A JAVA FUNCTION VISUALIZER

Herbert L. Dershem
Department of Computer Science
Hope College
Holland, MI 49422-9000
(616) 395-7508
dershem@cs.hope.edu

Daisy Erin Parker
Department of Computer Science
College of William & Mary
Williamsburg, VA 23187
depark@mail.wm.edu

Rebecca Weinhold
Department of Computer Science
Kalamazoo College
Kalamazoo, MI 49006
k96rw01@kzoo.edu

ABSTRACT

Visualization is an important tool for student learning. The Function Visualizer provides a resource that permits the visualization of function calls for any set of Java functions. It provides a control panel where the user specifies the speed of the visualization and the parameter values for the initial function call. When the visualization begins, lines of code are highlighted in a function window as they are executed and when another function is called, a new window appears on top of the previous one, producing a visualization of the function call stack. This tool provides a flexible and intuitive visualization aid for the teaching of recursion.

INTRODUCTION

Visualization is an important classroom tool. All teachers use it to help their students "see" difficult concepts. A recent overview paper [1] listed many motivating factors for the use of visualization including better learning, attracting student attention, faster coverage of topics, increased student understanding, encouraging under-prepared students, facilitating debugging, and using classroom time efficiently.

There are many opportunities for visualization in Computer Science instruction and it is natural to use the computer to generate such visualizations dynamically and interactively. Work in Computer Science instructional visualization has been focused on

two specific areas: program visualization and algorithm animation. A taxonomy of the former category is found in [5] and much recent work has been done in the DynaLab project [2]. Algorithm animation work has been done by many including Brown [3], Stasko [6], and Naps [4].

One of the most difficult concepts for Computer Science students to learn is that of recursion. In this paper we describe a tool that generates visualizations of function calls for any set of Java functions (methods). In particular, this tool enables students to view recursive calls in a manner that traces through the statement of the functions on a window and pops up a new function window every time a function call is executed. This gives a trace of the function calls and a visual representation of the call stack.

BACKGROUND

Work on the Function Visualization project was begun by James Vanderhyde, who created the Function Visualization tool as a visual way to teach recursion as a part of his work on the Object Visualizer [7]. The original Function Visualizer gives the user a choice between two functions to visualize, Factorial and QuickSort. Embedded in the Java code of each of these functions are code lines for executing actions of the code windows, as well as a string representing the actual text to appear in these windows. Extending the visualizer to other functions is difficult because the code for the functions must be so carefully constructed. The original Function Visualizer also allowed very little interaction with the user. Our work generalizes the original tool to allow the visualization of other functions and incorporate more user involvement in the visualization.

All improvements to the original Function Visualizer have been in an effort to generalize the tool. The Function Visualizer now accepts the Java source of any user-provided function, supports the use of non-standard classes as parameter types, has an interface that allows the user to input values for the function's parameters, and provides the user with the return value or the status of the parameters upon the function's completion.

COMPONENTS OF THE FUNCTION VISUALIZER

The Control Panel

The Function Visualizer control panel provides control to the user in several ways (Figure 1). The "Start Function" button allows the user control in initiating the visualization. When the user clicks on the "Start Function" button, execution of the user-provided function will begin and the code windows start appearing. Once the function is running, the button's label changes to "Stop". Clicking on the button at this point will interrupt the function's execution and destroy any code windows that are up. When the function's execution completes or is interrupted, the button's label changes to "Clear Parameters & Restart Function." Clicking the button at this point will clear the parameter input fields, hide the parameter result frame, and basically reload the function. The button label changes back to "Start Function", and the visualizer is again ready for the user to begin the function's execution.

FIGURE 4. Java Code for fracSort Function using the Class Frac

```
import Frac;
public void fracSort(int start, int end, Frac[] fracArray)
{
qsort(start, end, fracArray);
}

public void qsort(int start,int end,Frac[] array)
{
if (start<end)
{
int split=partition(start,end,array);
qsort(start,split,array);
qsort(split+1,end,array);
}
}

public int partition(int start,int end,Frac[] array)
{
int top = end;
int bottom = start;
Frac pivot = array[(start+end)/2];
Frac temp;
while (top>bottom)
{
while ((bottom<top) && (pivot.lessThan(array[bottom],pivot)))
{
 bottom++;
}
while ((bottom<top) && (pivot.lessThan(pivot,array[top])))
{
 top--;
}
if (bottom<top)
{
temp=array[bottom];
array[bottom]=array[top];
array[top]=temp;
}
}
if (bottom<top)
{
temp=array[bottom];
array[bottom]=array[top];
array[top]=temp;
}
return top;
}
```

