jGRASP: A Simple, Visual, Intuitive Programming Environment for CS1 and CS2

By Alexander Miller, *University of Washington*, Stuart Reges, *University of Washington*, Allison Obourn, *University of Arizona*

Instructors of CS1 and CS2 have access to a wide range of sophisticated Integrated Development Environments (IDEs) to choose from, and most are free. Having such a wealth of options can be overwhelming, especially given that each instructor must consider the most relevant issues for their institution. This paper describes the experience of the authors—instructors at a large, public university—in using the jGRASP IDE to teach both a Java CS1 course and a Java CS2 course. The primary considerations were ease of installation and use, support for interactive programming exercises, and debugging support with intuitive visual representations of data structures.

No programming course can exist without a code editor, both for the lecturer to teach with and the students to learn with. The choice of editor greatly impacts the quality of the course, but the abundance of editors doesn't make the choice easy. At one end of the spectrum, an instructor could choose a simple text editor such as TextPad [10] that includes support for syntax highlighting and program compilation and execution, but no support for debugging. On the other end of the spectrum, an instructor could choose a professional-strength, feature-rich IDE such as Eclipse—described on the download page as an environment intended for "developers" [7].

At the University of Washington, we have settled on using a development environment in between these two extremes. We wanted an environment with a debugger, but we also wanted to avoid the complexity of a sophisticated editor like Eclipse. In addition, we were convinced by our experience with DrJava [5] that an IDE with a read-eval-print loop (REPL) feature could be helpful as teaching tool for instructors and as an exploration tool for students [1]. Finally, because of the large numbers of students taking our courses, we wanted an environment that would install consistently across multiple platforms with minimal configuration.

CS1 AND CS2 AT THE UNIVERSITY OF WASHINGTON

A choice of editor is context specific; at the University of Washington, our CS1 and CS2 courses have the following properties:

- *Large lecture sizes*. In the autumn quarter of 2016, there were 1,078 CS1 students (split between two lecture times) and 408 CS2 students. The same set of lecturers rotate between the CS1 and CS2 courses every quarter. Code walkthroughs, in which the instructor writes code along with the class, make up a large portion of our lectures.
- *Undergraduate teaching assistants*. Teaching assistants (TAs) lead weekly sections that supplement lectures and give students hands-on programming experience through solving sample problems. Each section consists of around 20 students. We had 73 TAs in autumn 2016. TAs choose between CS1 and CS2 every quarter.
- *Weekly projects*. Students are assigned project-based homework that requires them to write Java programs from scratch at home. They can receive homework help from TAs in a dedicated lab.

Our relatively large class sizes have influenced our choice of editor. For example, because it is difficult to provide consistent, individualized editor support for our students, simplicity and ease of use were of special importance to us.

AN INTRODUCTION TO jGRASP

jGRASP is a lightweight integrated development environment created and maintained by the Department of Computer Science and Software Engineering at Auburn University [8]. Its target audience is teachers and students, which means its suite of features was picked with the novice programmer in mind. The creators of jGRASP have written several papers describing its features [2,3,4,5].

The features that have proven most useful for us are:

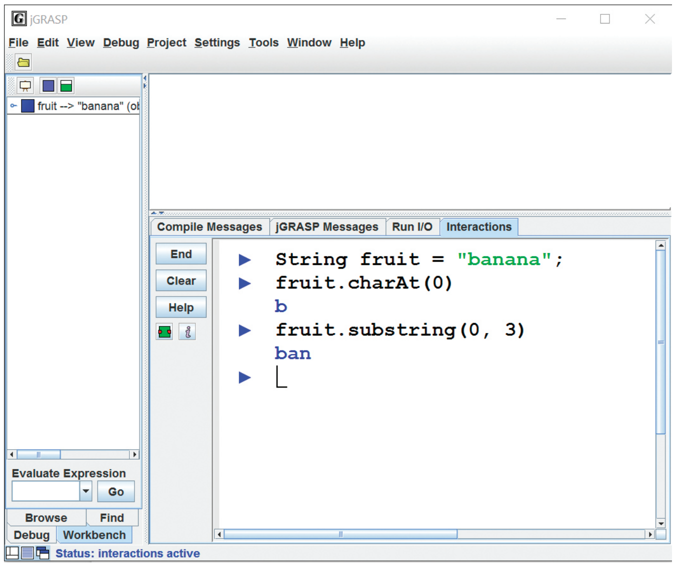
- *Simplicity*. While jGRASP supports standard IDE features, such as program compilation and execution, debugging, and syntax highlighting, it is missing certain advanced features like code autocomplete and version control integration. We consider the lack of these advanced features to be an asset, because they don't intimidate novice programmers. In addition, although jGRASP does allow code to be organized into projects, a source file does not need to be part of a project to be run. This contrasts with Eclipse, which requires all code to be part of a project—something we've found confuses students.
- *Easy installation across platforms*. There are few configuration issues even with thousands of students installing the program on a wide variety of platforms.
- *Interactions pane*. jGRASP's interactive window provides a REPL for exploring Java code snippets (note that JDK 9's JShell, new at the time of this writing, makes this functionality available outside of jGRASP).
- *Simple debugging*. We leverage jGRASP's debugging tool in the lecture to step through code, and students can use the debugger to fix problems in their own code.
- *Data structure viewers*. jGRASP can display a wide range of data structures, including arrays, ArrayLists, linked lists, and binary trees. These visualization capabilities are integrated with jGRASP's debugger.

Many environments provide some of these features, but jGRASP packages them into a single tool that exposes a good level of functionality for novice programmers. The data structure viewers are particularly unique relative to other popular IDEs.

We were also pleased that we could use a single IDE that would be appropriate for both CS1 and CS2. We have attempted to minimize any uncertainty and nervousness a CS1 student might experience in considering whether to continue to CS2, so it has been helpful for students to know that they will continue to use the same familiar IDE if they continue in the sequence.

jGRASP IN THE LECTURE

Instructors use jGRASP when giving lectures, taking advantage of a variety of features that allow lectures to be dynamic. Consider, for example, how one might introduce the String type in Java. One option would be to create a PowerPoint slide with examples of String usage—although we consider that approach dull. A more hands-on instructor might write example code in an IDE, but this requires compiling and executing an entire program. We've found the interactions pane to be ideal for introducing topics like the String type: instructors can type a variable declaration, and then see how Java evaluates various expressions involving that variable. Figure 1 shows how that might look to a student after the instructor types in some code. The lecturer might accompany the code in Figure 1 with the following narration: "Now I'm going to declare a String variable called fruit. Notice how jGRASP lists it as a new variable defined in this scope. Now I'm going to ask it for the character at position 0 and it says b. Now I'm going to ask for a substring…"



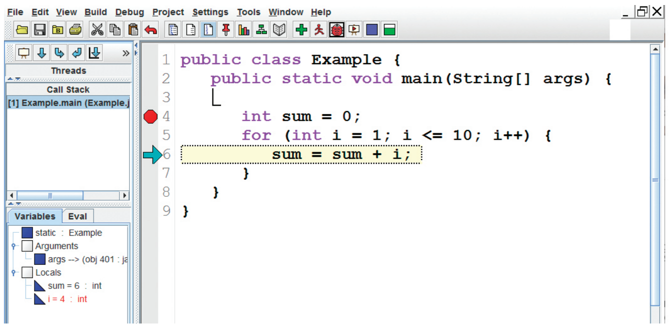
fruit --> "banana" (ol

```
▶ String fruit = "banana";
▶ fruit.charAt(0)
  b
▶ fruit.substring(0, 3)
  ban
▶
```

Status: interactions active

Figure 1: The instructor can use the jGRASP interactions pane, a REPL interface for Java, to demonstrate String methods in the lecture.

Furthermore, the jGRASP debugger provides a dynamic representation of program execution. Figure 2 demonstrates how a CS1 instructor can set a breakpoint with the debugger and then step through the execution of a for loop, watching how local variables change as the loop executes.



