Impact of Static and Dynamic Visualization in Improving Object-Oriented Programming Concepts

Brandon Earwood, Jeong Yang, Young Lee
Electrical Engineering and Computer Science
Texas A&M University-Kingsville
Kingsville, TX, USA
brandon.earwood@students.tamuk.edu,
{jeong.yang, young.lee}@tamuk.edu

***Abstract*— In computer science education, the programming environment is just as crucial as the classroom environment. Students must be provided with adequate tools for developing an understanding of computer science paradigms. These paradigms include object-oriented programming (OOP) concepts, such as inheritance, polymorphism, and encapsulation. JavelinaCode is a web-based integrated development environment (IDE) for Java programming that provides students with both a static and dynamic visualization of code; the static structure of code at compile time and the dynamic execution of code at run time. The objective of this research is to present the differences between JavelinaCode and similar programming environment tools to demonstrate how JavelinaCode improves students' understanding of OOP concepts. Comparative analysis is conducted between JavelinaCode and programming environment tools, including BlueJ, Jeliot 3, AguiaJ, JIVE, and jGRASP. Each of these tools is 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. The results of the comparative analysis show that JavelinaCode provides students with a more effective tool for understanding OOP concepts than those considered. This is due to a simplified set up process and user interface and a better integrated set of visualizations. The combination of these factors contributes to a programming environment that emphasizes developing and running code.**

Keywords—object-oriented programming, static visualization, dynamic visualization, programming education, web-based programming environment

I. INTRODUCTION

In the scope of computer science, becoming familiar with fundamental programming concepts is dependent on the IDE and related tools. Often, students encounter difficulties such as the programming language syntax, programming environment, and the need to develop problem solving skills [6]. For this research, the programming concepts considered are related to OOP. OOP modularizes a program into a set of classes that can be related to some real world entities. Instances of a class, also known as objects, are constructed and used by users. Generally, a programming language allows each class to be its own distinct file. The terms class and file will be used interchangeably for this reason. A useful provision in many IDEs and tools is some static or dynamic visualizations, which may include various universal modeling language (UML) diagrams or some line by line program execution visuals. These visualizations help students better understand how syntax translates to the construction and use of instances of classes, how each line of code influences the programming environment, and how to effectively utilize programming skills. The three primary UML diagrams used among these tools are class diagrams, object diagrams, and sequence diagrams. Class diagrams visually present each class of a program and how classes are interrelated. Object diagrams allow for a more specific interpretation of a particular program, showing instances of each class that have been generated. Sequence diagrams show the initialization of objects, instances of classes, and method calls as well as the lifetime of each of these elements.

Since inheritance and polymorphism are two of the most fundamental concepts of OOP, these concepts are emphasized in this paper. Inheritance refers to the hierarchy of class structures in programming, which allows lower level classes to inherit certain properties, such as variables or methods, from higher level classes. Polymorphism, in the case of OOP, allows for interchangeable data typing in which a higher level class may be of a type of lower level class. Lecturers generally present some analogies for relating these concepts to real world entities. For example, a person at a university might be either a professor or a student. A class named Person could be created as well as two classes for Professor and Student. Variables and methods for the Person class are inherited by the Professor and Student class. These analogies, however, do not fully assist students in transitioning to practical programming applications. Many programming assignments will present multiple levels of inheritance, implement nonintuitive polymorphic statements, and present conflicting coding segments.

The comparative analysis section addresses these issues by analyzing various development tools: BlueJ, Jeliot 3, AguiaJ, JIVE, jGRASP, and JavelinaCode. These tools were selected due to the emphasis on source code with visualizations included. Tools such as Alice, Greenfoot, Lightbot, or Scratch, while also being web-based visualization tools, will not be used in these comparisons as they do not emphasize source code. Criteria to be considered include time and complexity constraints, similar to those used in the evaluation phase of [7].

II. COMPARATIVE ANALYSIS

JavelinaCode was compared against other development tools in terms of three criteria: time constraints, complexity

constraints, and the provision of static and dynamic visualizations. These factors are measured and compared to determine how students will be able to efficiently use their time and critical thinking skills for reading and writing code rather than being impeded by the set up and interface. The specifications of the computer system used to conduct this analysis are as follows: OS X Yosemite 10.10.5, 2.5 GHz Intel Core i5, 4 GB 1333 MHz DDR3. A stable internet connection was maintained at the Texas A&M University-Kingsville campus during the download and installation process. Download and install times as well as other characteristics of each tool are presented in table 1. These include 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.