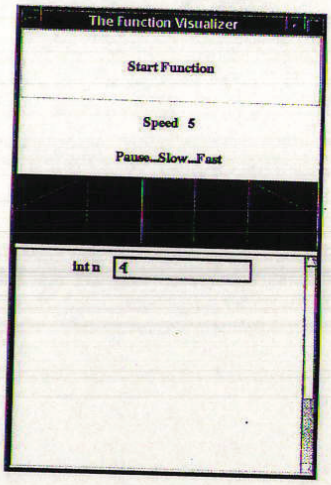
User Provided Function

One of the main features of the Function Visualizer is the minimal form in which the user supplies the function to be visualized. Apart from the Java code for the function itself, very little is required. User-defined classes are supported: the function code need only include the appropriate `import` statement and the class must be provided to the function visualizer. An example of a function that includes a parameter of a user-defined class can be seen in Figure 4 below. The function `fracSort()` performs a quicksort algorithm on an array of class `Frac` (fractions). The first method in the file is assumed by the Function Visualizer to be the initial function to be visualized, while any additional functions such as `qsort()` and `partition()` are considered to be secondary methods called by the first.

Preprocessing of the User-Provided Function

In order to visualize the user's function, the preprocessing portion of the Function Visualizer creates a text file and a Java file from the user's code. The text file is used to provide the contents of the code windows described above and it differs from the user's code only superficially. The text in the file is indented uniformly and the use of curly braces is also regulated. The Java file is generated specifically to work with the Function Visualizer and its purpose is to extend and utilize a specially designed parent class of the Function Visualizer. The Java file preserves the original code from the user's function with additional code inserted to effect the visualization.

FIGURE 5. Control Window for Fibonacci Function Visualization



Visualization of the User-Provided Function

After the Perl preprocessing has been completed successfully, the Java file that was generated can be used to visualize the user's original function. The generated Java file contains code that calls the parent class implementations of several visualization methods that control the creation, destruction, and highlighting of code in the code windows in addition to the actual code that was given in the user's function. Using the control panel, the user is able to enter values for each parameter of the original function, start the visualization of the function, and control speed of execution.

FIGURE 6a. A snapshot of the initial call to fibonacci as it is preparing to make its first recursive call.

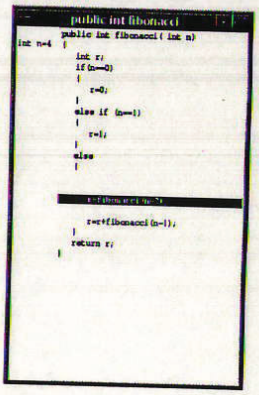
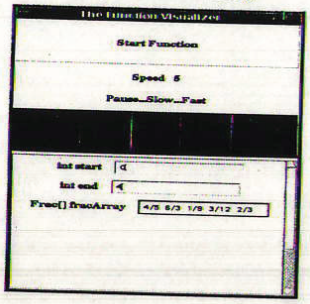


FIGURE 1.The Control Panel For the Example Function fracSort()



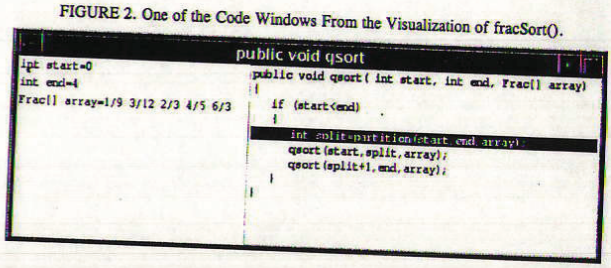
The speed slider allows the user to adjust the function's execution speed. Initially, the speed is set to "5". The user can decrease speed to 0, pausing execution, or increase speed to 10 by clicking on the left or right arrow, respectively. The user can adjust the speed before starting the function, or while the function is running. If the user moves the speed slider while the function is running, its execution is paused and the speed is adjusted.

The parameter input fields accept user-provided values for the function's parameters. For every parameter in the function, an empty text field and a label containing the parameter's type and name are placed on the control panel. The values for the parameters are provided by the user; any legal values will be accepted.

