Mindanao Journal of Science and Technology Vol.10 (2012) 63-80

A Program Visualization Approach in Developing an Interactive Simulation of Java Programs for Novice Programmers

Aurora Cindy G. Agno-Balabat[1*] and Jay Noel N. Rojo[2]
[1]College of Information Technology
Liceo de Cagayan University
Rodolfo N. Pelaez Boulevard, Cagayan de Oro City, 9000 Philippines
**cindy_agno@yahoo.com*

[2]College of Industrial and Information Technology
Mindanao University of Science and Technology
CM Recto Ave., Lapasan, Cagayan de Oro City, 9000 Philippines

Date received: July 17, 2012
Revision accepted: November 03, 2012

Abstract

It is widely agreed that learning to program is extremely difficult. Beginning programmers tend to have serious difficulties in grasping the abstract concepts and notations that programming involves.To become an expert in programming, it requires a deliberate practice and the ability to comprehend a computer program, so to establish a valid mental presentation of the problem solved by the program. Because of the lack of knowledge and experience, novice programmers have problems with constructing the viable models of problems.

In this study, program visualization was designed meant especially to aid novice programmers in Java language. It visualizes the data and control flow of the program. The program visualization design used a modular approach that permits both internal and external extensibility, which consist of two systems, a visualization engine and a Java source interpreter.

Keywords: interactive simulation, program visualization, java programming

1. Introduction

Programming is an exceedingly difficult activity for most beginners. Lecturers have introduced many ways to assist their own lecturing and to help students to acquire new knowledge in these fields. The learning aids have varied from paper, pen and scissors to the static images on the slides. However, it appears most natural to illustrate the program's execution over time with an animation. Thus, for the last couple of decades the software visualization has been an active field of study (Baecker, 1981; Hundhausen *et al.,* 2002).

When considering the possibility to produce a visualization of a program, it may appear that the visualization is a superior way to illustrate their behaviour. Software is commonly visualized in an attempt to facilitate program comprehension and support software engineering activities.

Studies have shown that a number of visualization software have been developed in previous years. One of the first and most renowned systems to teach introductory object-oriented programming is BlueJ (Kölling *et al.*, 2003; BlueJ, 2003). The key feature of the system is the static visualization of the class structure as a Unified Modeling Language (UML) diagram. Furthermore, it allows the learner to interact with the objects by creating them, calling their methods and inspecting their state with the easy use menus and dialogs. However, it does not provide any dynamic visualizations of the program, which is the purpose of this study.

There are two versions of the same system, namely Javavis (Oechsle and Schmitt, 2002) and Jvisual (Birkheim, 2002), developed on top of the Java Debugging Interface (JDI) to obtain information about the runtime behaviour of the program. They visualize the state of the program and its changes during execution. These systems are not meant for real novices, because the visualization they produce expects the students are familiar with UML and the basics of programming. However, this kind of system could be very useful for advanced courses in programming.

JIVE-Java Interactive Visualization Environment (Gestwicki and Jayaraman, 2002), features interactive visualization; query based debugging and reverse stepping. The program execution is visualized by means of sequence diagrams which display calls from method to method and using object diagrams which display the “use” relations between objects and class hierarchy as contours. The visualization uses data gathered at run-time. But the code itself is not visualized, and neither is the data visualized.

Jeliot 2000 (Ben-Bassat Levy *et al.*, 2003) is a stand-alone Java application, which has a simplified user interface. The animation show both the control and data flow. The design is based on the idea that visualization of the program is actually a consequence or a side-effect of the interpretation of the program. This means that the program is no longer first compiled to annotated source code and then to a program, but is directly interpreted with a Java interpreter. However, it only supports a relatively small subset of Java language and does not support object-oriented programming.

iC++ (Ang and Sioson, 2009) , developed a visualization tool to help novice programmers in understanding variable declarations, assignment statements and the three basic program structure namely : sequence, selection, and repetition. However, the systems only visualize the state of the memory variables in the computer's main memory as each C++ statement in a program fragment executes.

