Program state visualization tool for teaching CS1

Otto Seppälä
Helsinki University of Technology
Department of Computer Science and Engineering
Helsinki, Finland

oseppala@cs.hut.fi

1 Introduction

To understand a natural language one must first understand the vocabulary, the meaning of each word. When the vocabulary is combined with grammar, it becomes possible to explain more complex concepts. However, this is usually not enough. To understand the whole meaning of any piece of text or speech, it is crucial to know the frame of reference. The same words carry a different message at different times and in different places. The same applies to computer programs, but with a fundamental difference: the frame of reference is different for the same line of code each time that single piece of code is executed. The programmer thus has to understand how the runtime state of the program changes when the program is executed.

In imperative or procedural programming this frame of reference is easier to understand, as using only local and global variables is equal to object-oriented programming within a single object. In object-oriented programming we have multiple objects and thus the visible variables can change as a result of each method call. Before the learner fully adopts the object-oriented way of thinking, following the program state can require a lot of attention. When we conducted a questionnaire about programming habits on our main programming course, 43 percent of the students claimed that they spent most of their time trying to make their programs conform to exercise specifications or trying to fix runtime errors. This implies that understanding of the dynamic state of the program should be given more attention. This view is supported by Mulholland and Eisenstadt (1998), who found that in most of the cases the best way to help the students with their programming problems was just to show them how their programs actually worked. This is essentially something that can be done with a debugger.

Holland et al. (1997) propose using counterexamples to avoid object misconceptions. We believe that such examples will be more powerful using a program state visualization tool to support the process.

2 Background

Using a debugger for lecture demonstrations or as a supportive tool when making exercises has been found to be an effective way of explaining how a program executes. Cross et al. (2002) used the *jGRASP* debugger as an integral part of their CS1 course and reported being surprised by the positive response from the students. Their hypothesis is that the added complexity in object–oriented programs compared to procedural programming impedes the students' understanding. This complexity is lowered by using a debugger to step through the programs, revealing layer by layer how the program state changes.

The debugger was used on lectures to explain programs. It was reported that using the debugger both motivated students and made lecturing easier. jGRASP has all the functionality of a typical debugger and also creates control structure diagrams to help in program understanding. Cross et al. (2002) conclude that any modern debugger could be used on lectures to better explain programming concepts.

Different debuggers and learning environments have their specific strong and weak points. The educational debugger/animator, *Jeliot 2000* (Levy et al., 2001), is well suited to teach the basics of procedural programming, but rejects Java programs that use any object-oriented features. With the growing tendency to teach object-oriented programming objects-first, this is a bit troublesome.

Kölling and Rosenberg (2002, 2001) describe how the *BlueJ* environment can be used to effectively teach object-oriented programming. BlueJ has been designed to lower the introductory level to programming by eliminating the use of many relatively complex language constructs that have to be written to try out a single method. BlueJ also draws an UML-like diagram of the class hierarchy of the program.

One crucial problem with BlueJ is that it does not display the references between objects. Another is that it assigns names to objects. This easily leads to misunderstanding, as students are known to conflate variable names with object identities (Holland et al., 1997). We believe that a diagrammatic notation without any artificial naming is less likely to create such misconceptions.

Such diagrams are created at runtime by the free *Data Display Debugger (DDD)* (Zeller and Lütkehaus, 1996) which makes beautiful visualizations of references between data structures. It also has a huge amount of different visualizations for displaying how data evolves during the program. DDD can be used for lecture demonstrations and student use the same way as Cross et al. (2002) used jGRASP. We feel that the diagrams explaining the references between the data structures would make it even more useful in explaining the concept of reference variables and showing how the state of each instance evolves during program execution. DDD and jGRASP use a separate window for displaying the execution stack. In a procedural programming environment, this is usually enough, but in object-oriented programming the stack is bound to have a lot of references to objects. A special case is the *this*-object on which the method call was made. The *VizBug* system (Jerding and Stasko, 1994) uses arrows to represent method calls. These arrows emanate from and end on visual elements representing both instances and global (static) methods. Doing so, the arrows essentially reveal all the "this"-objects currently in the stack. A similar notation, the collaboration diagram, which can be used to describe the order of method calls between instances is found in UML (The Object Management Group, 2003).

Vizbug does not visualize references between objects. However, the system visualizes the inheritance hierarchy in the same diagram. This was done to better display ideas of polymorphism and inheritance when methods and constructors are invoked.

One system still worth mentioning is the JAVAVIS (Oechsle and Schmitt, 2002). JAVAVIS visualizes the program state by showing in separate windows the contents of each of the stack frames currently in the stack. This information is shown with an object-diagram-like notation, essentially revealing which information could be referenced through the objects in the stack. JAVAVIS also uses sequence diagrams to better show the control flow during the execution.

As told before, our goal was to find a system that would visualize both the method invocations and references simultaneously in the same graph. While many systems came close, such a tool was not be found. The author therefore constructed a program visualizer/debugger for testing this concept in computer science education. This debugger uses a diagram notation we call the *program state diagram*. This notation combines some features found in visualizations constructed by the VizBug and DDD debuggers. Our visualization closely resembles UML, but with intentional differences where we do not want the learner to confuse the two.