TABLE I. IDE CHARACTERISTICS

IDE	Version	OS Compatibility	Plugin	File Size	Download Time	Install Time
Netbeans	8.1	Windows, Mac OS, Linux	N	275.6/678.9	0:25	3:40
Eclipse	4.5.1	Windows, Mac OS, Linux	N	45.7	0:19	3:10
BlueJ	3.1.6	Windows, Mac OS, Linux	N	188.3/368.8	2:11	N/A
Jeliot 3	3.7.2	Windows, Mac OS, Linux	N	1.6	0:02	N/A
AguiaJ	1.1.2	Windows, Mac OS, Linux	Y (Eclipse)	N/A	1:11	
JIVE	1.9.27	Windows, Mac OS, Linux	Y (Eclipse)	N/A	2:10	
jGRASP	2.0.1_09	Windows, Mac OS, Linux	N	5/10.6	0:01	0:20

Each tool is also compared on the quality of its interface in terms of usability. Both the fundamental design and the means of handling two Java projects, a PolyShape project and a test project for the Yoyo problem, are evaluated. The PolyShape project introduces some fundamental class hierarchy with a Polygon, Rectangle, Sphere, and Cylinder class. The latter three inherit some variables and methods from the former and have an overriding method for calculating area. The Yoyo problem is an anomaly that occurs when execution "bounces" up and down the class hierarchy, much like a yoyo, and encounters some unintentional modification to a method's behavior [12]. This problem serves as a primary example of the issues students face in practical programming applications. The project used to demonstrate this consists of an Employee class as well as a StaffEmployee and StudentEmployee class that inherit from Employee. The hourlyRate() method in StaffEmployee and StudentEmployee overrides that in Employee but fails to set a value for rate, resulting in an output of zero for the payment. The following sections provide a description of each tool's purpose and interface followed by an analysis of visualizations. This includes determining which visualizations are used, how well these are integrated with the source code, and how students are to interpret these visualizations to better understand any underlying OOP concepts. Table 2, at the end of this section, provides a brief overview of these comparisons.

A. BlueJ

BlueJ is an IDE designed with the intent of aiding novice programmers in understanding the abstractions associated with OOP [8]. The interface is found to be straightforward and intuitive for building a new project, writing code to classes, and compiling the project. Running a project, however, was not integrated with the other features and required a counterintuitive process of right-clicking the Main class and selecting the main method. The source code and output of 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. Files added to the project were connected based on relationships of inheritance and association. These relationships are graphically displayed using arrow notation that replicates a UML class diagram. Two primary issues with this class diagram are that no details about individual classes are displayed in this view and that the automatically generated relationships in BlueJ are not always necessarily correct and must be corrected manually. In order to observe the properties and methods of a class, students must use the source code of each file.

Relying on just the static visualization, it is difficult to determine the source of the error with the Yoyo problem. By setting a breakpoint at a meaningful line in the Main class, the student may 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 to analyzing the code but no visualization technique is employed. This makes stepping through lines of code more difficult to read and follow.

B. Jeliot 3

Jeliot 3 is a program animation system for teaching and learning elementary programming [3]. Source code is displayed along the lefthand side of the window. The buttons for compiling and executing the code are listed along the bottom of this window. Execution is performed line by line with an animation scheme called the theater in the center of the window to show how values are assigned to variables and how output is displayed to the appropriate window. Jeliot 3 does not support multiple files in a single project. All classes are specified in a single file, limiting students' understanding of how classes are distinct from one another.

While building the PolyShape project, there was a notable lack of some class or object diagram that may serve as a static visualization of the program. Visualization techniques for Jeliot 3 are restricted to the theater, offering a dynamic visualization in its running state or a partial static visualization while paused. Execution of PolyShape shows that 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.

Assuming students can identify the location of program error, those lines of code can be analyzed using the theater and stepping through each line. Students 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 students' assumptions about the program's execution and a trial and error method.

C. jGRASP

jGRASP is a lightweight IDE for program visualizations [10, 11]. It allows for standard program execution or the use of a canvas window to step through code line by line. Students may also generate a class diagram of the project that presents a simplified static visualization using a similar style to BlueJ. The canvas is displayed in a separate window from the source code. Program output is displayed at the bottom of the initial window.

A jGRASP tutorial [2] was referenced to add a new project to jGRASP for PolyShape. Execution of the program 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 a static visualization, generally an isolated object diagram with the name of the object, its variables, and associated values. Students may transition between the class diagram, object diagram, and source code in order to gain a full understanding of the program, but this process is disconnected by the use of multiple windows in jGRASP.

For handling the Yoyo problem, students can observe the three objects created near the end of the program's execution. By dragging each of these objects into the canvas, students can observe that the object of Employee has a nonzero rate while the objects of StaffEmployee and StudentEmployee each have a rate of zero. Analyzing the class diagram indicates the inheritance relationship between StaffEmployee and StudentEmployee to Employee.

