An Abstract Window Toolkit Visualizer for Computer Science Instruction

Josiah Dykstra
Department of Computer Science
Hope College
dykstra@cs.hope.edu

Keith Suppes
Department of Computer Science
Alma College
99suppes@alma.edu

Herbert L. Dershem
Department of Computer Science
Hope College
dershem@cs.hope.edu

Abstract

In the teaching and learning of Computer Science, visualization is an important tool. The goal of this project is to provide a tool that will permit students to visualize the execution of a program that includes event-driven interface components. In particular, it is designed to visualize programs written in Java using the Abstract Window Toolkit (AWT).

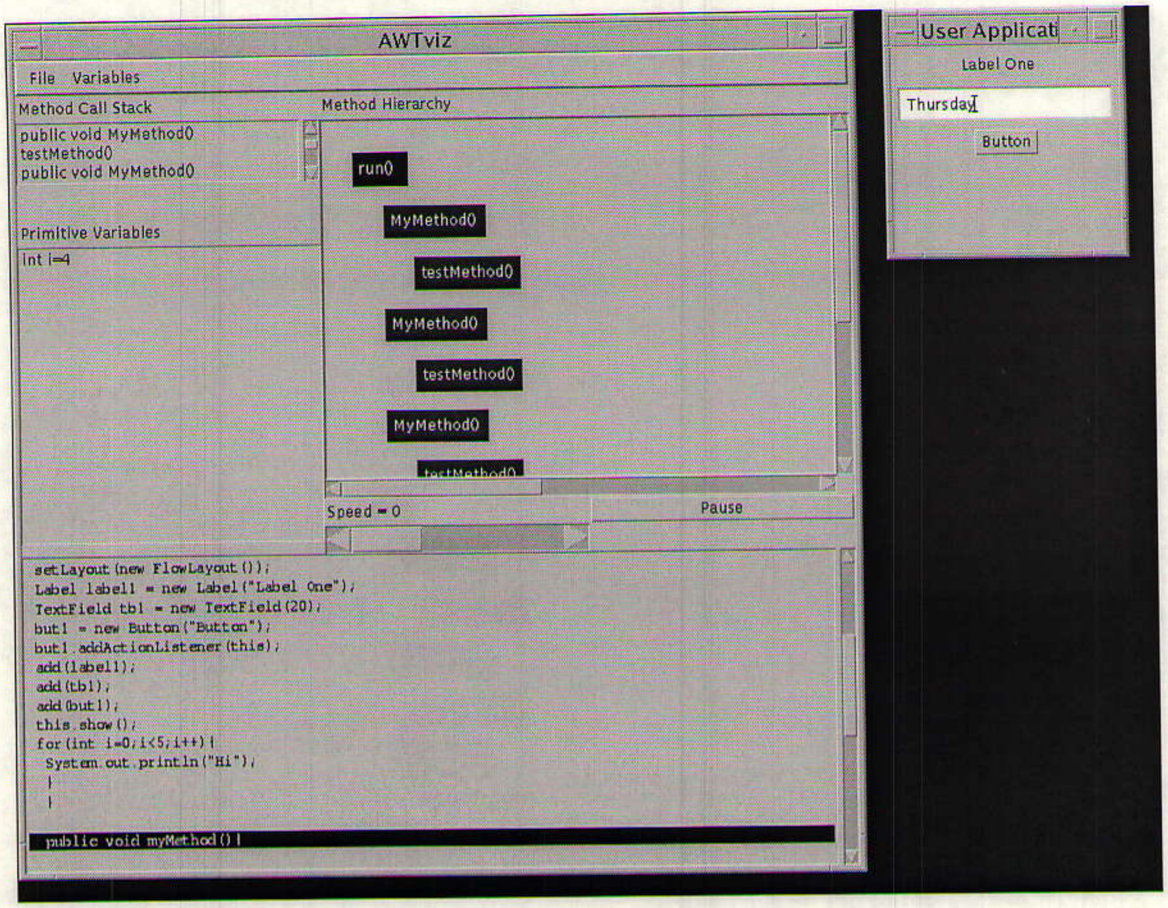
When the program is executed under control of the AWT Visualization software, a visualization window is displayed to monitor execution under user control and the components produced by the program's execution are displayed concurrently. This enables the student to view the code and its corresponding action and to visualize the hierarchy of calls that results.

