Memview: A Pedagogically-Motivated Visual Debugger

Paul Gries, Volodymyr Mnih, Jonathan Taylor, Greg Wilson, Lee Zamparo
University of Toronto, Department of Computer Science
Toronto, Ontario M5S 3G4

Abstract **- Novice programmers often have difficulty understanding the interactions between the objects in their programs. Many studies have shown that visual representations of computer memory can aid students comprehension. One such representation, developed by Gries and Gries, divides computer memory into three areas: one for the call stack, one for static objects allocated on the heap ("static space"), and one for normal heap objects ("object space"). Memview, an extension to the DrJava IDE developed at Rice University, is a dynamic, interactive display of computer memory based on this model. Its simple three-pane representation shows novices the life cycle of objects, and helps them understand three key concepts: the notion of an "address" in memory, how storing an address creates a reference from one object to another, and the differences between the heap, the stack, and static space. User tests conducted during the summer of 2004 demonstrated that Memview facilitated faster completion of common introductory programming problems. Since then, Memview has been used in introductory programming courses to illustrate basic data structures such as linked lists. We are presently refining the tool based on further feedback from students and instructors.**

Index Terms - CS1, CS2, Debugging Aids, Java, Program Visualization

Introduction

Many CS1 and CS2 students have difficulty with the following concepts, in roughly this order:

- the difference between a variable and an object;
- the difference between primitive and reference variables;
- how local, static, and instance information interacts;
- how method calls work;
- inheritance and polymorphism;
- recursion.

Several visual representations of computer memory have recently been proposed in order to help students understand these concepts [4], [5]. The representation developed by Gries and Gries divides computer memory into three areas, drawn as panes: one for the call stack, one for static objects allocated on the heap ("static space"), and one for normal heap objects ("object space").

In exercises, students were required to trace the execution of simple programs, drawing the memory model after specific events. Experience shows that doing so helps them master the concepts listed in the bulleted list more quickly [5]. However, students often complain about these exercises, as drawing (and re-drawing) diagrams is tedious.

Having adopted DrJava [1] (a lightweight pedagogic IDE) for first-year teaching, Gries began a project to add a dynamic, interactive representation of the memory model to it. The work was done by three senior undergraduate students (Mnih, Taylor, and Zamparo) as a course project, for which Wilson acted as project manager. The result is now fully functional, and was used successfully in a CS1 class during the 2004 fall semester. In addition, the project received the attention of a new team of undergraduates who further developed the code base during this time leading to a more polished product.

Related Work

Many projects have produced tools to aid beginning programmers in visualizing or understanding programs. CMeRun augments C++ code with output statements that print each of the executable statements in the unaugmented code along with current values of all variables of primitive type referenced in the statement [3]. LIVE automatically creates data structure visualizations by parsing source code, and includes support for multiple languages, including Java and C++ [2]. Tango [7] is one of the many existing frameworks for building algorithm animations with the help of code augmentation.

After evaluating the above systems we did not find any of them to be suitable for our needs, mainly because they do not place enough emphasis on object-oriented concepts. Giving students the ability to see a visual representation of computer memory will leave them with a better understanding of the usually problematic concepts we presented in the introduction.

Memory Model

0-7803-9077-6/05/$20.00 © 2005 IEEE

The memory model represents computer memory as three areas, drawn as panes: one for the call stack, one for static objects allocated on the heap ("static space"), and one for normal heap objects ("object space").

Consider the following code, which counts the number of times the letter 'a' appears in the word "abracadabra"; the line numbers for each file appear on the right. We use this code to illustrate the memory model.