There are also several pedagogical environments that help the novice students to overcome problems in compiling and debugging, the software Dr.Java (Allen *et. al.*, 2002; Stoler, 2002) is one of them. It is not a software visualization tool; however it is mentioned here due to the usage of DynamicJava, a Java source interpreter which is also used in this study. It is elegantly employing the features of DymanicJava to help students to interact with self-written classes by creating objects and executing separate statements on them easily. The read-evaluate-print loop is introduced into Java teaching, which means that students do not have to write complete program before they can test the programs but they can evaluate each line of code separately. Because of the integrated interpreter, students do not have to compile the program with standard Java compiler that could introduce different kinds of problems.

Some of them share a similar kind of architecture and used an approach that the user code was translated to the closest programming language (i.e. Eliot-C was transformed to C++ and EJava to Java) and then compiled with a standard compiler. This made the framework stable and the visualization of programs semiautomatic but restricted the abstraction level of the visualization by only visualizing the variables of the programs. There were two problems in the design. Firstly, the interpreter and the visualization engine were strongly coupled, which meant that modifications to the visualization of the programs would lead to modifications in the interpreter. Secondly, the used interpreter was hand-crafted and its further development would have taken much effort.

The study aims to build an engaging interactive learning tool for novice programmers at that focuses on unit-level Java programming.

Thereby, this study focuses on the development of a system that involves the new kind of approach in visualization. The following are the specific objectives of the study: (1) to design visualization software in a modular approach which means that an independent interpreter will be used and a separate visual engine as well; (2) to implement the visualization software in a prototype that should support the visualization of as large a subset of programs written in Java language as possible. For portability and animation, the system will be implemented with an object-oriented language such as Java 2 Software Development Kit (J2SDK) and Dynamic Java and; (3) to test the prototype's functionalities and verify that the prototype performs and functions correctly according to its design and specifications.

2. Methodology

The program visualization software requires two separate modules: interpretation and visualization of java programs. On one side, an open source Java interpreter that processes the user program. On the other side, a visual engine creates respective animation of the program interpretation. To connect both parts, an intermediate code was used.

The intermediate code would produce the textual representation of a running program, or a program trace. It not only describes the changes in the variables, as a normal Java debugger would do, it also details the operations that produce those changes and keep a history data of it. All this information is required, to animate every step in the execution of a program.

The model of Stratton (2001) is the most similar when the program code and its interpreter are considered the visualization target, the intermediate presentation the program visualization meta-language, intermediate presentation interpreter the mapping declarations and the visualization engine the visual display. The difference is that instead of using a debugger an interpreter is use as the visualization target.

It has also a similar design with Domingue *et al.* (1992) in which the program code is run and history data is collected from it. The system uses the interpreter to run the code and extracts history data in the form of the intermediate presentation. This history data is then visualized with a procedure that differs in the design but in any case has a similar kind of an approach. The main difference is that the system is designed to work only on-line meaning that visualization is done during the interpretation, whereas in Domingue *et al.* designed seems to visualize the program history data after the whole program is executed making it more post-mortem. With the similarities been discussed above, the model can be thought as a hybrid of the previous models of Domingue *et al.* (1992) and Stratton (2001).

The model consists of various modules that will eventually interact together to provide the specific functionality of the entire system. The model structure of the program visualization software is shown in Figure 1.