[1] Example.main (Example.j

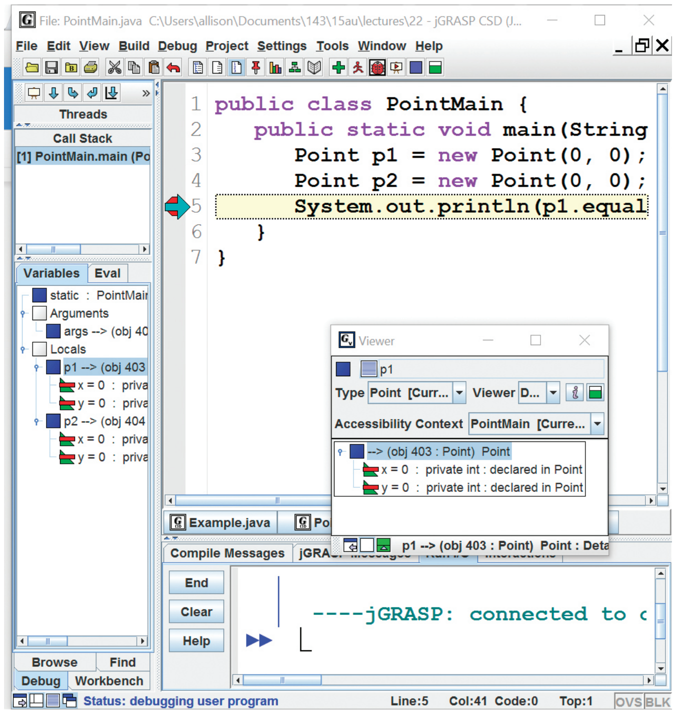
```
public class Example {
   public static void main(String[] args) {

      int sum = 0;
      for (int i = 1; i <= 10; i++) {
         sum = sum + i;
      }
   }
}
```

static : Example
Arguments
args --> (obj 401 : ja
Locals
sum = 6 : int
i = 4 : int

Figure 2: Example of instructor use of the debugger in the lecture.

The debugger also lends itself well for teaching Java objects. Figure 3 demonstrates how the instructor can inspect individual objects to see the fields inside. The debugger provides a benefit of numbering the objects, which allows the instructor to discuss subtle concepts like the difference between a comparison with the == operator and a comparison with Java's equals method. For example, an instructor might state, "See how jGRASP is referring to the first point as object 403 and the second as object 404. They are equivalent in the sense that they both contain coordinates of 0 and 0, but they are not the same object."



File: PointMain.java C:\Users\allison\Documents\143\15au\lectures\22 - jGRASP CSD (J...

```
public class PointMain {
    public static void main(String
        Point p1 = new Point(0, 0);
        Point p2 = new Point(0, 0);
        System.out.println(p1.equal
    }
}
```

Call Stack: [1] PointMain.main (Po

```
static : PointMai
Arguments
    args --> (obj 40
Locals
    p1 --> (obj 403
        x = 0 : priva
        y = 0 : priva
    p2 --> (obj 404
        x = 0 : priva
        y = 0 : priva
```

Viewer: p1 — Type: Point [Curr... — Viewer: D... — Accessibility Context: PointMain [Curre...

```
--> (obj 403 : Point)  Point
    x = 0 : private int : declared in Point
    y = 0 : private int : declared in Point
```

p1 --> (obj 403 : Point) Point : Deta

```
----jGRASP: connected to c
```

Status: debugging user program   Line:5   Col:41   Code:0   Top:1

Figure 3: Example of instructor use of the debugger to show the internal state of objects in the lecture.

