산업기술연구(강원대학교 산업기술연구소 논문집), 제26권 B호, 2006.
Journal of Industrial Technology, Kangwon Natl. Univ., Korea, No. 26 B, 2006.

Visualizer: An Interactive Execution Tool for Java Programs

Kim, Hyung-Rok* Kim, Yong-Seok**

Abstract

Effective use of debugging in computer science education is a topic that has received little attention. Existing tools either are too overwhelming for the novice or are too basic and inflexible. Furthermore, the integration of the debugger into complex IDEs have unfamiliar debuggers from students, resulting in a harmful proliferation of console I/O. In this paper, a new stand-alone educational debugging tool is presented, with simple but powerful tools for easily observing and modifying program execution state. This tool is analyzed from an user perspective, and is compared with existing tools.

Keywords : *Interactive Execution, Java, Debugger, IDE*

1. Introduction

Traditional command-line debuggers, like `gdb` and `jdb`, have been available for a very long time since Unix and the C programming language became dominant. However, these programs, often considered complicated and arcane, are unfamiliar and difficult to use for most computer users today, who are accustomed to graphic user interfaces (GUIs)[1,2].

Also, debuggers within integrated development environments (IDEs) such as NetBeans, Eclipse, and Visual Studio, although widely used, are typically designed for professional programmers and are not emphasized by students and instructors, despite the fact that actually examining the actual program flow is an integral part of programming[3,4,5].

To address this problem, there has been attempts at educational, alternative IDEs suitable for student use, such as BlueJ and DrJava, but none are targetted specifically at simplifying debugging for students. These are enumerated and examined in detail in the Related Works section[6,7,11,12].

Faced with the lack of such programs, many novice programmers resort to the use of

* Korea Science Academy, Student
** Dept. Computer Science and Engineering, Kangwon National University, Professor

primitive console I/O (such as `printf` in C, `std::cout` in C++, or `System.out` in Java). Such approaches are bad for several reasons:

- **Performance**: Redundant logging can slow down performance considerably, as I/O is very slow compared to other operations a student program might perform.
- **Difficulty of use**: In order to print logging output, the quantities logged must be converted to string form. Although in Java all classes have a `toString` method, it still requires the implementation of `toString` for every class, which is annoying.
- **Difficulty of removal**: When polishing a program for final release, all the logging statements must be removed manually. Although there are idioms (such as `#define NDEBUG`) and libraries (such as `java.util.logging`) that ameliorate these problems, such frameworks are often difficult to use.
- **Ineffectiveness**: Haphazard, random, inconsistent logging is not very helpful; the student is confused by other logging statements.
- **Bad habits**: Habitual use of console I/O is unacceptable in professional GUI programming. Abusing console I/O at such early stages encourages formation of bad, hard-to-break habits.

Thus, in order to fill this void, we have developed a tool named *Visualizer* that introduces and explains program execution to beginners and eases debugging. The architecture and implementation of this program will be detailed in the following sections.

2. Implementation

2.1 Which language?

There are many widely-used programming languages in the computing world, each with different linking models and means of execution, and it is not feasible to create one tool that will handle all of these languages.

In rankings of programming language popularity (such as the well-known one from TIOBE Software)[8], the Java programming language from Sun Microsystems, Inc., has consistently held the top position for over a year, followed by other languages like C and Visual Basic. Also, Java is widely used in introductory programming courses. For instance, College Board recently switched to Java from C++ for their Computer Science exams. This widespread usage of Java makes it an attractive candidate.

Also, Java's dynamic nature, unlike more static languages such as C or C++, enable more flexible and intuitive interface. Of note are Java's extensive introspection facilities, defined in the core library in `java.lang.reflect`, and JPDA (Java Platform Debugging Architecture)[9], Sun's API for manipulating the Java Virtual Machine itself, also in the standard library; these enable direct observation and monitoring of program state through well-defined APIs.

Thus, Visualizer supports programs written in the Java programming language, and is itself written in Java to utilize the rich Java APIs mentioned above.

2.2 Displaying execution state

Traditional IDEs, as noted, have overly complex user interfaces that alienate neophytes. For instance, Fig. 1 shows the integrated debugger of a widely used Java IDE, NetBeans. Note the presence of various other functionality, including extraneous buttons, toolbars, menus, and tabs. Although this might provide useful functionality to professional, seasoned programmers, novices are often overwhelmed by this complexity. A useful UI design should cut back and simplify.