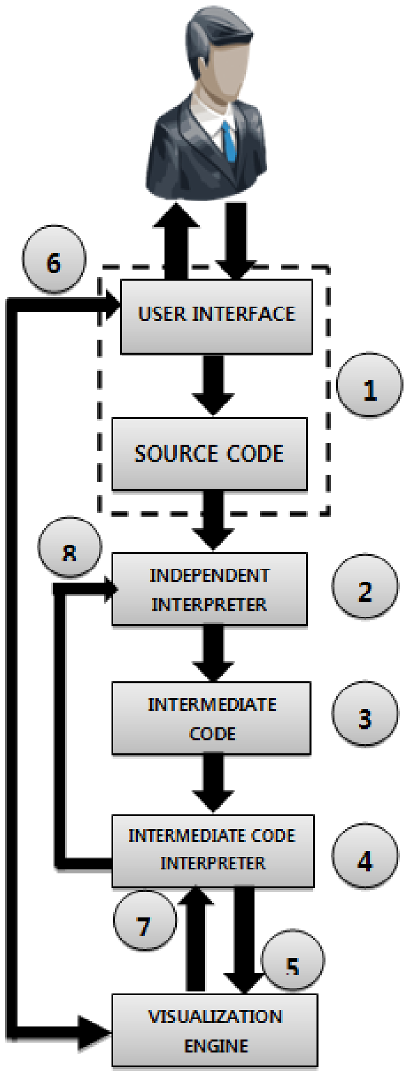


Figure 1.Model of the interactive program visualization system

The program visualization software model has the following process: (1) User Interface and Source Code. The user interacts with the user interface and creates the source code of the program. (2) Interpreter - The source code is sent to the interpreter to process the program for interpretation. (3) Intermediate Code - The intermediate code is extracted during the execution of the code. (4, 5) Intermediate Code Interpreter - The intermediate code is interpreted and the directions are given to the visualization engine. (6-8) Furthermore, the user can give input data, for example, an integer or a string, to the program executed by the interpreter. Figure 2 represents the algorithm used in the model upon the visualization of a java program.

```
1.  Select , Write or Edit java source code in the editor
2.  The code is sent to the interpreter upon compiling.
    2.1. If possible error is encountered
         2.1.1. Messages are shown by the user interface upon start of the
                visualization.
         2.1.2. Go back to step 1
    2.2. If there are no errors, the evaluation of the intermediate code is started.
3.  Visualize source code in the visualization view area.
    3.1. Request highlighting of source code.
    3.2. Request variable creation and display in the visualization area.
         3.2.1. Request variable image creator to create a variable image.
                3.2.1.1. Calls the constructors of the variable image
                3.2.1.2. Return the variable image
         3.2.2. Display the variable image in the visualization area.
                3.2.2.1. Ask for the location of the variable image in the current
                         method stage
                3.2.2.2. Returns the coordinates of the location on the method stage
    3.3. Request the variable image declaration
    3.4. Returns the variable image declaration
    3.5. Request to show the variable image declaration visualization
    3.6. Returns control as soon as the visualization has finished
    3.7. Binds the variable image to the current method stage.
4.  Returns the control back.
5.  End
```

Figure 2. Algorithm of the program visualization

The program visualization prototype is design to supports the following: (a) Highlight active code such as statements and expressions, (b) Display line numbering, (c) Run until line features, (d) Data Types –Numbers, Boolean, Character, String, 1-dimensional Array and Object , (e) Control Flow - Conditional statement , Loops, Static method calls, Object method calls and Recursion (f) Operators - unary operators (unary plus, unary minus, and not) and binary operators. Other constructs not mention above is not supported.

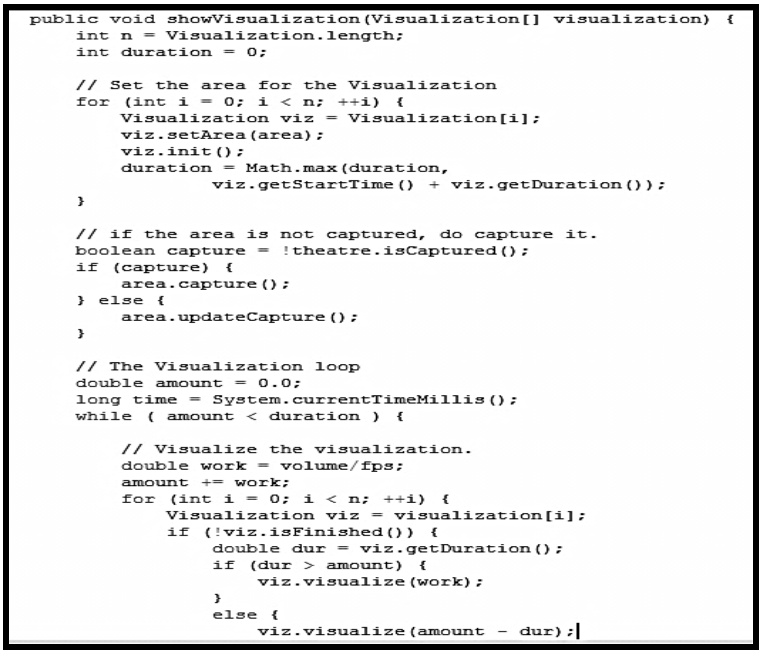
In the implementation of the design of the system, to develop interactive program visualization software, as the ultimate output of this investigation, it uses the open-source Java programming language.