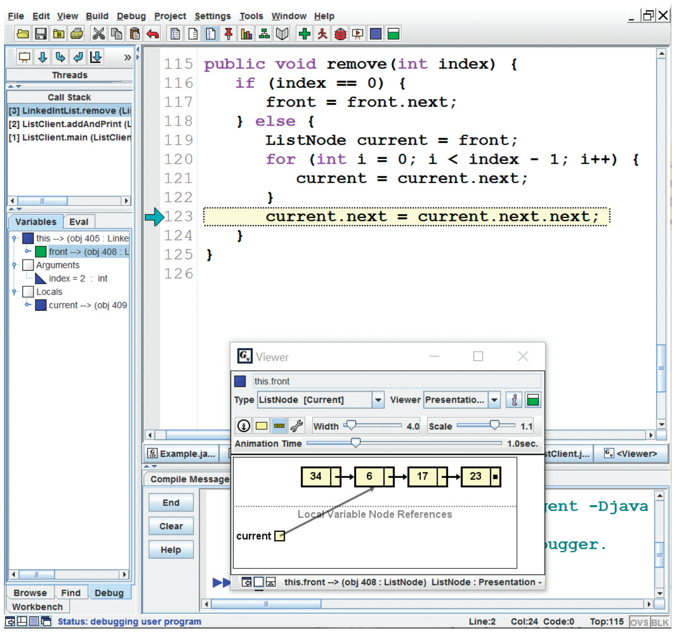
Finally, instructors can leverage jGRASP's dynamic object viewer tools to help students develop mental models for how program execution changes the state of data structures. Consider the case where an instructor is explaining how to manipulate the nodes of a linked list. As shown in Figure 4, jGRASP's rendering of the linked list structure gives students a picture of what the code is doing (in this case, removing a node from the list). We've also found that the viewers are effective in demonstrating why incorrect code doesn't work. For example, by changing line 123 in Figure 4 from:

```
current.next = current.next.next
```

To:

```
current = current.next;
```

and rerunning the code with the viewer open, the students can see why this seemingly intuitive solution fails to manipulate the linked list correctly.



```
115 public void remove(int index) {
116     if (index == 0) {
117         front = front.next;
118     } else {
119         ListNode current = front;
120         for (int i = 0; i < index - 1; i++) {
121             current = current.next;
122         }
123         current.next = current.next.next;
124     }
125 }
126
```

Call Stack: [3] LinkedIntList.remove (Li — [2] ListClient.addAndPrint (L — [1] ListClient.main (ListClien

```
this --> (obj 405 : Linke
    front --> (obj 408 : L
Arguments
    index = 2 : int
Locals
    current --> (obj 409
```

Viewer: this.front — Type: ListNode [Current] — Viewer: Presentatio... — Width 4.0 — Scale 1.1 — Animation Time 1.0sec.

this.front --> (obj 408 : ListNode) ListNode : Presentation -

Status: debugging user program   Line:2   Col:24   Code:0   Top:115

Figure 4: Example of instructor use of the data structure viewer to show exactly what linked list remove code is doing in the lecture.

The common theme for these examples is the ability to make visible what is normally invisible to students: the internal execution of a program. We've found that jGRASP gives instructors a useful set of tools for discussing program execution in a lecture setting.

jGRASP FOR TEACHING ASSISTANTS

We employ a small army of approximately 80 undergraduate teaching assistants (TAs) that lead supplemental sections for CS1 and CS2. A teaching section involves reviewing general concepts from the lecture and working through sample programming problems. TAs are not required to use jGRASP in the section, but many choose to—of 60 TAs who responded to a survey, 95% used it in teaching. Considering that TAs learned Java with jGRASP themselves—and the TAs' desire to remain consistent with the lecture—this high usage of jGRASP does not surprise us.

TAs can leverage jGRASP's features to lead their sections in the same ways instructors use jGRASP in the lecture, as described previously (for example, by stepping through broken code in the jGRASP debugger, or by using the interactions pane to demonstrate Java syntax). The debugger specifically was cited by many TAs as essential to teaching certain topics. For example, one TA stated: "The debugger is really helpful when explaining confusing topics like recursive backtracking." Another TA raved about the data structure viewers: "[it's] extremely helpful for teaching because the diagrams it generates are very intuitive and I can make changes to code based on student suggestions to immediately show how that changes execution."