The interface of the visualization tool is intentionally simple, containing only the basic stepping and running operations for traversing the program execution. There are two views to the debugged program, one showing the diagram and the other displaying the source code.

3 Program State Diagrams

Combining features from both the object diagram and collaboration diagram, we come up with a new notational scheme: the program state diagram. This notation attempts to show most of the runtime state of the program in a single diagram. Essentially, this means displaying all relevant instances, all references to them and some of the contents of the runtime stack together.

The diagram displays instances as white rounded rectangles and references between them as yellow arrows. The name of the referring variable is shown next to the arrow.

Method calls, their parameters and local variables are shown in the same diagram using a visualization style that sets them apart from the instances and references. In our approach, the method calls are depicted in the same diagram using red curved arrows to better separate them from the reference arrows. Arrows have been used for visualizing method invocations for example in VizBug and in the UML collaboration diagram.

Static and local reference variables are also visualized. As these cannot be seen to belong with any specific instance, they are shown with separate ellipsoids. Local references have a blue background and static references purple. The objects these variables reference are shown with reference arrows in the same style as used to show references between instances. Static method calls use a notation similar to that used in VizBug. A static method is shown as a separate rounded rectangle with a green background color to distinguish it from instances. All of these code elements (with the exception of a static variable) are used in the example given in the next section and thus the corresponding visual elements are present in Figures 1 and 2.

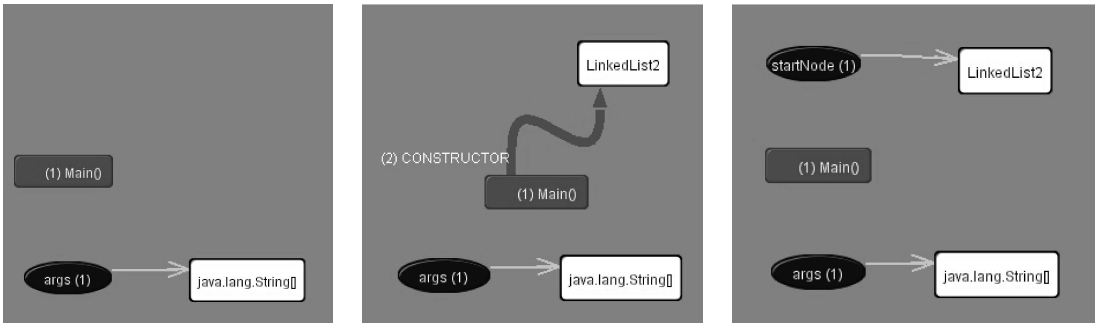
4 Usage

Recursion and reference variables are the source of many elementary misconceptions, because they are somewhat hard to explain verbally. Visualizing recursion is quite easy using the program state diagram. Three diagrams created by the debugger are shown in Figure 1. We will now explain how to interpret the notation by studying these figures and the example program they were generated from. The example program we are debugging will construct a doubly linked list with four nodes. The program code is shown in Tables 1 and 2.

As with any Java application, on the bottom of the runtime stack lies the method *main*. In our example this method creates the linked list by adding new nodes one by one and linking them to the end of the list using a recursive method call. In the leftmost image, we see the beginning state with only the *main*–method and its command line parameter *args*. The middle image shows the *main*–method invoking the *LinkedList2* constructor. Constructor calls are visualized with red arrows, similar to method calls. On the right we see the program state just before the loop which adds the next three nodes. This image shows another example of a reference by a local variable.

We now skip a number of steps to reach Figure 2, which shows the program state when adding the last node to the linked list is nearly finished. The recursion progressing through the list is clearly visible, as are the parameters given to the method invocations. The new node is given as parameter *link* for all the other calls but the last. The latest method call made was *setPrevious*, invoked on the *LinkedList* instance on the far left. The parameters and local variables for this call can be found by reading the stack height, which is shown both next to method arrows and local variable names. After this method call ends, its arrow retracts to its starting position. In the image we can now also see references between the instances that create the linked list from the separate nodes.

The debugger also has the typical prettyprinted code window where the currently executing location in the code is shown with emphasized background color. These two views should support each other, both aiding in understanding the runtime behavior of computer programs.