Since extensibility was raised as one of the key design issues, the program visualization prototype would consist of two systems, a visualization engine and a Java source interpreter. DynamicJava is used as the interpreter to extract run-time information which is then converted into a representation of the program's execution in an assembly-like language specially developed for describing the visualizations.

The program visualization software is prototyped in JAVA programming language using the IDE Eclipse 3.5.0 – Galileo Edition on at least 800 MHz Processor that runs in Windows XP format of Operating system.

Figure 3 shows a sample JAVA code for the visualization engine.

A series of tests on the program visualization prototype using several different type of java program in introductory java programming subjects was performed.



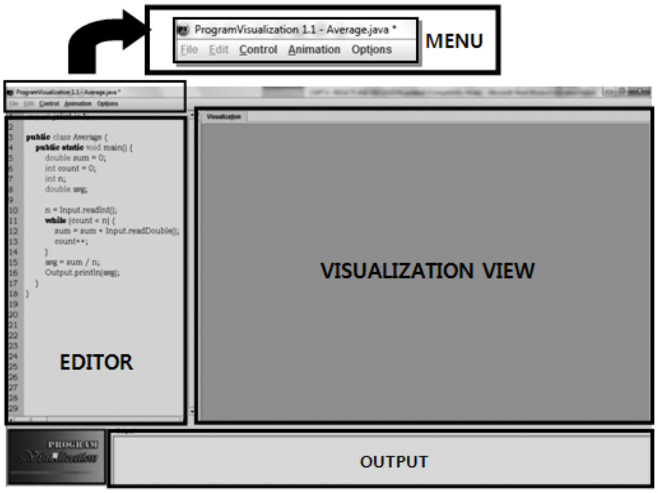
```
public void showVisualization(Visualization[] visualization) {
    int n = Visualization.length;
    int duration = 0;

    // Set the area for the Visualization
    for (int i = 0; i < n; ++i) {
        Visualization viz = Visualization[i];
        viz.setArea(area);
        viz.init();
        duration = Math.max(duration,
                viz.getStartTime() + viz.getDuration());
    }

    // if the area is not captured, do capture it.
    boolean capture = !theatre.isCaptured();
    if (capture) {
        area.capture();
    } else {
        area.updateCapture();
    }

    // The Visualization loop
    double amount = 0.0;
    long time = System.currentTimeMillis();
    while ( amount < duration ) {

        // Visualize the visualization.
        double work = volume/fps;
        amount += work;
        for (int i = 0; i < n; ++i) {
            Visualization viz = visualization[i];
            if (!viz.isFinished()) {
                double dur = viz.getDuration();
                if (dur > amount) {
                    viz.visualize(work);
                }
                else {
                    viz.visualize(amount - dur);
```

Figure 3. Sample JAVA code (visualization engine)

The evaluation of the performance and functionality of the software followed to make sure that it meets up the requirement definition based on the series of tests done. Evaluation of the system was done using a Functional Testing. In such testing method and approach, it verifies if the system executes the functions it is supposed to execute.

3. Results and Discussion

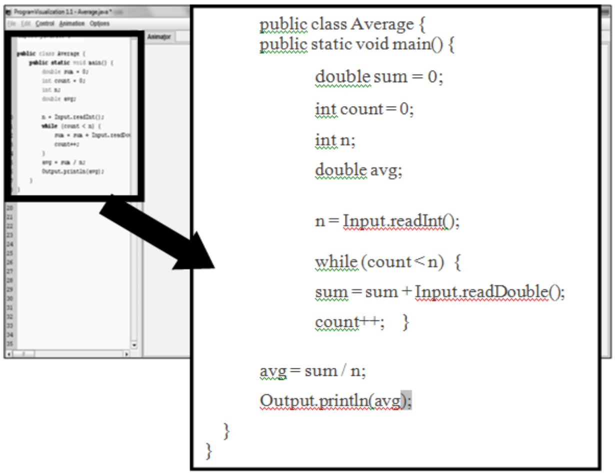
Figure 4 shows the user interface structure of the interactive program visualization software.The interface is being anchored from the Jeliot 2000 interface design which is divided into two main parts. The editing panel consists of the editor bar. The editor can be found below menu. The text editor supports syntax highlighting and brace-matching. The editor is blocked during the visualization. Above the editor is the menu, containing the menus that control the visualization, whether to start it, pause it, stop it, or edit it. The right side of the window contains the animation area, all the visualization occurs there. The visualization area is complemented by a console terminal that will print out the system output of the visualized program.