The Code Windows

A code window (Figure 2) is a frame containing pertinent information about the particular function method being visualized. This window pops up when the function is called in the execution of another function. Three important pieces make up any code window. First, the actual text of the method's code appears in the east portion of the frame. This panel of the frame is scrollable if the code text does not fit in the window. Second, the method parameters appear in the west portion of the frame. This panel contains the type, name and value for each parameter of the method. Third, the execution of the function is traced by highlighting each line as it is executed.

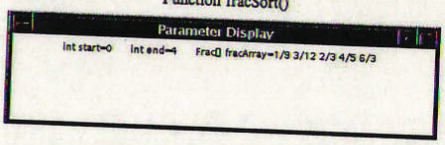
FIGURE 2. One of the Code Windows From the Visualization of fracSort().



The Result Frame

The result frame (Figure 3) appears at the conclusion of a user-provided function's execution. The frame displays the status of parameter values when the function terminates. If the function returns a value, that is the only value displayed. Otherwise, all of the function's parameters are displayed.

FIGURE 3.. The Result Frame From the Visualization of Function fracSort()



STEPS FOR FUNCTION VISUALIZATION

In order to visualize the user-provided function, the Function Visualizer modifies the function, forming new files that are compatible with the tool and are used to produce the visualization.

FIGURE 4. Java Code for fracSort Function using the Class Frac

```
import Frac;
public void fracSort(int start, int end, Frac[] fracArray)
{
qsort(start, end, fracArray);
}

public void qsort(int start,int end,Frac[] array)
{
if (start<end)
{
int split=partition(start,end,array);
qsort(start,split,array);
qsort(split+1,end,array);
}
}

public int partition(int start,int end,Frac[] array)
{
int top = end;
int bottom = start;
Frac pivot = array[(start+end)/2];
Frac temp;
while (top>bottom)
{
while ((bottom<top) && (pivot.lessThan(array[bottom],pivot)))
{
bottom++;
}
while ((bottom<top) && (pivot.lessThan(pivot,array[top])))
{
top--;
}
if (bottom<top)
{
temp=array[bottom];
array[bottom]=array[top];
array[top]=temp;
}
}
if (bottom<top)
{
temp=array[bottom];
array[bottom]=array[top];
array[top]=temp;
}
return top;
}
```

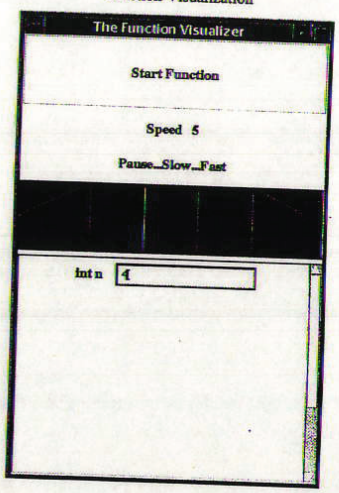
User Provided Function

One of the main features of the Function Visualizer is the minimal form in which the user supplies the function to be visualized. Apart from the Java code for the function itself, very little is required. User-defined classes are supported: the function code need only include the appropriate `import` statement and the class must be provided to the function visualizer. An example of a function that includes a parameter of a user-defined class can be seen in Figure 4 below. The function `fracSort()` performs a quicksort algorithm on an array of class `Frac` (fractions). The first method in the file is assumed by the Function Visualizer to be the initial function to be visualized, while any additional functions such as `qsort()` and `partition()` are considered to be secondary methods called by the first.

Preprocessing of the User-Provided Function

In order to visualize the user's function, the preprocessing portion of the Function Visualizer creates a text file and a Java file from the user's code. The text file is used to provide the contents of the code windows described above and it differs from the user's code only superficially. The text in the file is indented uniformly and the use of curly braces is also regulated. The Java file is generated specifically to work with the Function Visualizer and its purpose is to extend and utilize a specially designed parent class of the Function Visualizer. The Java file preserves the original code from the user's function with additional code inserted to effect the visualization.

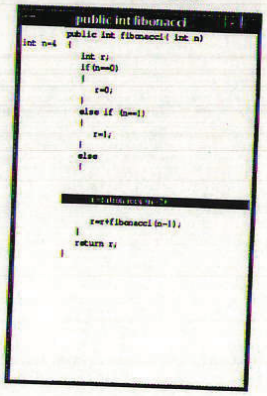
FIGURE 5. Control Window for Fibonacci Function Visualization



Visualization of the User-Provided Function

After the Perl preprocessing has been completed successfully, the Java file that was generated can be used to visualize the user's original function. The generated Java file contains code that calls the parent class implementations of several visualization methods that control the creation, destruction, and highlighting of code in the code windows in addition to the actual code that was given in the user's function. Using the control panel, the user is able to enter values for each parameter of the original function, start the visualization of the function, and control speed of execution.