Both the debugger and the data structure viewers are consistently popular across all TAs—of those TAs who use jGRASP to teach, 90% use the jGRASP debugger, and 90% use the data structure viewers. But other features are better suited to teaching CS1 than CS2. The interactions pane, which is good for exploring Java basics early in a CS curriculum, finds more use among CS1 TAs (with usage at roughly 80%) than CS2 TAs (with usage at 70%). Executing single-line Java statements in a REPL interface isn't always useful for teaching CS2 topics, because exploring more sophisticated CS programming concepts sometimes requires writing a complete program.

While TAs have a generally positive opinion of jGRASP, Figure 5 shows that CS1 TAs rate the usefulness of jGRASP higher than CS2 TAs. We believe that jGRASP can be used effectively in both CS1 and CS2, but jGRASP's weaknesses, such as the lack of code autocomplete, make it potentially less attractive for a CS2 context. We discuss this further in the Limitations section below.

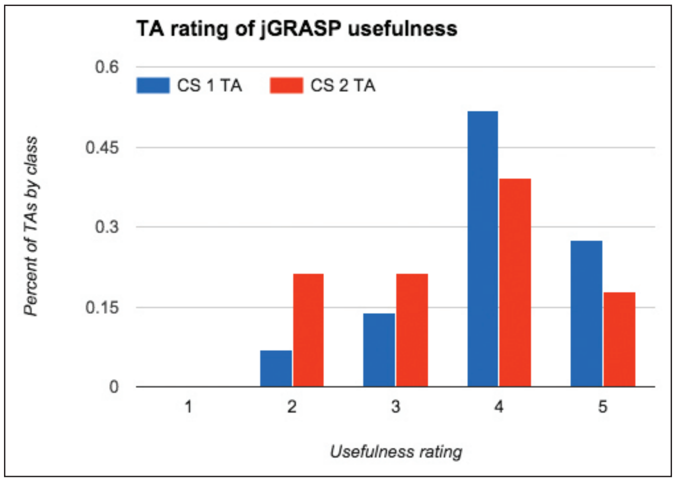


Figure 5: CS1 and CS2 TAs were asked to rate the usefulness of jGRASP on a scale from 1 (not helpful) to 5 (very helpful). CS1 TAs rated jGRASP higher than CS2 TAs, although both groups considered jGRASP helpful.

While we don't go so far as to mandate jGRASP usage among TAs, we're convinced that the wide adoption of jGRASP by our TA community contributes to jGRASP's effectiveness in our program. TAs can reinforce the topics learned in the lecture using an interface common to that of the lecture, and can offer technical support for students who are learning to use an IDE for the first time.

jGRASP FOR STUDENTS

Although students are not required to use jGRASP in CS1 and CS2 (except for an optional lab section), many do, as indicated in Figure 6. In a poll of students attending their section, 86% of CS1 students and 72% of CS2 indicated that they used jGRASP. Why the drop in CS2 usage? We speculate that CS2 students are less likely to be intimidated by more advanced IDEs like Eclipse, and benefit more from features like Eclipse's code autocomplete while working on larger CS2 homework projects. In addition, some CS2 students took CS1 at a different school, and are more familiar with a different editor.

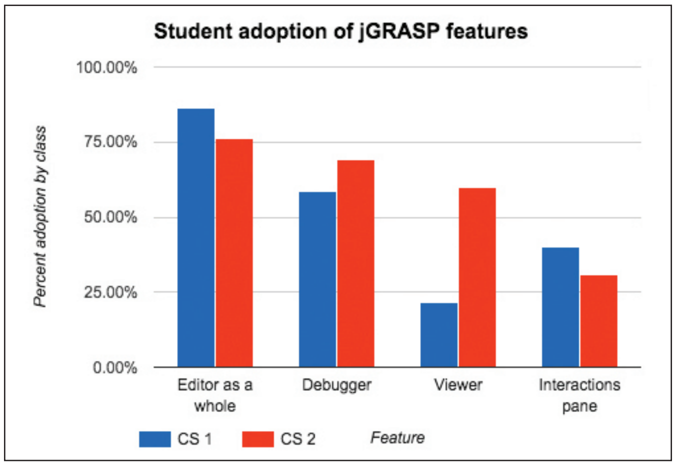


Figure 6: Slightly more CS1 students use jGRASP than CS2, but CS2 students make more use of the debugger and the data structure viewers.