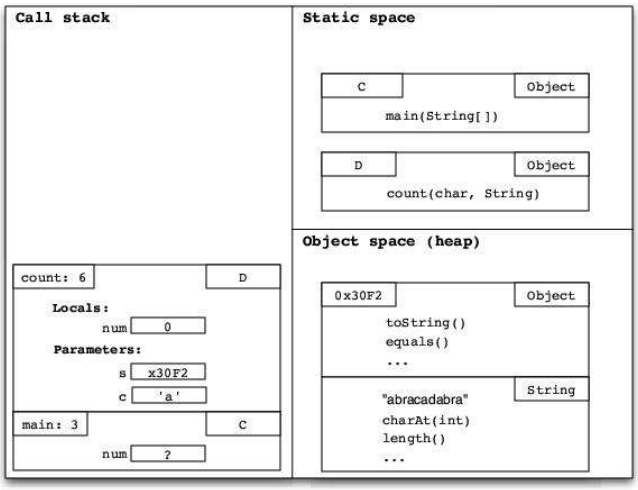

FIGURE 1
THE MEMORY MODEL AT LINE 6 OF METHOD D.COUNT

```
public class C {                                          1
  public static void main(String[] args) {               2
    int num = D.count('a', "abracadabra");                3
  }                                                       4
}                                                         5

public class D {                                          1
  /** Return how many times c occurs in s. */            2
  public static int count(char c, String s) {             3
    int num = 0;                                          4
    for (int i = 0; i != s.length(); i++) {               5
      if (s.charAt(i) == c) {                             6
        num++;                                            7
      }                                                   8
    }                                                     9
                                                         10
    return num;                                          11
  }                                                      12
}                                                        13
```

Figure 1 shows the state of memory at line 6 in D.java, just inside the first iteration of the for loop, as if a breakpoint had been reached in a debugger. The left pane contains the call stack, the upper right pane the static space, and the bottom right the object space.

Call Stack

The call stack contains a stack of boxes. The function main is on the bottom because it was called first. As the program runs each line of code is executed, and at line 3 of C.java method count is called, which places the box for count on the call stack. Again, Figure 1 shows that the count method is paused on line 6.

Each box in the call stack is labeled with the name of the method and the line number that is being executed in that method. The contents of the boxes include parameters and local variables, drawn as flat boxes labeled with the variable names. (We do not show main's String[] parameter.)

Objects

There is a single object shown in the object space: the String representing "abracadabra". x30F2 is a unique, made-up hexadecimal number, which we call the "memory address" of the object. There are two parts to this object: instance information inherited from class Object, and instance information declared in the String class.

Static Information

Static variables and static methods are drawn in the static space. Each class is represented by a box containing the static information. Boxes exist for C and D. They each have only one method; if any static variables were declared they would appear inside the appropriate box. The upper-right subbox of each indicates the superclass; in this case both C and D derive from Object.

Name and Scope

Each memory box (method call, object, static box) looks roughly the same: the upper-left subbox indicates the "name" of the box, and the upper-right subbox indicates the scope of that box. The scope box indicates where to look for more information.

DRJAVA

DrJava is a lightweight, yet powerful, integrated development environment for Java developed at Rice University. Its simple interface and syntax-aware editor allow students to concentrate on developing their programming skills instead of struggling with an environment. DrJava includes a built-in debugger, and an interactions pane, which allows students to type Java expressions and statements and see them immediately evaluated in a read-eval-print loop [6].

CS1 classes at the University of Toronto use DrJava in two ways: to help beginning programmers focus on concepts, rather than mechanics, and to demonstrate new concepts live during lecture. Students find that seeing new Java features "in action" as soon as they are introduced aids comprehension and speeds learning. In particular, the DrJava debugger and interactions pane are invaluable when trying to clarify ideas for confused students.

DrJava is open source software, available under a very liberal license. Thanks to the generosity of the Rice University team in making its full source available, we were able to implement Memview as an extension of DrJava's built-in debugger.

MEMVIEW

Memview's primary goal is to aid in the average CS1/CS2 students understanding of objects and their interactions. We start by explaining the design decisions made that help Memview achieve this goal. In particular, the choice was made to provide Memview as an extension to the regular DrJava debugger. We then describe how this choice gives Memview a robust architecture resistant to upstream developments. The results of preliminary user testing are presented with an analysis of the problems these tests revealed. These tests were conducted with individual CS1 and CS2 students and provided us with a chance to polish Memview before taking it into the classroom. The performance of Memview in the classroom is the true test for its success so in the final subsection we discuss our experiences with taking Memview into a real CS1 class.

Design

In order to leverage the existing debugging functionality of DrJava, we decided to implement Memview as an extension that can coexist with the standard DrJava debugger. This means that anyone who has used DrJava in the past will be able to debug with Memview using the familiar interface for setting breakpoints and stepping through code.

Like the standard debugger, Memview is presented as a pane within the main window. This pane can be brought up through the Debug menu at any time during a debugging session. Since Memview uses the memory model we described in an earlier section as its visualization format, the Memview pane is divided into three sections - one for each part of the memory model.

Still following our memory model, the call stack is visualized by a series of boxes. Each box represents a method on the call stack. As shown in Figure 3, method boxes used by Memview closely resemble the ones pictured in Figure 1. A tree component is used to display parameters and local variables in separate folders. This allows users to view only information that is relevant to them.