In comparison, the Visualizer window, shown in Fig. 2, is completely different. Most of features unrelated to the observation of program execution, including editing, have been cut out. Also, the relevant displays, such as the list of classes and variables, have been clearly labeled, aiding comprehension. Such a minimal interface might feel too drastic, but the simplicity is ultimately beneficial, as it clarifies the interface.

The interface consists primarily of five displays:

- *The source display*, located in the left half of the window, which displays the source code and highlights the line to be executed.

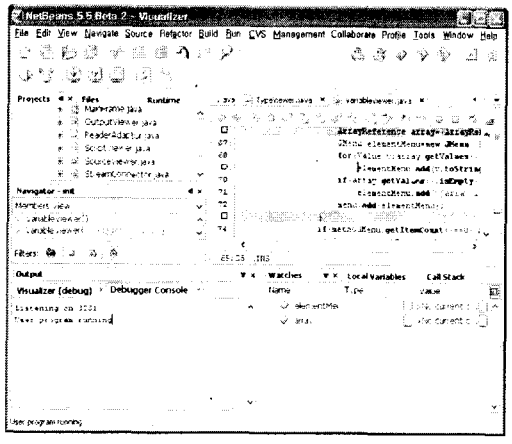

Fig. 1 A typical debugging session within NetBeans 5.5

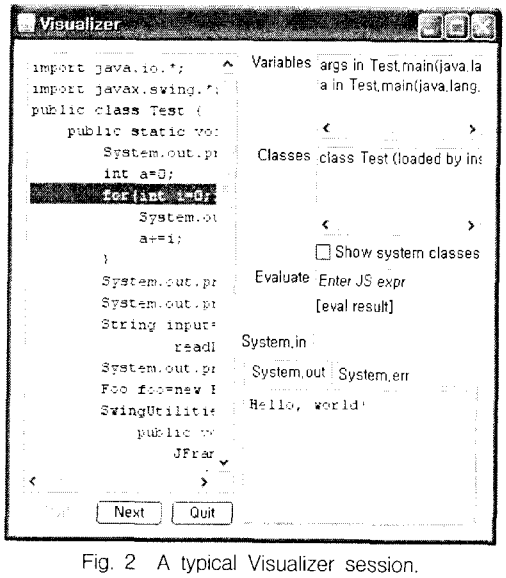

Fig. 2 A typical Visualizer session.

- *The variable display*, located in the top right corner of the window, which lists the accessible variables in the current stack frame, including local variables and method arguments.
- *The class display*, located in the middle right section of the window, which lists the currently loaded classes. An option is available to filter out array classes, classes from the standard library, and classes that are specific to the virtual machine and show only custom classes.
- *The I/O display*, located in the lower right section of the window, which represents `System.in`, `System.out`, and `System.err`, he three standard streams of Java. These correspond to `stdin`, `stdout`, and `stderr` in C, respectively.
- *The scripting display*, located between the I/O display and the class display. This will be explained further in the next section.

The class and variable displays have context menus that list the available fields and methods, as shown in Fig. 3. For arrays, this shows the elements of the array; for primitive values (such as int and char) and strings, the value is shown directly.

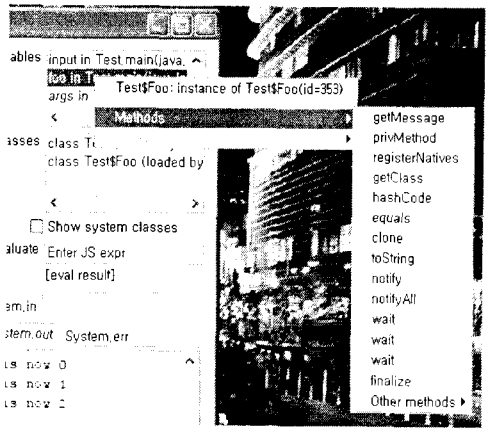

Fig. 3 The context menu in class and variable displays.