Below, we discuss the usage patterns of students we observe for certain key features of jGRASP: the debugger, the data structure viewers, and the interactions pane.

STUDENT USAGE OF THE DEBUGGER

jGRASP's debugger is ideal for beginner programmers in two respects—first, the debugging interface is somewhat simpler than an industry level debugging tool like Eclipse; and secondly, the debugger's integration with other jGRASP features, like data structure viewers, provides a highly visual debugging experience.

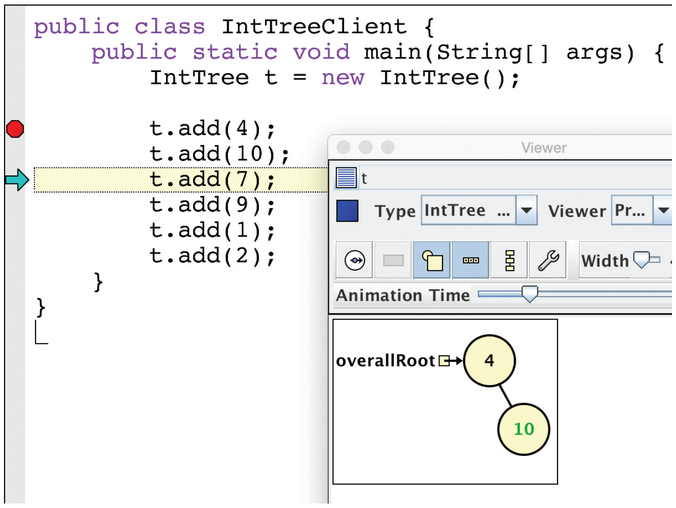
Although the instructors and TAs emphasize the quality of jGRASP's debugger, not all students end up making use of it. In a rough poll, we found that about 58% of CS1 students had used the debugger.

We found a higher debugger adoption rate among CS2 students (77%). There are many possible explanations. It's possible that students who use the debugger are more likely to succeed in CS1 and go on to take CS2. It's also possible that those students who choose to take a second programming course tend to be more deliberate about debugging their programs. Finally, it could be that programs written in CS2 are complicated enough to warrant more debugging (in other words, students run into more bugs—or more subtle bugs—so they turn to the debugger more). We suspect a combination of these causes is responsible for the increased debugger usage among CS2 students.

STUDENT USAGE OF THE OBJECT VIEWERS

While we are treating the debugger and the data structure viewers as separate features here, we should again emphasize that the two are tightly integrated; the viewers complement the debugger. For example, with the execution of a program paused in debug mode, a student can drag out a linked list to view its contents, and then step through the program to see how the state of the linked list is transformed step-by-step to discover where she went wrong (see Figures 7a and 7b).