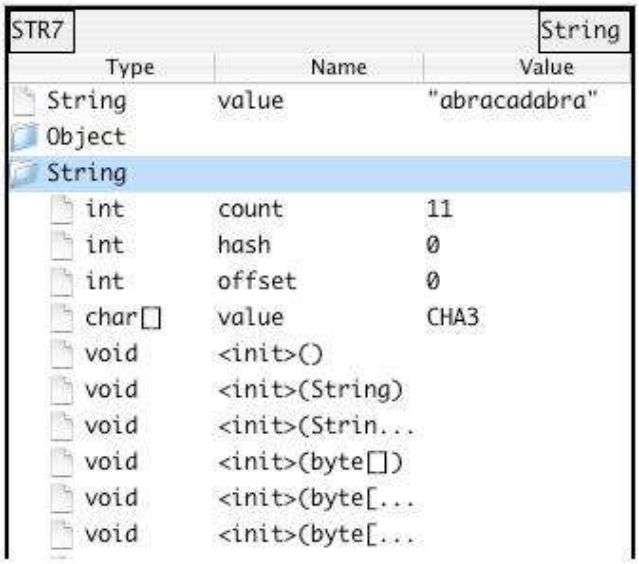

FIGURE 2
SCREENSHOT OF A STRING INSIDE MEMVIEW

Figure 2 shows a partial screenshot of a String object on the heap with the String part expanded and the Object part collapsed. At the top is what the object represents, "abracadabra".

We represent a process' heap by populating a panel with boxes representing objects on a heap. Memview uses an object discovery algorithm that recurses through the objects on the stack in an acyclic manner. The objects are placed from left to right along a three column grid as they are discovered. The user level customization of this grid is a future goal for the project.

Objects in the JVM's (Java Virtual Machine) static space are represented by boxes in a left to right fashion in a static space panel. These objects show the static object's fields and methods. Unfortunately, some of the different flavours of JVMs provide these fields and methods only after they are used. Devising a method to force all JVMs to perform consistently in this manner is a high priority for the Memview team.

Architecture

Users are able to manipulate the JVM by setting breakpoints and stepping in/over instructions via DrJava's debugger as usual. When the program being debugged is suspended, DrJava notifies Memview, which updates its display. This is accomplished by querying the JVM for information concerning the target program through the standard Java Platform Debugger Architecture (JPDA) interfaces [8].

At present, the only interaction between Memview and the standard debugger is that Memview listens (via JPDA) for breakpoint occurrences, which have been set through the standard debugger. At this point, Memview refreshes itself with information from the JVM. We chose to keep the two separate in order to minimize development time, and to avoid

complicating DrJava's interface any more than necessary. This separation will also shield us from complications that might arise from the overhaul of DrJava's debugger planned for later in 2005.

User Testing

We evaluated Memview's usefulness by conducting usability studies with several students and instructors. Test subjects were presented with Java code for a simple linked list class. We then used this class to create a short linked list, and asked our test subjects to use Memview to find the value stored at the tail item of the linked list without looking at the code that had originally generated it.

Before the study began, the instructors who took part believed that the memory model was useful, but that drawing memory model diagrams in class was tedious. After using Memview, the instructors agreed that using it interactively to generate such diagrams would be faster, easier, and also more compelling, since students would see the diagram evolve in step with the debugger's traversal of example programs.

The user tests did reveal some problems with the current implementation of Memview. Perhaps the most serious is the system's lack of scalability. While Memview was able to handle programs from CS1 assignments from past years, the visualizations became increasingly cluttered for programs representative of the larger CS2 assignments. Since these tests were conducted, the code has been patched by the fall 2004 team. Object boxes may now collapse by double clicking on the appropriate box. This partially addresses the shortcoming but new ways of limiting the amount of information that is displayed are still being considered.