2.3 Interactive evaluation

An interesting feature is the support of interactive evaluation of expressions during execution. This feature utilizes the new scripting integration in the upcoming Java SE 6 (informally known as Mustang), specified by Java Specification Request (JSR) 223[10]. Short code snippets of scripts may be entered in the scripting display, as shown in Fig. 4, and a scripting engine dynamically evaluates the snippet, within the stack frame of the currently executing Java program, and sharing the variable name space. Using scripting languages, such as ECMAScript (i.e. Javascript), Python, or Ruby, one can dynamically observe program state, evaluate expressions, and change program behavior.

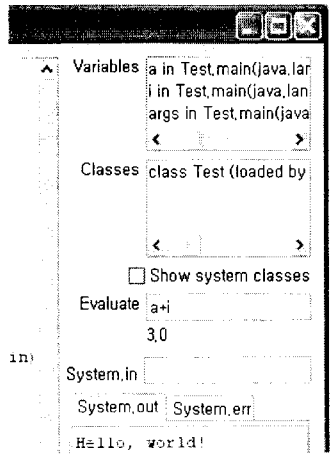

Fig. 4 The scripting display.

A scripting language, with convenient syntactic shortcuts and dynamic typing, offer convenience in writing little, throw-away snippets than more restrictive, statically typed languages. Also, it was felt that the existing syntaxes of Java-like scripting languages, such as ECMAScript, Groovy, and BeanShell are similar enough to Java that students would have no trouble adapting. Indeed, it would not be obvious to most students without prior exposure to such languages that a different language was being used. In addition, evaluation of Java expressions require on-the-fly compilation and other technical difficulties. Nevertheless, it should be noted that DrJava, further discussed in the Related Works section, takes this approach.

3. Applications

Visualizer can be used for educational purpose. Some of benefits are illustrated below.

Resolving the cause of bugs:

This is the classic debugger use case. A debugger's benefits in debugging is widely known, and will not be elaborated here.

Exploring and understanding an unfamiliar program:

Another common, but often neglected use of a debugger is in understanding how an unfamiliar piece of code operates. By following the sequence of method calls, a programmer can often have a better understanding of the internal implementation of a program, especially novices who often have trouble reading difficult source code and who are without the aid of a professional IDE.

Dynamic, run-time testing:

Still another use made possible by the scripting features of Visualizer is to modify variables and other parameters at run time, and see the resulting execution. Often, manually changing the value of a variable and recompiling is tedious and error-prone, and building a GUI interface for such tasks is even more difficult. By making such changes at run time, such manual activities may be reduced.

Learning control flow:

By observing the flow of control as the currently executing line highlighted by Visualizer moves around, a student can get a better understanding of control structures and method invocation.

Understanding the internals of the run-time environment:

Finally, Visualizer can be used to gain a better understanding of the Java Runtime Environment (JRE) and computers in general. A typical novice's mental model of the JRE is that of a black box that somehow executes the fed program magically. Although such abstraction and simplification might be argued to be actually beneficial, such might give false impressions to the beginner. For instance, a common misconception of neophyte Java programmers with a C or C++ background is that the methods in a Java classes, like C functions, are statically linked. This is clearly not the case in Java; a glance at the class display would show that class files are loaded by the JVM on-the-fly as needed. Also, one can learn that many bootstrap classes, including the classes in APIs internal to the JVM, are invoked before the main method is run.

4. Related Works

A number of development tools dedicated to educational use have been developed. Here, two such tools, namely BlueJ and DrJava, are presented and compared to Visualizer.

Table 1 Comparison of debuggers in educational use

	Visualizer	BlueJ	DrJava
Interactive Evaluation	Yes	No	Partial[1)]
Simple Interface	Yes	Yes	No

1) Not integrated with the debugger

4.1 BlueJ

BlueJ, described in detail in the paper [6], is a JavaIDE suitable for educational use, with class diagrams and interactive object manipulation. It emphasizes good design practices, such as unit testing (with JUnit) and documentation, and has an integrated debugger as well.

Its integrated debugging tool is simple, light, and well-integrated, as shown in Fig. 5. It offers basic debugging functionalities, such as break points, stepping, and halting, with an elementary interface suitable for beginners' use. However, it neither displays classes, nor does it enable one to interactively observe and modify a program during execution, like Visualizer.