D. AguiaJ

AguiaJ is an Eclipse plug-in designed to emphasize class instantiation. This tool embodies novel interaction metaphors to illustrate object-oriented programming concepts with first-class representations [4]. This is made possible by introducing a graphical environment for creating and controlling classes and objects interactively [5]. From what was observed, there were three windows for presenting an overview of the classes involved, holding instantiated objects of each class, and inputting a line of test code for the objects.

Classes and objects in AguiaJ are not bound to the runtime environment of the program. Objects are freely created and tested using either the appropriate button in the interface or inputting a line of constructor code. Students 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. AguiaJ offers students an incredibly useful and flexible static visualization of code.

Restrictions in AguiaJ became more noticeable while testing the Yoyo problem 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 other 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 students were to test lines of code from the current runtime environment, this would require stepping back and forth between the AguiaJ interface and the standard Java interface for Eclipse. To solve the Yoyo problem using AguiaJ, students 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.

E. JIVE

JIVE is a system that provides a novel approach to the runtime visualization and analysis of OOP [9]. To use JIVE in Eclipse, students must properly configure the debugger. 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 available. The interface for JIVE itself is separate from the source code and appears overly clustered. For example, two UML diagrams are included in the interface, an object and sequence diagram, and each is given its own set of buttons for stepping through the code. This is extraneous as students will still stay in the same location of the code on both diagrams.

Given the use of an object and sequence diagram in JIVE, as students step through the code using these visualizations, the execution of code can be observed based on instantiations and method calls. Considering the design space and view of the interface, it should be noted that windows can become cluttered with larger projects. This was noticeable in the PolyShape project, which creates only six shape objects. Students 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. This also makes debugging of large scale projects less time consuming.

The biggest issue with JIVE while testing the Yoyo problem is that output is not generated alongside a line by line step through of the code. Similarly, the default setting for the object diagram does not provide enough visual evidence of the expected output of the program. Observing the values of variables within a particular object requires setting the object diagram to objects with tables. This also better shows the relationship between objects by enclosing child classes within parent classes. Once the object diagram has been modified as such, students 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 type StaffEmployee or StudentEmployee.

F. JavelinaCode

The JavelinaCode interface presents a static visualization of the code as a class diagram on the lefthand side, the source code in the center, and a dynamic visualization on the righthand 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.

For PolyShape, during execution, the dynamic visualization was 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. The class diagram generated for the program may also be expanded in order to observe all details about classes, such as the individual properties and methods of each shape in the PolyShape project. The simplified class diagram is integrated in the same window as the source code while this detailed class diagram is displayed in a separate window.

By stepping through each line of code in the project for the Yoyo problem, it can be observed that rate was not properly set for StudentEmployee or StaffEmployee. Shown in figure 1, this is identified by the dynamic visualization showing a value of zero being assigned to rate for instances of these classes. Students are then able to analyze the class diagram to determine the appropriate relationship between classes. This provides some insight into the location of the error within the source code.

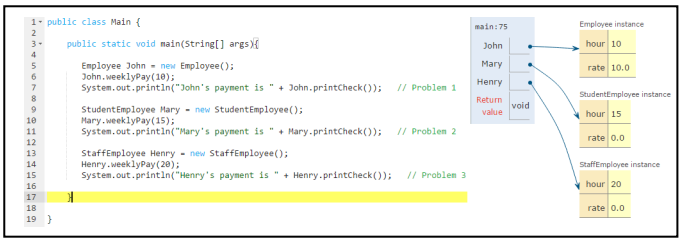

Fig. 1 Objects of Yoyo Problem in JavelinaCode

TABLE II. VISUALIZATION & INTERFACE COMPARISONS

IDE	Static Visualization	Dynamic Visualization	Integrated Interface
BlueJ	Yes - simple class diagram	No	No
Jeliot 3	No	Yes - animation scheme	Yes
jGRASP	Yes - simple class diagram	Yes - runtime execution w/ objects	No
AguiaJ	Yes - detailed object diagram	No	No
JIVE	Yes - detailed object diagram	Yes - sequence diagram	Yes
JavelinaCode	Yes - detailed class diagram	Yes - runtime execution w/ objects	Yes

III. STUDENT EVALUATION

The evaluation of JavelinaCode will be dependent on student responses to the tool based on experiences over a reasonable period of time. This involves developing well-reasoned experiments based on meaningful criteria in relation to the student's experiences and learning outcomes. There are two means by which the effectiveness of JavelinaCode may be measured. The first is based on the usability of the tool and interface, taking into consideration those criteria that would simplify the student's experience while accessing JavelinaCode and studying programs built in this interface. The second is based on the student's capacity to comprehend and understand code after having used JavelinaCode, focusing on student's general capacity to learn beyond the scope of what is presented to them during usage of JavelinaCode. In both cases, students would be exposed to JavelinaCode and an equivalent tool, such as one of those used in comparative studies.