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
MENU
Visualization
VISUALIZATION VIEW
EDITOR
PROGRAM Visualization
OUTPUT

```
public class Average {
    public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDouble();
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

Figure 4.The user interface structure of the program visualization software

Figure 5 shows a sample of a Java language written in the Editor which will be visualized in the Visualization View.

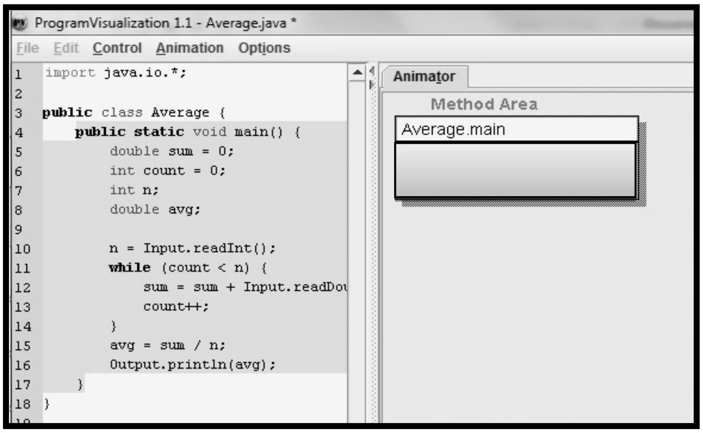


```
public class Average {
public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();

        while (count < n)  {
        sum = sum + Input.readDouble();
        count++;    }

    avg = sum / n;
    Output.println(avg);
  }
}
```

Figure 5. Sample java program written in the editor

The sample program will find and display the average of a set of positive numbers entered by the user. The average is the sum of the numbers inputted, divided by the number of inputs. The program will ask the user to enter one integer at a time as long as the conditional statement returns to true. It will keep count of the numbers entered, and it will keep a running total of all the numbers it has read so far until the conditional statement returns to false.

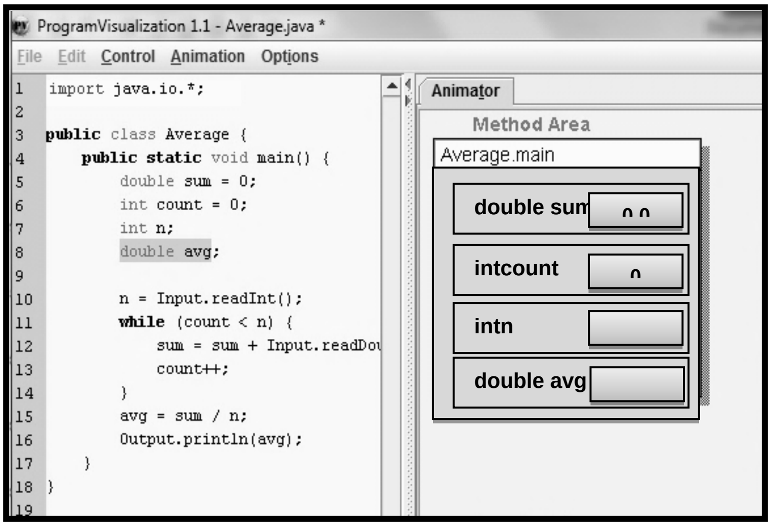
Figures 6 to 14 show fourteen different phases of program visualization of a java program. Figure 6 illustrates the beginning of the class allocation.



```
import java.io.*;

public class Average {
    public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDou
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

Figure 6. Class allocation

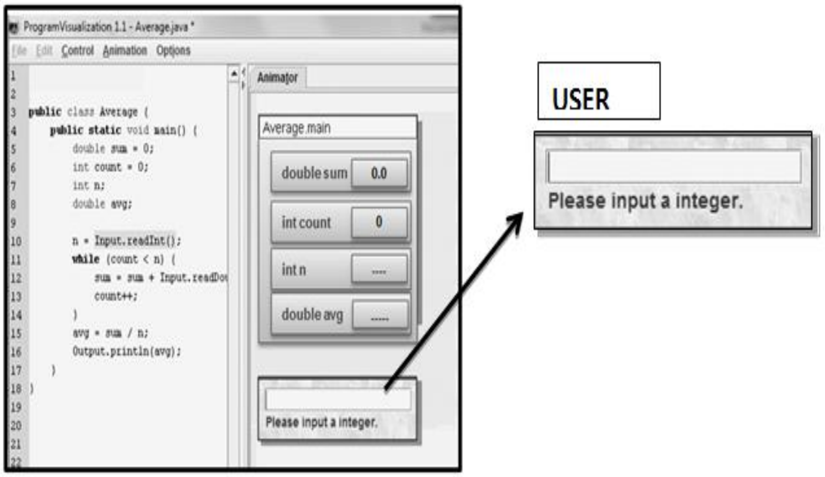
Figure 7 shows the creation of objects and field initialization of *sum, count, n and avg.*


