory on the server. The speed can be improved by removing all unnecessary steps in the visualization and visualizing only the key values of, for instance, variables and objects or replacing it by an object diagram, which can still provide necessary information about objects and their elements in such a simple way. A reasonably better performance in speed can be achieved with careful design consideration.

As source code becomes large and complex, current sequence diagrams generated by the source code provide limited execution coverage in such a way that they contain redundant, dead and faulty driven methods, therefore the size and complexity of the diagrams is getting increased. Optimized sequence diagrams are being developed in order to reduce complexity and provide complete behavior of the system [52, 53]. The optimized sequence diagrams, which are less complex and provide more detailed description of functionality of a system, are planned to be embedded in the JavelinaCode system. This is to effectively represent complete run-time behavior of the project which can help student programmers in program comprehension.

8 References

- [1] The A-Z of Programming Languages (interviews with programming language creators), Computerworld, 2008-2010.
- [2] I. Lavy, R. Rashkovits, and R. Kouris, Coping with abstraction in object orientation with a special focus on interface classes, *Computer Science Education*, 19 (3):155-177, 2009.
- [3] N. Liberman, C. Beerli, and Y. Ben-David Kolikant, Difficulties in learning inheritance and polymorphism, *Trans. Computer Education*, 11(1):4:1-4:23, Feb. 2011.
- [4] Nghi Truong, Peter Bancroft, Paul Roe, A Web Based Environment for Learning to Program, ACSC '03 Proceedings of the 26th Australasian computer science conference - Volume 16, pp. 255-264.
- [5] Dianne Hagan and Selby Markham, "Teaching Java with the BlueJ Environment," Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference ASCILITE, 2000.
- [6] Philip J. Guo, "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education," Proceedings of the ACM Technical Symposium on Computer Science Education, March 6-9, 2013.
- [7] Noa Ragonis and Mordechai Ben-Ari, "On Understanding the Statics and Dynamics of Object-Oriented Programs," SIGCSE '05, February 23-27, 2005, St. Louis, Missouri, USA.
- [8] Michael Kölling, "The design of an object-oriented environment and language for teaching," PhD Dissertation, University of Sydney, 1999.

- [9] Michael Kölling and John Rosenberg, "Guidelines for Teaching Object Orientation with Java," The Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001), Canterbury, 2001.
- [10] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg, "The BlueJ system and its pedagogy," Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology, Vol 13, No 4, Dec 2003.
- [11] Van Haaster, K. and Hagan, D., "Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool." Information Science & Information Technology Education Joint Conference, Rockhampton, QLD, Australia, June, 2004.
- [12] Jens Bennedsen, "Teaching and Learning Introductory Programming - A Model-Based Approach," PhD Dissertation, University of Oslo, 2008.
- [13] Cornelis Huizing, Ruurd Kuiper, Christian Luijten, and Vincent Vandalon, "Visualization of Object-Oriented (Java) programs," CSEDU 2012 4th International Conference on Computer Supported Education, pp. 65-72, 2012.
- [14] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari, "Visualizing Programs with Jeliot 3," Proceedings of the International Working Conference on Advanced Visual Interface AVI 2004, May 2004.
- [15] C. A. Luijten, "Interactive visualization of the execution of object-oriented programs. Master's thesis," Eindhoven University of Technology, 2009.
- [16] Andrés Moreno and Mike S. Joy, "Jeliot 3 in a Demanding Educational Setting", Fourth International Program Visualization Workshop, June 29-30, 2006.
- [17] Nuttanont Hongwarittorn and Donyaprueth Krairit, "Effects of Program Visualization (Jeliot3) on Students' Performance and Attitudes towards Java programming," The spring 8th International Conference on Computing, Communication and Control Technologies, 2010.

- [18] Andre L. Santos, “AGUIA/J: A Tool for Interactive Experimentation of Objects,” ACM ITiCSE’11, June 27-29, 2011, Darmstadt, Germany.
- [19] Andre L. Santos, “Novel Interaction Metaphors for Object-Oriented Programming Concepts,” 14th International Conference on Computer Science Education, Koli, Finland, 2014.
- [20] Andre L. Santos, An open-ended environment for teaching Java in context, ACM ITiCSE’12, 17th Annual Conference on Innovation and Technology in Computer Science, pp. 87-92, Haifa, Israel, 2012.
- [21] Paul V. Gestwicki, Bharat Jayaraman. JIVE: Java Interactive Visualization Environment. In Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 226–228, 2004.
- [22] Paul V. Gestwicki, Bharat Jayaraman. Methodology and Architecture of JIVE. In Proceedings of the 2005 ACM Symposium on Software Visualization (SOFTVIS), pp. 95–104, 2005.
- [23] Bharat Jayaraman, Charlotte M. Baltus. Visualizing Program Execution. In Proceedings of the 1996 IEEE Symposium on Visual Languages, pp. 30–37, 1996.
- [24] J. Swaminathan, Kishor Kamath, and Bharat Jayaraman: Towards program execution summarization: Deriving state diagrams from sequence diagrams. Seventh Intl Conference on Contemporary Computing, pp. 299-305, August 2014.
- [25] Ace, The High Performance Code Editor for the Web, <http://ace.c9.io/>.
- [26] PlantUML, <http://plantuml.com/>.
- [27] Java Visualizer, http://www.cs.princeton.edu/~cos126/java_visualize/.
- [28] Online Python Tutor, <http://www.pythontutor.com/>.