The paper contains a description of how the software is used and includes details of its implementation. In addition, it provides a complete example that demonstrates how this tool can be used for Computer Science instruction and program debugging.

1.0 Introduction

For the past two years, students in Hope College's Computer Science REU program have worked in the field of Program Execution Animation. In 1997, James Vanderhyde created the Function Visualizer as a tool to teach recursion. This program was revised and greatly expanded in 1998 by D. Erin Parker and Rebecca Weinhold. It was generalized to allow for the visualization of any user-defined function during execution. One of the primary assets of this tool is the manner in which it helps to visualize the function call stack. Also during the summer of 1997, James Vanderhyde created the Object Visualizer, a powerful tool for deconstructing a class file into its methods and the objects that those methods return. This allows for an easy visualization of how objects interact between different methods and classes. We expanded these two programs, adding further refinements to both the user interfaces and the functionality of each.

In addition to the enhancement of these existing tools, we created our own visualization program called the Abstract Window Toolkit Visualizer (AWTviz). This program was created for the purpose of visualizing the execution of an entire program. In particular, we wanted to visualize programs that incorporate Java's Abstract Window Toolkit (AWT). AWTviz provides two views, the Visualization Window and User Application Window. The Visualization Window contains AWTviz's display and control components and the User Application Window contains the output of the user's running program.



```
public void MyMethod()
testMethod()
public void MyMethod()
```

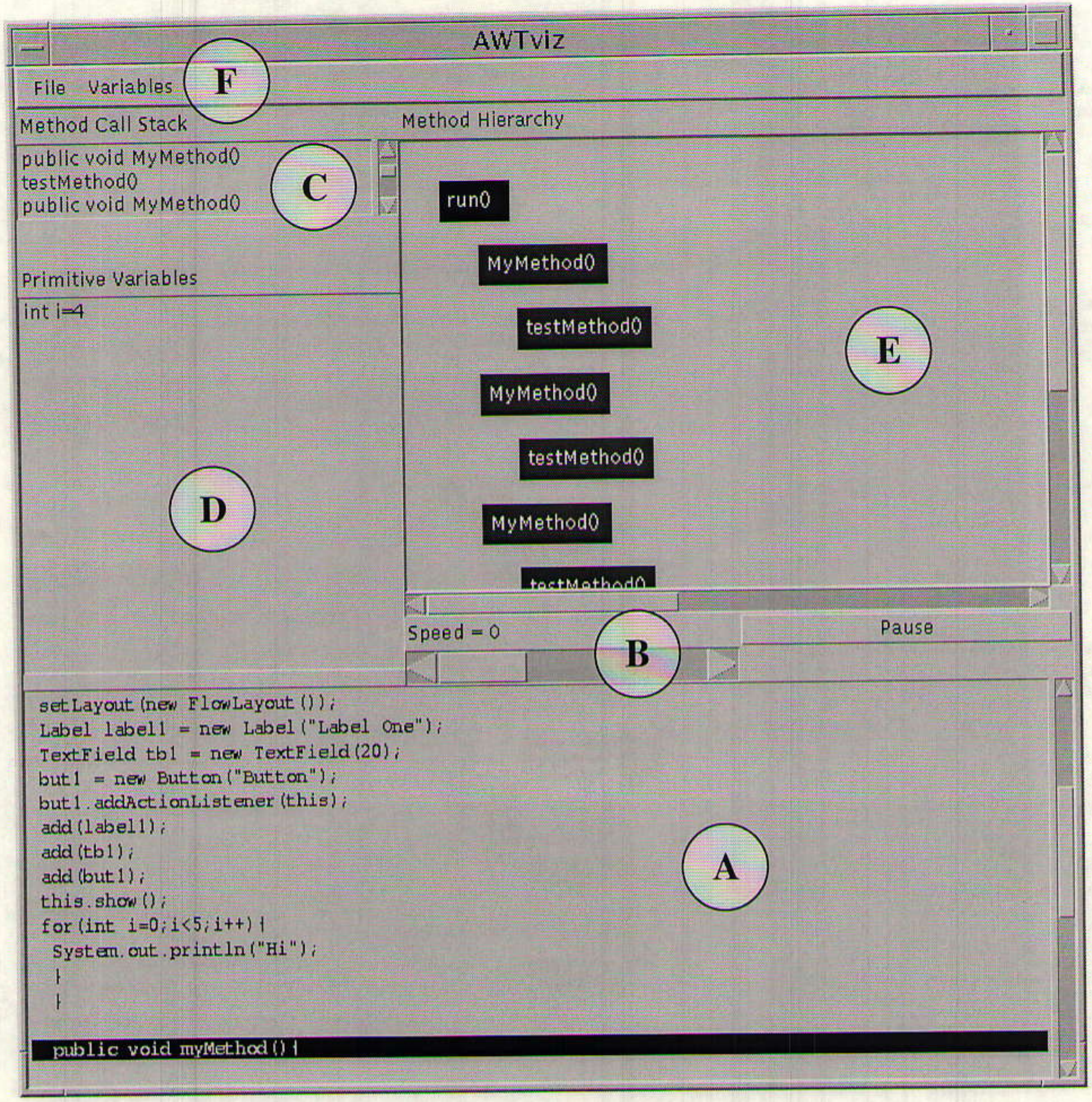
```
int i=4
```

```
setLayout(new FlowLayout());
Label label1 = new Label("Label One");
TextField tb1 = new TextField(20);
but1 = new Button("Button");
but1.addActionListener(this);
add(label1);
add(tb1);
add(but1);
this.show();
for(int i=0;i<5;i++){
 System.out.println("Hi");
 }
 }
public void myMethod(){
```