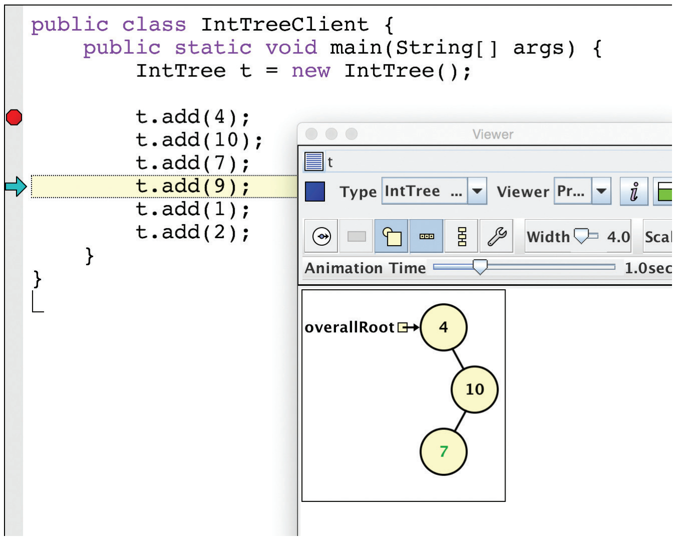
Like debugger usage, data structure viewer adoption is low among CS1 students, at 22%. CS1 students work with simple data structures, like strings and arrays, which don't benefit as much from jGRASP's visualization—they can be visualized intuitively. On the other hand, CS2 students, faced with complicated data structures such as linked lists and binary trees, find the data structure viewers invaluable to debugging their code: 71% of CS2 students said they used the feature.



```
public class IntTreeClient {
    public static void main(String[] args) {
        IntTree t = new IntTree();

        t.add(4);
        t.add(10);
        t.add(7);
        t.add(9);
        t.add(1);
        t.add(2);
    }
}
```

Figure 7a: A student wanting to verify their linked list code can use jGRASP's data structure viewers in conjunction with the debugger. Here, the student has paused execution of their program after two nodes have been added to the linked list.



```
public class IntTreeClient {
    public static void main(String[] args) {
        IntTree t = new IntTree();

        t.add(4);
        t.add(10);
        t.add(7);
        t.add(9);
        t.add(1);
        t.add(2);
    }
}
```

Figure 7b: When the student steps over a line of code in the debugger, they can see the live transformation of their binary tree.

STUDENT USAGE OF THE INTERACTIONS PANE

The interactions pane shines in the lecture for exploring the small parts of Java—how to construct a String and use the substring method on it, for example. If they wish to review syntax and structures presented in the lecture, a student can leverage the interactions pane in the same way on their own. We find this use case less common though. Less than half of CS1 and CS2 students said that they used the interactions pane.

jGRASP FOR HOMEWORK HELP

Our students visit a dedicated lab to get homework help from TAs. The CS1 and CS2 instructors emphasize to TAs that they should not give away solutions ("coding by TA," as we call it), but should help students discover the solution on their own—a challenging task.

We believe jGRASP lends itself particularly well to this context. Rather than telling a student why their program doesn't produce the correct output, for example, a TA can ask the student to open the jGRASP debugger and step through the program with the student. A TA can have a student working on a binary tree homework open the data structure viewer to see exactly which line of code causes the state of the tree to deviate from what the student expected. Students come away from the interaction not only having conquered the bug in their program, but having learned a new skill—how to use a debugger. One TA stated that "...the debugger is really helpful when students know how to use it, and I always teach students in the lab to use it." Another TA said that "It's very nice for the students to be able to see visual representation of their code working. It definitely helps visual learners see what their code is doing." One TA contrasted jGRASP's debugger with more powerful, but complex, debuggers, saying that "The [jGRASP] debugging tool is easy, and we don't need to worry about as many details as with Eclipse."

Using jGRASP (or any similar tool, for that matter) works well when TAs and lecturers provide instruction and support for it. Students see jGRASP used in the lecture, are required to use jGRASP to do lab assignments, and are given one-on-one support for jGRASP when completing homework. It provides a common set of tools and vocabulary for teachers and students to share and communicate with. This thorough integration of jGRASP makes the program effective in our introductory programming courses.

LIMITATIONS OF jGRASP

When we surveyed our TAs, they mentioned three features that they would like to see in jGRASP:

- *Auto-indent.* They wish that jGRASP would either automatically maintain proper indentation or provide an option for reformatting a program to correct indentation. (After reviewing this paper, the maintainers of jGRASP have stated that they are addressing this issue.)
- *Code Autocomplete.* TAs like the way that Eclipse and other IDEs suggest possible completions when, for example, a user types a variable name followed by a dot. While this is unquestionably a useful feature in many contexts, it was not universally supported by TAs. One TA said that "I [...] like that jGRASP doesn't auto-complete any code or offer suggestions to fix type mismatches or casting issues, because it really forces students to understand the code they're writing." Another commented that "It's very beginner friendly. I love the debugger tool and the fact that it doesn't auto-complete."