- [29] Amazon Web Service, <https://aws.amazon.com/>.
- [30] Codecall, <http://codecall.net/2014/11/12/11-best-desktop-web-ides-java-programmers/>.
- [31] T. Dean Hendrix, James H. Cross, and Larry A. Barowski, “An extensible framework for providing dynamic data structure visualizations in a lightweight IDE,” Proceedings of the 35th SIGCSE technical symposium on Computer science education, March 03-07, 2004, Norfolk, Virginia, USA.
- [32] James H. Cross, T. Dean Hendrix, Jhilmil Jian, and Larry A. Barowski, “Dynamic object viewers for data structures,” Proceedings of the 38th SIGCSE technical symposium on Computer science education, March 07-11, 2007, Covington, Kentucky, USA.
- [33] Larry A. Barowski, “A Low-Effort Animated Data Structure Visualization Creation System,” Ph.D. dissertation, August, 2014, Auburn University, Auburn, AL.
- [34] L. Montgomery, J. H. Cross, T. D. Hendrix, and L. A. Barowski, “Testing the jGRASP Structure Identifier with Data Structure Examples from textbooks,” In Proceedings of the 46th Annual Southeast Regional Conference, Auburn, Al, 2008.
- [35] J. Jian, J. H. Cross, T. D. Hendrix, and L. A. Barowski, “Experimental Evaluation of Animated-Verifying Object Viewers for Java,” in Proceedings of SoftVis 2006, Brighton, UK, 2006.
- [36] J. H. Cross, T. D. Hendrix, D. A. Umphress, L. A. Barowski, j. Jian, and L. Montgomery, “Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches,” ACM Transactions on Computing Education, vol. 9, no. 2, pp. 13:1-13:32, June 2009.
- [37] J. H. Cross, T. D. Hendrix, L. A. Barowski, and D. A. Umphress, “Dynamic Program Visualizations – An Experience Report,” Proceedings SIGCSE 2014, Atlanta, GA, March 5-8, 2014, 609-61.

- [38] Röthlisberger, David, et al. "Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks." *Software Engineering, IEEE Transactions on* 38.3 (2012): 579-591.
- [39] Lewis, S. and Watkins, M. (2001) In 5th Java in the Computing Curriculum Conference (JICC 5) South Bank University, UK.
- [40] Jeong Yang, Young Lee, David Hicks, and Kai Chang, "Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization", *IEEE Frontiers in Education 2015: Launching a New Vision in Education Engineering*, pp. 806-810, 2015.
- [41] B. de Alwis and G.C. Murphy, "Answering Conceptual Queries with Ferret," *Proc. 30th Int'l Conf. Software Eng.*, " pp. 21-30, 2008.
- [42] S.P. Reiss, "Visualizing Java in Action," *Proc. ACM Symp. Software Visualization*, pp. 57-66, 2003.
- [43] W. Löwe, A. Ludwig, and A. Schwind, "Understanding Software —Static and Dynamic Aspects," *Proc. 17th Int'l Conf. Advanced Science and Technology*, pp. 52-57, 2001.
- [44] Offutt, J., Alexander, R., Wu, Y., Xiao, Q., Hutchinson, C.: A fault model for subtype inheritance and polymorphism. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering*. pp. 84-93. *ISSRE '01*, IEEE Computer Society, Washington, DC, USA (2001).
- [45] Jeffrey Rubin and Dana Chisnell, *Handbook of Usability Testing, Second Edition: How to Plan, Design, and Conduct Effective Tests*, Wiley Publishing, Inc., 2008.
- [46] Almstrum, V. L., Dale, N., Berglund, A., Granger, M., Currie Little, J., Miller, D. M., et al. (1996). *Evaluation: Turning technology from toy to tool: Report of the Working Group on Evaluation*, In *ITiCSE '96: Proceedings of the 1st conference on integrating technology into computer science education*, Volume 28 Issue SI, 1996.

- [47] Ethical Principles of Psychologists and Code of Conduct, <http://www.apa.org/ethics/code/>.
- [48] Erik Frøkjær, Morten Hertzum, and Kasper Hornbæk, Measuring usability: are effectiveness, efficiency, and satisfaction really correlated?, *Proceeding CHI '00 Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 345-352, 2000.
- [49] ISO 9241-11:1998(en), Ergonomic requirements for office work with visual display terminals (VDTs) — Part 11: Guidance on usability.
- [50] ISO 9241-210:2010(en), Ergonomics of human-system interaction - Part 210: Human-centered design for interactive systems.
- [51] Jeong Yang, Young Lee, and David Hicks, “Synchronized Static and Dynamic Visualization in a Web-Based Programming Environment,” *IEEE International Conference on Program Comprehension (ICPC)*, May 16-17, 2016.
- [52] Madhusudan Srinivasan, Jeong Yang, and Young Lee, “Case Studies of Optimized Sequence Diagram for Program Comprehension,” *IEEE International Conference on Program Comprehension (ICPC)*, May 16-17, 2016.
- [53] Madhusudan Srinivasan, Young Lee, and Jeong Yang, “Enhancing Object-Oriented Programming Comprehension using Optimized Sequence Diagram,” *IEEE Conference on Software Engineering Education and Training (ICSEE)*, pp. 81-85, April 6-8, 2016.