Figure 1: Visualization Window and User Application Windows for test.

2.0 Visualization Window Components

The Visualization Window contains six key display and control components. The display gives the user information about what is happening within their program. The control functions allow the user to adjust the speed of visualization and some of what is being visualized.



```
setLayout(new FlowLayout());
Label label1 = new Label("Label One");
TextField tb1 = new TextField(20);
but1 = new Button("Button");
but1.addActionListener(this);
add(label1);
add(tb1);
add(but1);
this.show();
for(int i=0;i<5;i++){
 System.out.println("Hi");
 }
 }
public void myMethod(){
```

Figure 2: Visualization Window for test.

2.1 Code View (Figure 2 – A)

The code view portion of the Visualization Window displays the code of the user-defined program. As each line of code is executed, that line is highlighted.

2.2 Speed Control (Figure 2 – B)

The speed control consists of two parts, a scrollbar and a button. The scrollbar allows the speed of the user's application to be increased or decreased during execution. The button pauses and resumes the execution.

2.3 Call Stack (Figure 2 – C)

The Call Stack is a listing of the names of each currently active method called by the user's application sequentially displayed.

2.4 Variable Display (Figure 2 – D)

The Variable Display is another listing that displays the variables in the user's program. As the values of the variables change, these changes are reflected in the display.

2.5 Method Hierarchy (Figure 2 – E)

The Method Hierarchy is a scrollable area upon which colored boxes are drawn for each method of the user's application. The boxes are drawn as the methods are called and appear in a hierarchy, with sub-calls being indented further to the right than the method that called them. The Method Hierarchy canvas will expand when it runs out of room to display more boxes.

2.6 Menus (Figure 2 – F)

File

- Restart - This option causes the program to start again, with all values reinitialized.
- Quit - This option will quit the program.

Variables

- <depends on the variables in the user-defined program> - each variable can be selected or deselected in this menu. When selected, a variable will appear in the variable display. When deselected, it will not appear in the display.