FIGURE 6a. A snapshot of the initial call to fibonacci as it is preparing to make its first recursive call.



AN EXAMPLE

To illustrate the power of the Function Visualizer to enhance the understanding of recursion, we will follow through an example execution. The function that will be used is the recursive function for evaluating the Fibonacci sequence. After the Java code for this function has been properly processed, the Function Visualizer presents the control window shown in Figure 5.

Next the process of recursive calls is generated. A sequence of snapshots along the way is shown.

CONCLUSIONS AND FUTURE WORK

In summary, there are a number of benefits supplied by the tool described above. The Function Visualizer requires only minimal information from the user. The user submits a function and any methods called by that function. If the function accesses an outside class, the user must include an `import` statement for the class, as well as supplying the `class` file in the base directory. However, the user's function is not required to have a `main()` or contend with the inputting of parameters and the outputting of results. The Function Visualizer control panel allows the user complete control in running the visualization. The user can choose when to begin the visualization, when to terminate it and how fast to run it. Also, the control panel provides for parameter input from the user before the function begins execution. Generalization has succeeded in making the Function Visualizer easy and efficient to use with arbitrary functions. Enhancing the user interaction increases the user's involvement in the visualization. This will hold the user's attention and facilitate the learning of function concepts.

FIGURE 6b. The view immediately after the first recursive call is made.

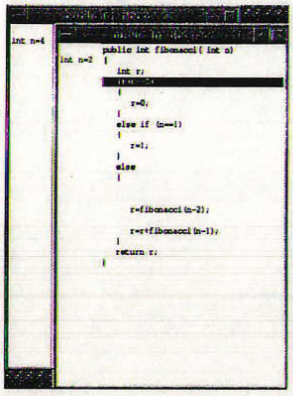
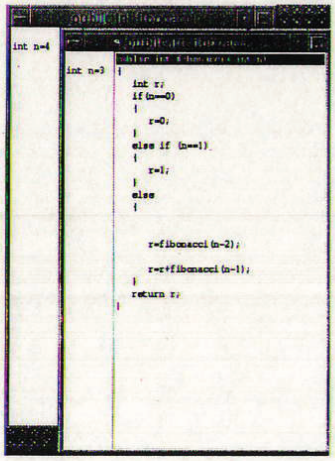


FIGURE 6c. The first recursive call is here preparing to take its first recursive call with n=0.



There are a number of enhancements that we would like to add to the Function Visualizer in the future. As is, the Function Visualizer supports a function with up to ten parameters. The tool would be enriched if it could dynamically support any number of parameters. For functions that call on an outside class, the tool could visualize the class member methods as well as the function's methods. Finally, the Function Visualizer has potential as a debugger. The tool is helpful in detecting run-time errors in the user's function. Because the Function Visualizer illustrates the function's call structure, it has the advantage of producing information that is more visual than an ordinary debugger and more linear in its presentation. Typically, the tool will cease the tracing at a particular code statement and output an exception error message to the console when it encounters a run-time error. The call structure demonstrated up to that point will remain visible and can be analyzed by the user.

FIGURE 6d. This is the view after the recursive call to the left is executed.

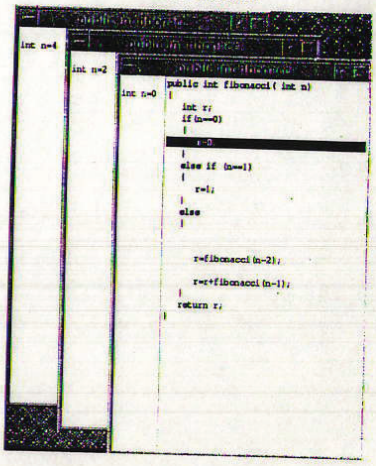
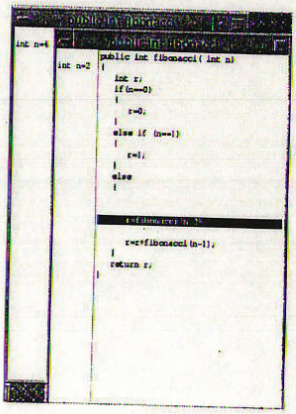


FIGURE 6e. This occurs much later in the execution when the original call has made its second recursive call of fibonacci(n-1).