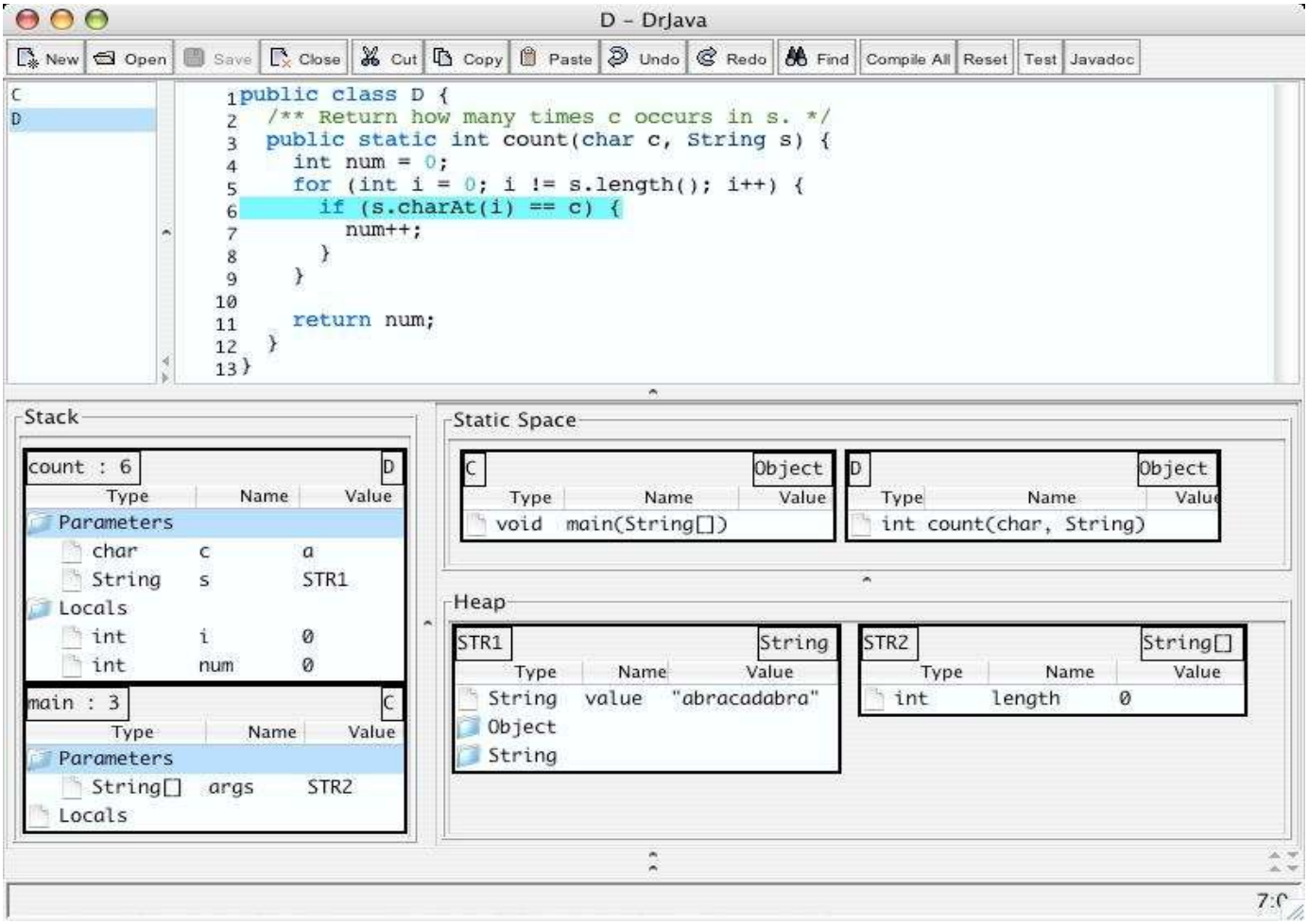

FIGURE 3
DEBUGGING A PROGRAM WITH MEMVIEW

Second, Memview has no knowledge of the Java source files being worked on, but instead retrieves data directly from the JVM of the target program. This means that Memview cannot differentiate between objects created directly by users, objects created by DrJava, and objects internal to the the JVM. These last two categories generally account for over two hundred objects, which, if visualized, would saturate the interface with useless information. By implementing namespace filters (e.g. do not display objects from packages edu.rice.cs.drjava.*), we were able to restrict our display to the subset of objects most likely to have been created by the user. In the future, this static system will be replaced by a

customizable filter set which will both solve this problem and allow instructors to zoom in on sets of objects deemed most important.

Finally, at the time of these tests, Memview did not preserve context between debugging breaks. When the next breakpoint is reached, Memview would rebuild the visual display without regard to what was being visualized in the previous step, effectively collapsing all expanded folders in every object. Users found this tiresome, since they would have to re-expand all the same tree tables just to see the same information every subsequent break. The fall 2004 team implemented a caching mechanism that stores information regarding object state across breakpoints. When the display is rebuilt, this cache is queried and if an object has already been displayed, it is restored to its previous visual state.

Overall, we found the user tests very encouraging, and signified that Memview was ready to be used in the classroom. We were particularly pleased by the interest of instructors in using Memview to teach CS1. In particular, Memview was used in the fall 2004 CS1 class. This is discussed in the next subsection. Feedback gathered from these tests determined the most promising directions for the future development of Memview. The fall 2004 capitalized on the information we gained from these tests in order to prioritize various features most useful to the target audience. With this newer polished version of Memview, the responses from such user tests would only be more positive.