3.0 Using the Program

There are two phases that a user-provided program must go through in order to be visualized. The program must first be processed by a set of Perl scripts to correctly format it prior to passing the formatted code to the Java application for visualization.

```
import java.awt.event.*;
import java.awt.*;

public class test extends Frame implements ActionListener{

 Button but1;

 public void init(){
        myMethod();
        setSize(200,200);
        setLayout(new FlowLayout());
        Label label1 = new Label("Label One");
        TextField tb1 = new TextField(20);
        but1 = new Button("Button");
        but1.addActionListener(this);
        add(label1);
        add(tb1);
        add(but1);
        this.show();
        for(int i=0;i<5;i++){
                System.out.println("Hi");
                }
}

public void myMethod(){
        testMethod();
}

public void testMethod(){
}

public void actionPerformed(ActionEvent e){
        if(e.getSource() instanceof Button)
                {
                if(e.getActionCommand().equals("Button"))
                        {
                        but1.setBackground(Color.black);
                        }
                }
        }
}
```
Figure 3: Java code for test.

3.1 Specifications/ Limitations

One of the goals of this project was to design a tool that would be useful on a wide variety of problems and situations. While much work was done to ensure this, a number of unavoidable limitations still exist. In order to run the program the user must have a local copy of the AWTviz program, the Java Development Kit (JDK) 1.1 or higher and Perl installed, and provide their own Java program for Visualization. When writing the Java program, the user must abide by the following rules. First, the user-provided program must extend java.awt.Frame. Second, the user may not have any primitive variables that end with an underscore. Third, there may not be more than 99 primitive variables within the program. Fourth, primitive variables of different types may not have the same name (ie. int alpha and char alpha are not permitted, however int alpha and char Alpha are acceptable). Fifth, users should not have a method named run() anywhere in their program.

3.2 Pre-processing

The test program, as written by the user, has no way of communicating or interacting with AWTviz. In order to make this possible, a set of Perl scripts must modify the user-provided code. The resulting Java file is an exact version of the user-provided code along with a number of AWTviz methods.

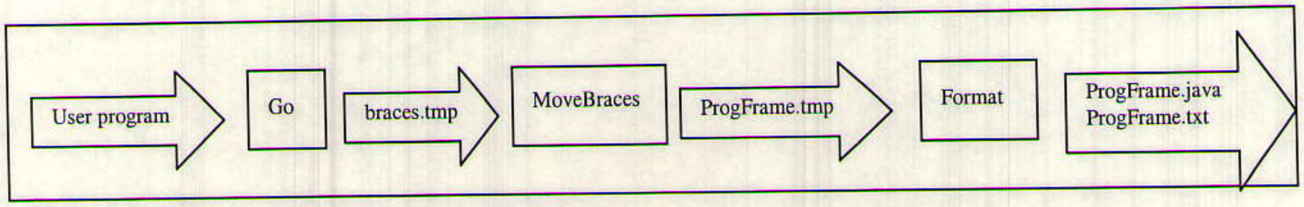


Figure 4: Path of test input code through pre-processing.

3.3 Visualization

When the pre-processing is complete and the resulting files are created the Java application is executed. The Java program reads in the class and text files and uses them to determine the visualization. Utilizing the controls in AWTviz, the user can control various aspects of the program. The user can also interact with his or her own program from within it's own window.

4.0 Code Structure

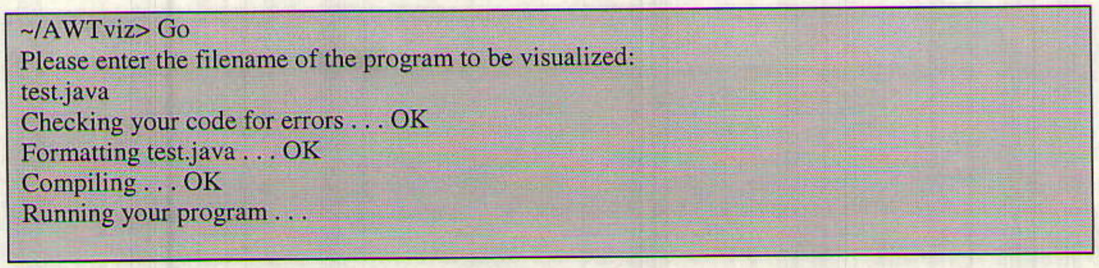
Coding for AWTviz was done in Java and Perl. The full program consists of three Perl scripts and six Java classes. All files reside and are executed from the AWTviz directory. Figure 4 shows the movement of input through the various code sections.