ACKNOWLEDGEMENTS

Support for this project was provided by the National Science Foundation, grant number #EIA-9732339

REFERENCES

[1] Bergin, J., Brodie, K., Goldweber, M., Jiménez-Peris, R., Khuri, S., Patiño-Martínez, M., McNally, M., Naps, T., Rodger, S., and Wilson, J. An overview of visualization: its use and design. *ACM SIGCSE Bulletin,* 28, (Special Edition, Barcelona), (1996), 192-200.

[2] Boroni, C.M., Eneboe, T.J., Goosey, F.W., Ross, J.A., and Ross, R.J. Dancing with DynaLab. *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education (SIGCSE Bulletin)* 28,1 (Feb 1996), 135-139.

[3] Brown, Marc H. *Algorithm Animation.* MIT Press, Cambridge, MA 1987.

[4] Naps, T.L. and Stenglein, J. Tools for visual exploration of scope and parameter passing in a programming language course. *Proceedings of the Twenty-Seventh Technical Symposium on Computer Science Education (SIGCSE Bulletin)* 28,1 (Feb 1996), 305-309.

[5] Roman, G. and Cox, K.C. A taxonomy of program visualization systems, *IEEE Computer* 26, 12 (Dec 1993), 11-24.

[6] Stasko, J., Tango: a framework and system for algorithm animation. *IEEE Computer* 23,9 (Sep 1990), 27-39.

[7] Dershem, H. and Vanderhyde, J. Java Class Visualization for Teaching Object-Oriented Concepts, *Proceedings of the Twenty-Ninth Technical Symposium on Computer Science Education (SIGCSE Bulletin)* 30,1 (Mar 1998), 53-57.

FIGURE 6f. This snapshot shows a later innermost call where the recursion is now to four levels.

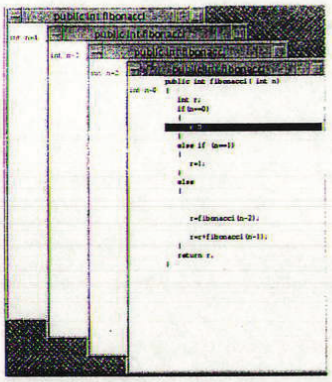
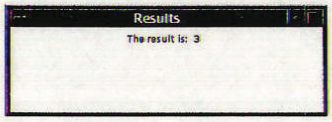


Figure 7. This shows the result of the initial function call as shown in the result window.



PROJECT BASED CS2 COURSE

INTRODUCTION TO COMPUTER SCIENCE, OOP, AND SOFTWARE ENGINEERING

John Norton and Allen Leigh
Westminster College, Salt Lake City, Utah
j-norton@wcslc.edu
a-leigh@wcslc.edu

ABSTRACT

This paper presents a method for using an individual comprehensive project to give CS2 students hands on experience with traditional course materials, software engineering, and project management. The traditional course in CS2 uses a series of stand-alone lab and homework assignments to introduce students to programming and elementary computer science, but it doesn't expose them to some of the more practical aspects of computer science. It is believed that early exposure to practical methods of managing complex projects in computer science will make students more successful in their computer science program. A CS2 course used for three semesters at Westminster College assigns a comprehensive project that emphasizes software engineering principles and project management skills while providing an opportunity to solve a more complex problem. This approach leads to more meaningful involvement with abstract data types, algorithm efficiency comparisons and object-oriented programming.

SETTING THE STAGE

On the first day of the class CS2 students enjoy hearing that there is only one assignment for the class. Their smiles quickly fade when they are told the assignment is worth 50% of their final grade. The assignment is well defined and is presented on the first day of class, and this sets the tone of the semester. A large portion of a student's fear for such a project stems from a perception of the instructor as a gate keeper. The use of a comprehensive project at Westminster has changed this perception and changed student focus to doing an assignment to learn, not just to get it done.

Student/Teacher roles are established immediately and methods for an objective evaluation are discussed in a frank manner. To do this the class is cast into a work environment. Students play a role as members of a software development project at the Star Software Company. Their assignment is to develop a new application to analyze words from text files. The instructor is the project manager, supervisor, coach and