Classroom Experience

In the fall of 2004, Gries taught the CS1 course. Memview was used to both demonstrate the use of objects within the classroom and to help students understand concepts during office hours.

Objects are introduced early in CS1 at the University of Toronto, in the second lecture hour. Students are shown a model of memory that is equivalent to the one in Memview. This has traditionally been done on the blackboard alternating with showing the same code in a debugger. Especially for students who had programming experience from high school, the model of memory can be a shock: each term several students argued that thinking about call stacks, objects, how and where variables are stored, and so on was not helpful. (Some even argued that it was inaccurate, although after much argument and many examples they always recanted.)

Because switching from the blackboard to a projector involves changing the lighting and raising or lowering the screen, teaching using Memview was much more natural: the debugger is the picture of memory. The presentation went more quickly, with no pauses while waiting for the screen to raise or lower. More interestingly, for the first term ever no student argued that learning a model of memory was not helpful.

During office hours, weaker students ask questions demonstrating that they do not understand fundamental concepts such as how a method call works, or how references to objects work. Memview has invariably proved helpful, not only because it demonstrates what is happening in memory but also because it provides a focus for discussion.

Availability

Memview is currently residing on the server reserved for Professor Wilson's supervised course projects, at http://pyre.third-bit.com. It is available under the same open source license as DrJava itself. Currently, the only access is through CVS, as up until recently there has been very active development. There are, however, CVS tags indicating useable milestones. In the future we would like to provide a 1.0 release at which time we will concentrate efforts to integrate Memview into the official DrJava codebase. Inquiries about Memview should be directed to Professor Wilson (gvwilson@cs.toronto.edu).

Conclusions and Future Work

Memview is now functional and has been through two rounds of development. Following the positive initial tests, Memview was given a trial run, being used in a CS1 course at the University of Toronto in the Fall 2004 term. As reference din the classroom experience section, the success of that trial demonstrates that Memview can be a useful tool in the classroom, and affirms that Memview has achieved its original goals.

During the Fall 2004 term, the development of Memview was continued by a fresh team of undergraduates under the direction of Gries and Wilson. The new team was able to:

- update the Memview code to be in line with the official DrJava codebase
- address scalability and robustness issues through the use of caching
- improve Memview's running speed
- allow objects to minimize when double clicked
- achieve persistent customizations to the user interface (e.g. expanded folders) across breakpoints
- polish the user interface and remove bugs

Although our tests have shown that Memview, in its current state, is ready for use in the classroom, these features will make Memview a more attractive choice for instructors looking for an in class teaching tool. As well, first year students struggling to understand the basics of Java's memory management will find it useful to use Memview on their own to walk through example code from textbooks and assignments.

Acknowledgment

We wish to thank the JavaPLT group at Rice University for creating DrJava, and for making it open source. We would especially like to thank Charlie Reis, a former JavaPLT

member now at the University of Washington, for explaining some of the trickier features of DrJava to us, and for providing so much helpful feedback on our designs. As well, we give our gratitude to the instructors and students who took part in our tests and thank them for their valuable feedback. Lastly, we thank the fall 2004 team consisting of **Adrian Horodeckyj, Ryan Liang and Qian Zhu** who continued the development of Memview.

References

[1] Allen, E. and Cartwright, R. and Stoler, B., "DrJava: a lightweight pedagogic environment for Java", Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education, 2002, pp 137-141.

[2] Campbell, A. E. and Catto, G. L. and Hansen, E. E., "Language-independent interactive data visualization", Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, 2003, pp 215-219.

[3] Etheredge, J., "CMeRun: Program Logic Debugging Courseware for CS1/CS2 students", Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, 2004, pp 22-25.

[4] Gries, P. and Gries, D., "Frames and Folders: A Teachable Memory Model for Java", Journal of Computing Sciences in Colleges, Vol 17, No 6., 2002, pp 182-196.

[5] Holliday, M. A. and Luginbuhl, D., "CS1 Assessment using Memory Diagrams", Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, 2004, pp 200-204.

[6] Sandewall, E., "Programming in an Interactive Environment: The 'Lisp' Experience", Computing Surveys, Vol 10, No 1., 1978, pp 35-71.

[7] Stasko, J. T., "A Framework and System for Algorithm Animation". IEEE Computer, Vol 23, No 9., 1990, pp 27-39.

[8] Sun Microsystems. "The Java Platform Debugger Architecture", http://java.sun.com/products/jpda/doc/ , 2003.