4.1 Perl

The Go script is what the user executes from the command line to begin the program. MoveBraces is a script that quickly formats the curly braces in the user's program. This is for AWTviz's use only and has no effect on the original program copy. Format is the primary Perl script that adds the necessary code to the user's program allowing it to interact with AWTviz. (see Figure 4)

4.1.1 Go

Go is a simple script that calls each subsequent script and the Java application. Go first ensures that the user's input file exists and is valid by compiling it. (see Figure 5)



```
~/AWTviz> Go
Please enter the filename of the program to be visualized:
test.java
Checking your code for errors . . . OK
Formatting test.java . . . OK
Compiling . . . OK
Running your program . . .
```

Program Initialized

Figure 5: Running AWTviz with Go.

4.1.2 MoveBraces

MoveBraces is a short script that handles the formatting of curly braces within the inputted program code. When the script finds an open curly brace on a line by itself, it places it on the end of the previous line. If it finds a closed curly brace on at the end of a line following other code, it places it on the next line by itself. This is done to make the job of the Format script easier and more efficient.

4.1.3 Format

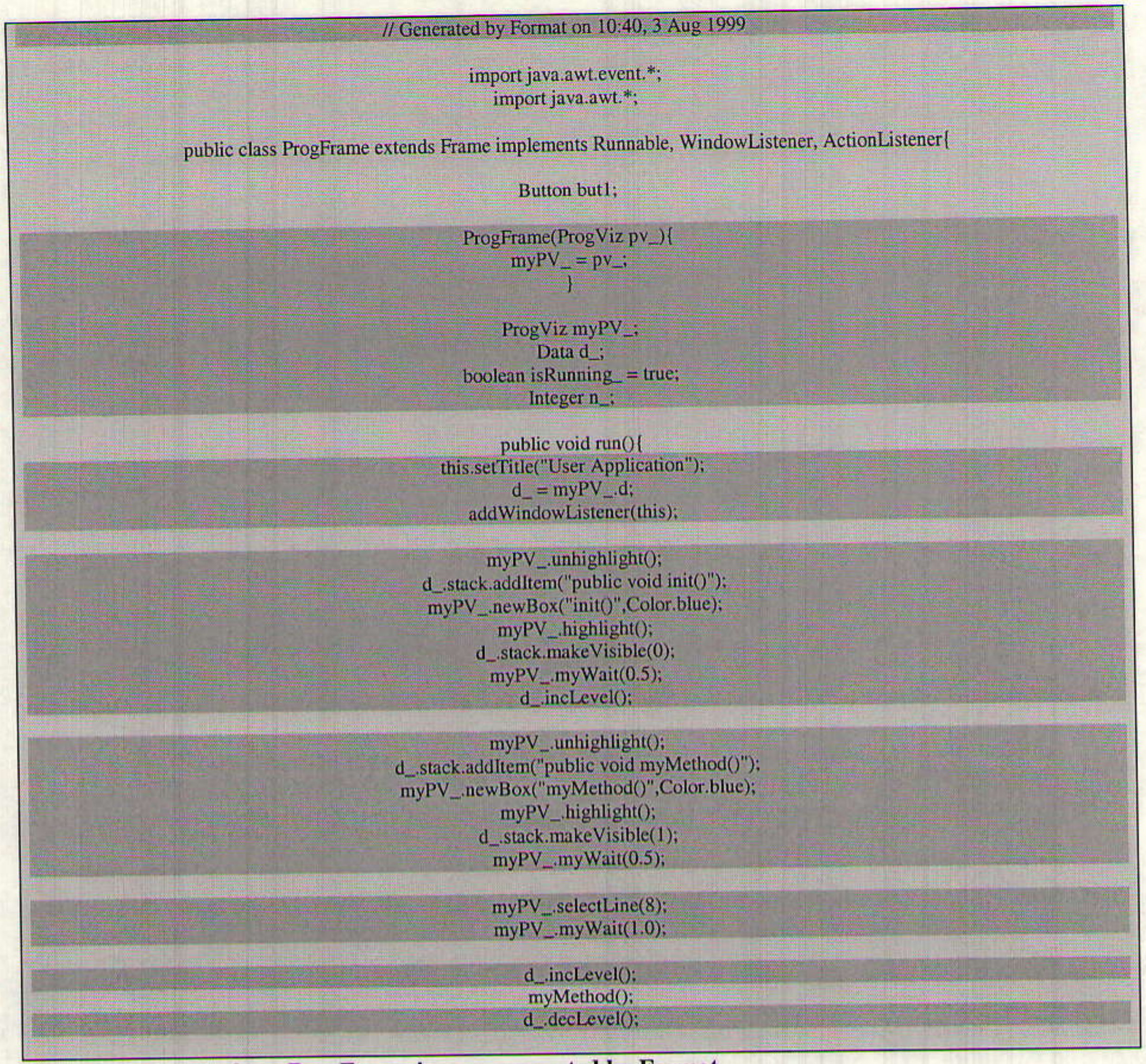
Format does most of the work in readying the user's program for use with AWTviz. It uses pattern matching to add instructions for AWTviz and generates a file called ProgFrame.java.