- *Incremental compilation*. Some IDEs underline compilation errors in red as the user types, giving immediate, real time feedback. jGRASP does not have this functionality.

In addition, we've found the following two small points of friction with jGRASP:

- *Language issues*. Students who have their computers set to certain languages (for example, Chinese) must manually configure jGRASP to produce English compiler messages.
- *Object viewer configuration*. When viewing an object with more than one data field, the user must configure which field should be displayed in the viewer (for example, for a linked list structure where every node has two pieces of data).

CONCLUSION

jGRASP meets our needs for a lightweight IDE suitable for beginning programmers. It is valued by instructors, TAs and students. One indication of its popularity among our students is that many continue to use it in upper level courses, even though the faculty may prefer a more complex professional grade IDE.

Our experiences suggest several open questions that would be useful to explore. The jGRASP interactions pane is a rare feature among Java IDEs, but one that our instructors value greatly. Given that many universities use Python where a REPL is a standard feature, we wonder whether careful research might validate the use of a REPL in the lecture as a best practice for introductory computer science courses. We also find ourselves asking the perennial question of why we don't see more students using the debugger, particularly in CS1. Finally, we know that the instructors and TAs find the jGRASP object viewers helpful, but we have not had a chance to gather evidence that would allow us to conclude whether it is helpful for students.

Each university will need to consider its own challenges when choosing an IDE. Because we teach very large CS1 and CS2 courses that are taken mostly by non-majors, we care a lot about simplicity and ease of installation. jGRASP has met our needs in this regard. In addition, jGRASP's interactions pane and object viewers provide unique tools for instructors and TAs to teach the internal execution of programs. Finally, we believe that the consistent use of a single editor across all layers of a large University course—whether it be jGRASP, or some other IDE—greatly impacts the effectiveness of the course. ❖

References

1. Allen, E., Cartwright, R., and Stoler, B. DrJava: a lightweight pedagogic environment for Java. *SIGCSE Bull.* 34, 1 (February 2002), 137-141.
2. Cross, J., Hendrix, T., and Barowski, L. Using the debugger as an integral part of teaching CS1. *Frontiers in Education, FIE 2002. 32nd Annual*, 2002, pp. F1G-1-F1G-6 vol.2.
3. Cross, J., and Hendrix, T. jGRASP: a lightweight IDE with dynamic object viewers for CS1 and CS2. *SIGCSE Bull.* 38, 3 (June 2006), 356-356.
4. Cross, J., Hendrix, T., Jain, J., and Barowski, L. Dynamic object viewers for data structures. *SIGCSE Bull.* 39, 1 (March 2007), 4-8.
5. Cross, J., Hendrix, T., Umphress, D., and Barowski, L. Exploring accessibility and visibility relationships in java. *SIGCSE Bull.* 40, 3 (June 2008), 103-108.
6. DrJava site; http://www.drjava.org/. Accessed 2016 April 27.
7. Eclipse download site; https://eclipse.org/downloads/. Accessed 2016 April 27.
8. jGRASP main site; http://www.jgrasp.org/. Accessed 2016 April 27.
9. JShell and REPL; https://blogs.oracle.com/java/entry/jshell_and_relp_in_java. Accessed 2016 June 13.
10. TextPad site; https://www.textpad.com/. Accessed 2016 April 27

Alexander D. Miller
University of Washington
Paul G. Allen Center for Computer Science & Engineering
185 E Stevens Way NE, Seattle, WA 98195
amiller@cs.washington.edu

Stuart Reges
University of Washington
Paul G. Allen Center for Computer Science & Engineering
185 E Stevens Way NE, Seattle, WA 98195
reges@cs.washington.edu

Allison Obourn
(Formerly University of Washington)
University of Arizona, Department of Computer Science
1040 4th St, Tucson, AZ 85721
urn@cs.washington.edu

DOI: 10.1145/3148562