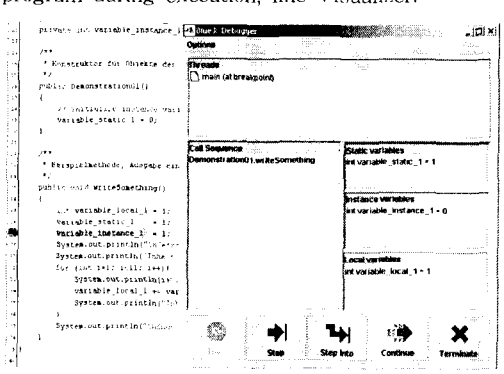

Fig. 5 The integrated debugger in BlueJ.

4.2 DrJava

DrJava, described in detail in the paper [7], is another Java IDE aimed at neophytes. It is modeled after the famed DrScheme environment[13] for Scheme.

Its primary feature of interest is its DrScheme-like interactive evaluation of Java commands, as shown in Fig. 6, a very convenient feature at times. However, this dynamic evaluation is not coupled with execution; that is, this feature cannot be used to modify a program while it is running, unlike Visualizer.

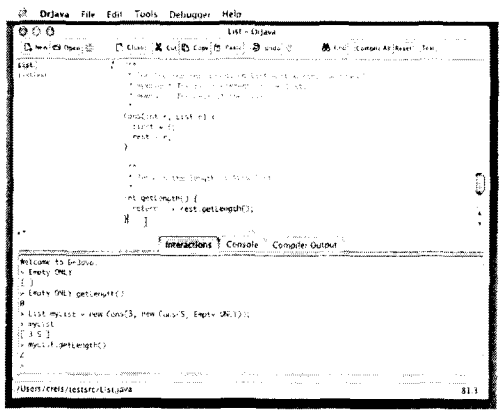

Fig. 6 Interactive evaluation in DrJava.

DrJava also has integrated debugging support, a scan be seen in Fig. 7, which is slightly more advanced and more traditional than that of BlueJ, with watches, call stack, and other advanced features. However, this debugger is also limited to standard features found in other debuggers, with little dynamic features.

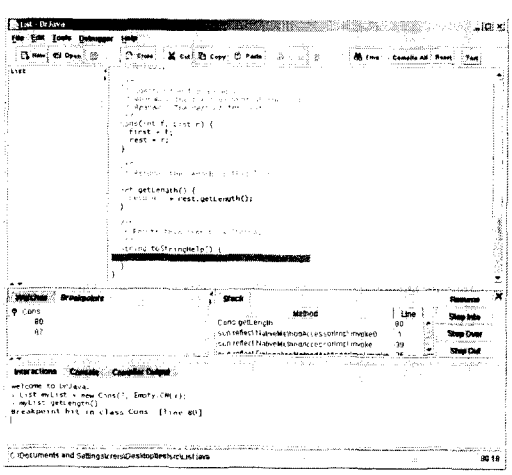

Fig. 7 Integrated debugger in DrJava

5. Conclusions and Future Works

Current debugging tools are either too complicated and abstruse or too basic to be used in computer science education today. Visualizer is an attempt to rectify this situation, and has already made much progress. The dynamic execution of scripting languages is stable and working well.

However, Visualizer is still rather primitive in its presentation of program elements: essentially as plain text. Diagrams and pictures would be a more comprehensible and visually pleasing alternative. This is currently being addressed.

References

[1] gdb (GNU Project Debugger). http:// gnu.org/ software/gdb.

[2] jdb (The Java Debugger). http:// java.sun.com/products/jpda/jdb.html.

[3] NetBeans IDE. http://netbeans.org/ ide.

[4] Eclipse JDT (Java Development Tools). http://eclipse.org/jdt.

[5] Microsoft Visual Studio. http://micro soft.com/vstudio.

[6] Michael Köling, 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)*, 13(4), pp. 249-268, December 2003.

[7] Eric Allen, Robert Cartwright, and Brian Stoler. DrJava: A lightweight pedagogic environment for Java. In Deborah Knox, editor, *The 33rd ACM Technical Symposium on Computer Science Education*. ACM SIGCSE, 2002.

[8] TIOBE Software BV. Tiobe programming community index. http://www.tiobe. com/tpci.html.

[9] Java Platform Debugger Architecture. http://java.sun.com/products/jpda.

[10] JSR 223: Scripting for the Java Platform. http://jcp.org/en/jsr/detail?id=223.

[11] BlueJ. http://bluej.org.

[12] DrJava. http://drjava.org.

[13] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, pp. 159-182, 2002.