On the first pass through the code,

1. Format locates and places in an array the names of all user-defined methods.
2. It also ensures that the necessary Java import statements are present.

On the second pass,

1. Format reads each line of code and interprets it.
2. The script will print to the output file the original line as well as necessary code before and after the line.
3. This part of the script checks to make sure the user's file extends java.awt.Frame and contains the necessary implement statements.
4. It changes the name of the class from the user's name to ProgFrame.
5. Special instructions to link the Visualization Window and User Application Window together are added to the init() method, in addition to renaming it from init to run.
6. If an actionPerformed() method exists, code specific to that is inserted.
7. Every other line in the program is checked to see if it contains the name of a user-defined method or variable declaration or assignment.
8. If the line contains a user-defined method, lines are added to tell AWTviz to increment the level of the Method Hierarchy, to add the method to the Call Stack, to create a new method box, to highlight that box, to pause long enough for these actions to complete, and to decrement the level of the Method Hierarchy.
9. If the line contains a primitive type variable declaration, a line is added that tells AWTviz to handle the addition of that variable to all necessary lists, arrays, and hashtables.
10. If the line contains a variable assignment, three lines are added that retrieve the current value, adjust the display in AWTviz's Variable List, and updates the value in AWTviz.
11. For every non-curly brace line after init() in the user's program, two lines are added. These instruct AWTviz to highlight the current line in Code View and to pause relative to the Speed Control.



```
// Generated by Format on 10:40, 3 Aug 1999

import java.awt.event.*;
import java.awt.*;

public class ProgFrame extends Frame implements Runnable, WindowListener, ActionListener{

Button but1;

ProgFrame(ProgViz pv_){
myPV_ = pv_;
}

ProgViz myPV_;
Data d_;
boolean isRunning_ = true;
Integer n_;

public void run(){
this.setTitle("User Application");
d_ = myPV_.d;
addWindowListener(this);

myPV_.unhighlight();
d_.stack.addItem("public void init()");
myPV_.newBox("init()",Color.blue);
myPV_.highlight();
d_.stack.makeVisible(0);
myPV_.myWait(0.5);
d_.incLevel();

myPV_.unhighlight();
d_.stack.addItem("public void myMethod()");
myPV_.newBox("myMethod()",Color.blue);
myPV_.highlight();
d_.stack.makeVisible(1);
myPV_.myWait(0.5);

myPV_.selectLine(8);
myPV_.myWait(1.0);

d_.incLevel();
myMethod();
d_.decLevel();
```

Figure 6: Excerpt from ProgFrame.java as generated by Format.