(a) Initial state (b) Constructor call (c) Before the loop

Figure 1: The three first steps of our debugging session

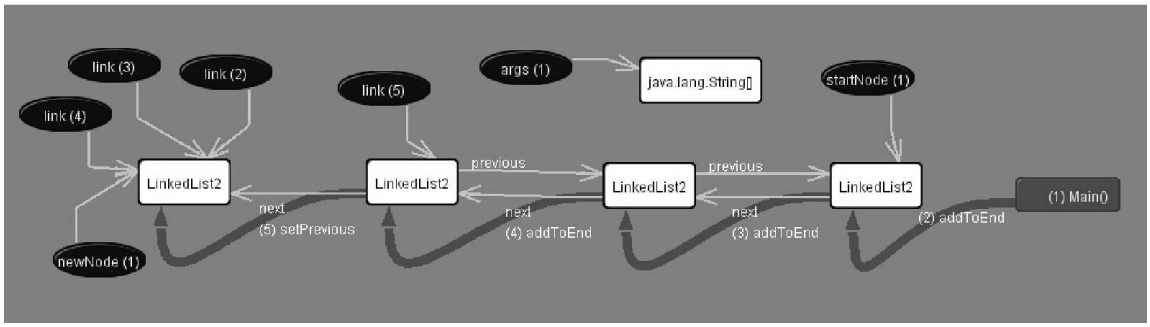


Figure 2: Program state after a number of steps

```
public class LinkedList2 {
   private LinkedList2 next;
   private LinkedList2 previous;
   private int value;

   public LinkedList2(int i) {
      next = null;
      previous = null;
      value = i;
   }

   public void addToEnd(LinkedList2 link) {
      if (this.next != null)
         next.addToEnd(link);
      else{
         this.next = link;
         this.next.setPrevious(this);
      }

      value++;
   }

   public void setPrevious(LinkedList2 link) {
      this.previous = link;
   }
}
```

Table 1: LinkedList2.java

```
public class Example{

   public static void main(String[] args) {
      LinkedList2 startNode = new LinkedList2(0);

      for (int i=1; i<4; i++) {
         LinkedList2 newNode = new LinkedList2(i);
         startNode.addToEnd(newNode);
      }
   }
}
```

Table 2: Example.java

5 Discussion

We have created a new tool for programming education for novices. This tool can be used to visualize some otherwise complex concepts such as recursion, references, stack etc. We plan to use the tool during lectures to better explain program examples. Experiences and examples of using a visualization tool for explaining program examples during lectures can for example be found in Cross et al. (2002).

The true challenge lies with the students using the program on their own to explore both lecture examples and their own programs. We have a hypothesis that seeing how variables change their values and how program execution proceeds through the code from line to line should help in building a correct mental model of program execution. Nevertheless, we can never be certain what students think caused the behavior they saw. One possible approach to evaluation of the tool in student use is to use a questionnaire that is filled while observing the execution of simple program. The questions probe typical misconceptions related to objects and classes as well as those related to references between objects. Our hypothesis of the results is that the comparison between students that try the exercise with or without the tool should show differences in the train of thought.

Evaluating the effect of using the tool on lectures is also not straightforward. As the tool essentially changes how examples are presented and code examined, it is hard to say which part of the observed effect is due to the tool and which is due to other changes in the style of presenting subjects or changes in how the time is divided between topics. While we cannot directly evaluate learning results, we can still collect student impressions using anonymous surveys as suggested in (Naps et al., 2003).

To what extent the program supports understanding the runtime state of the program remains to be seen. We aim to introduce the tool on our programming course in Fall 2004.

References

James H. Cross, T. Dean Hendrix, and Larry A. Barowski. Using the debugger as an integral part of teaching CS1. In *32nd ASEE/IEEE Frontiers in Education Conference*, volume 2, pages F1G–1–F1G–6. IEEE, November 2002.

Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. *ACM SIGCSE Bulletin*, 29(1):131–134, 1997.

Dean F. Jerding and John T. Stasko. Using visualization to foster object-oriented program understanding. Technical Report GIT-GVU-94-33, Graphics, Visualization and Usability Center, College of Computing, Georgia University of Technology, Atlanta, 1994.

Michael Kölling and John Rosenberg. Guidelines for teaching object orientation with java. In *The Proceedings of the 6th conference on Information Technology in Computer Science Education*, pages 33–36. ACM, 2001.

Michael Kölling and John Rosenberg. BlueJ - the Hitch-Hikers Guide to Object Orientation. Technical Report 2, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, September 2002.

Ronit Ben-Passat Levy, Mordechai Ben-Ari, and Pekka Uronen. An Extended Experiment with Jeliot 2000. In *Proceedings of the First Program Visualization Workshop*, pages 131–140. University of Joensuu, 2001.

Paul Mulholland and Marc Eisenstadt. Using software to teach computer programming: Past, present and future. In John Stasko, John Domingue, Marc Brown, and Blaine Price, editors, *Software Visualization: Programming as a Multimedia Experience*, chapter 10, pages 399–408. MIT Press, 1998.

Thomas Naps, Guido Rössling, Jay Anderson, Stephen Cooper, Wanda Dann, Rudolf Fleisher, Boris Koldehofe, Ari Korhonen, Marja Kuittinen, Charles Leska, Lauri Malmi, Myles McNally, Jarmo Rantakokko, and Rockford Ross. Evaluating the educational impact of visualization. *SIGCSE Bulletin*, 35(4):124–136, December 2003.

Rainer Oechsle and Thomas Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface(jdi). In Stephan Diehl, editor, *Software Visualization, State-of-the-art survey*, pages 176–190. Springer, 2002.

The Object Management Group. *Unified Modeling Language (UML) version 1.5*, 2003. Available on a webpage : `http://www.omg.org/technology/documents/formal/uml.htm` (last checked September 29^{th} 2004).

Andreas Zeller and Dorothea Lütkehaus. DDD—A Free Graphical Front-End for UNIX Debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.