For usability testing, observations must emphasize how students react to the environment and interface. Criteria to consider include the ease of access, ease of use, ease of immediate understanding, centralization on the user and areas that should hold the user's attention, and effective real time feedback. These criteria are discussed in the design principles of [6]. The argument in favor of JavelinaCode is that, based on the simplicity of the interface design and positioning of specific elements (source code and visualizations) within the students' view, the usability of this tool is far greater than the others considered in this study. For testing the students' capacity to understand programming concepts, the main criteria to observe is the cognitive workload. This is briefly addressed in [7] when discussing that one of the desires of JavelinaCode is reducing students' cognitive workload. An effective metric for analyzing cognitive workloads would be the number of words or sentences it takes students to explain what a program does and to determine output given a sample input. This is reinforced by a similar study in [1], in which students were asked to 'think out loud' while answering coding questions, as well as 'think-aloud protocols' discussed in [13] when dealing with students' structural or domain-specific knowledge. In this context, that domain would be OOP concepts. The style of metrics for either of these methods will be relatively similar. After allowing students to use JavelinaCode and one of the previously discussed tools as a control group, a set of questions related to understanding a particular piece of code will be given.

IV. CONCLUSION

Based on the comparative analysis, it can be argued that JavelinaCode provides students with an effective tool for learning OOP concepts. Through the use of static and dynamic visualizations to aid in analyzing coding projects, students can more easily identify abstractions such as polymorphic behavior, inheritance, and other OOP paradigms. Static visualizations provide students with a reference when analyzing the inheritance between classes, such as the relationships in a class diagram, and assessing problems likely to do with OOP related issues, such as the Yoyo 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 students to transition between identifying both the structure and behavior of object-oriented programs. JavelinaCode includes a class diagram with sufficient information about the relationships and individual characteristics of classes as opposed to tools such as BlueJ, Jeliot 3, or jGRASP. The runtime graphics presented in JavelinaCode provide an effective dynamic visualization not offered in tools such as BlueJ or AguiaJ. For tools besides JavelinaCode that offer both types of visualization techniques, JavelinaCode still 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 with jGRASP and JIVE. 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. Further applications will require fully integrating JavelinaCode into a course and comparing results based on students' understanding and completion of work against previous semesters.

REFERENCES

[1] Raymond Lister, "After the gold rush: toward sustainable scholarship in computing," Proceedings of the tenth conference on Australasian computing education, January 01-01, 2008, Wollongong, NSW, Australia.

[2] Projects, 1st ed. Auburn, AL, 2009, pp. pp. 7-2, 7-3, 7-4, http://www.jgrasp.org/tutorials187/07_Projects.pdf.

[3] M. Ben-Ari, R. Bednarik, R. Ben-Bassat Levy, G. Ebel, A. Moreno, N. Myller and E. Sutinen, "A decade of research and development on program animation: The Jeliot experience", Journal of Visual Languages & Computing, vol. 22, no. 5, pp. 375-384, 2011.

[4] Andre L. Santos, "Novel Interaction Metaphors for Object-Oriented Programming Concepts," 14th International Conference on Computer Science Education, Koli, Finland, 2014.

[5] Andre L. Santos, "Aguia/J: A Tool for Interactive Experimentation of Objects," ACM ITiCSE' 11, June 27-29, 2011, Darmstadt, Germany.

[6] Jeong Yang, Young Lee, David Hicks, and Kai Chang, "Enhancing Object-Oriented Programming Education using Static and Dynamic Visualization," IEEE Frontiers in Education 201: Launching a New Vision in Education Engineering, pp. 806-810, 2015.

[7] J. Yang, Y. Lee and D. Hicks, "Synchronized Static and Dynamic Visualization in a Web-Based Programming Environment,' IEEE International Conference on Program Comprehension (ICPC), May 16-17, 2016

[8] Michael Kolling, "The design of an object-oriented environment and language for teaching," PhD Dissertation, University of Sydney, 1999.

[9] Paul Gestwicki, Bharat Jayaraman, "Methodology and architecture of JIVE," Proceedings of the 2005 ACM symposium on Software visualization, May 14-15, 2005, St. Louis, Missouri.

[10] T. Dean Hendrix, James H. Cross, II, 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.

[11] James H. Cross, II, T. Dean Hendrix, Jhilmil Jain , 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.

[12] Offutt, J., Alexander, R., Wu, Y., Xiao, Q., Hutchinson, C., "A fault model for subtype inheritance and polymorphism," Proceedings of the 12th International Symposium on Software Reliability Engineering, pp. 84-93, ISSRE '01, IEEE Computer Society, Washington, DC, 2001.

[13] Jonassen, David H., "Toward a Design Theory of Problem Solving," Educational Technology Research and Development, vol. 48, no. 4, pp. 63-85, 2000.