4.2 Java

4.2.1 AWTviz

AWTviz implements the AWT Visualizer application. It creates an instance of ProgViz and calls the init() method.

4.2.2 ProgViz

ProgViz is the primary class for the visualization containing all the components mentioned in section 2. The class consists of the following methods:

init() - performs the layout of all the various components of the Visualization Window and sets initial values for much of the visualizer.

getCode() - gets the original code from the user-defined program, places it in the code list, and displays it in Code View.

selectLine() - selects the proper line in the code list while the program is executing.

myWait() - causes the thread running the user defined program to pause just long enough for the code selection to be in sync with the actual execution.

actionPerformed() - handles AWT actions for this object. This method processes events from the menus and pause/resume button in the Visualization Window.

adjustmentValueChanged() - handles the adjustment of the speed slider.

handleRestart() - does all the things necessary to restart the program.

addVariable() - adds a variable into the variable list and displays it in the Visualization Window.

itemStateChanged() - toggles whether a variable is visible in the variable list or not.

expandCanvas() - makes a larger MyCanvas object (see 4.2.4) and replaces the old one when there isn't any more room to draw a method box.

unhighlight() - calls unhighlight in MyCanvas.

highlight() - calls highlight in MyCanvas.

newBox() - creates a new box for any user-defined method when it is called, stores the information for it in an instance of Data (see 4.2.3) and then draws the box on the current instance of MyCanvas. If the canvas is too small, expandCanvas() is called.

4.2.3 Data

Data is a class made up of data structures that store information about the method boxes that are displayed in the current instance of MyCanvas. Since the MyCanvas object will likely need to be disposed of and a larger one put in its place, this class exists so that the data about the contents of the canvas may persist. There are two methods in this class:

incLevel() - increments the int variable level by one.

decLevel() - decrements the int variable level by one.

4.2.4 MyCanvas

MyCanvas is the canvas upon which boxes are drawn in a hierarchical format.

paint() - cycles through the array of method boxes, painting each one as it goes.

highlight() - highlights a method box if the program is currently executing that method.

unhighlight() - repaints all method boxes, with none of them highlighted.

4.2.5 MyPanel

MyPanel is an extension of java.awt.Panel that allows a specific size for the panel to be set by calling setPreferredSize().

4.2.6 ProgFrame

ProgFrame is the user created program, with some additional code to aid in the visualization.

5.0 Conclusions and Future Work

When looking at the topic of visualization, we saw that tools already existed to visualize objects and functions. The next logical step seemed to be a visualization of programs. The culmination of this summer's research is a tool which can successfully visualize a wide range of user-defined programs. Our program will format the input through a set of Perl scripts and then display both the user's program and the AWTviz control. The AWTviz program will display and track variables, follow the lines of code, and display in a graphical manner how method calls are made. The user has control over the speed of execution and which variables are shown.

While our intention for AWTviz was as an educational tool, it has the ability to be applied to many different situations. The primary purpose of this program is to visualize how the Java Abstract Window Toolkit handles program execution. The user can very clearly see how the often complex structure of method calls are made. A program with recursion, for example, is much more easily understood by seeing the parent/ child relationship in visual method calls. The ability to track variable assignment in a visual way may aid students who are confused as to when and where assignments are made. Finally, control over the speed of program execution allows for quicker comprehension of the event processing within one's own program. The academic setting is the most appropriate place for AWTviz, however it could easily be adapted for software development and product testing.

There are additional improvements that could be made to further the AWTviz program. However, because of time constraints, we were unable to accomplish everything. AWTviz would be an even more powerful tool if it could visualize non-frame Java applications and Java classes. The ability to dynamically support any number of variables in a user's program would also add to usability. The idea also arose to color code the method boxes displayed in the Method Hierarchy according to the level of indentation.

6.0 Acknowledgments

We would like to acknowledge the National Science Foundation for their financial support of the REU program. We would also like to thank Hope College and our advisor Dr. Herbert Dershem for their hospitality and continued support of computer science and REU.