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator
Method Area
Average.main
double sum
0.0
intcount
0
intn
double avg

```
import java.io.*;

public class Average {
    public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDou
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

Figure 7. Created the object *avg*

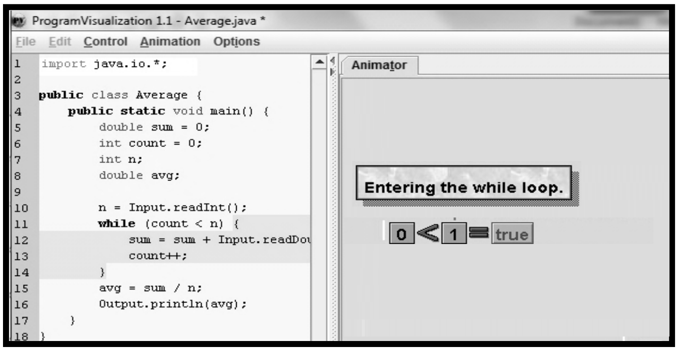
Figure 8 shows the assignment of the users input to field *n*as its current value.



```


public class Average {
    public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDou
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

Figure 8. Users input for field *n* value

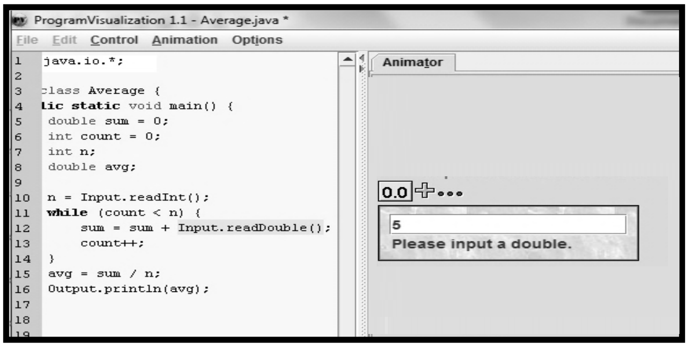
Figures 9 to 11 show the use of control structure in accumulating the users input by *n* times and assigned its value to field *sum*. Figure 12 shows the value of *count* incremented by 1


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator

```
import java.io.*;

public class Average {
    public static void main() {
        double sum = 0;
        int count = 0;
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDou
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

```
Entering the while loop.
0 < 1 = true
```

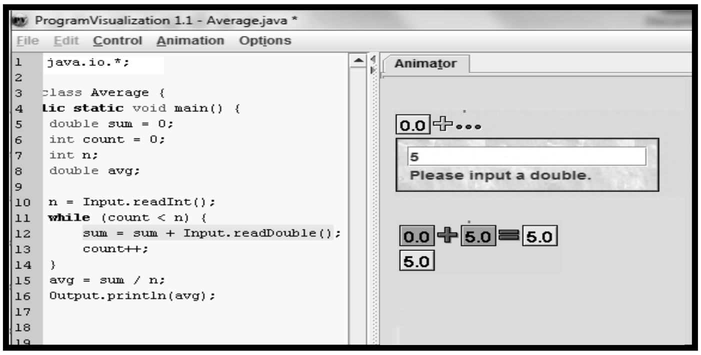
Figure 9. Test while loop statement


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator

```
java.io.*;

class Average {
lic static void main() {
 double sum = 0;
 int count = 0;
 int n;
 double avg;

 n = Input.readInt();
 while (count < n) {
      sum = sum + Input.readDouble();
      count++;
 }
 avg = sum / n;
 Output.println(avg);
```

```
0.0 + ...
5
Please input a double.
```

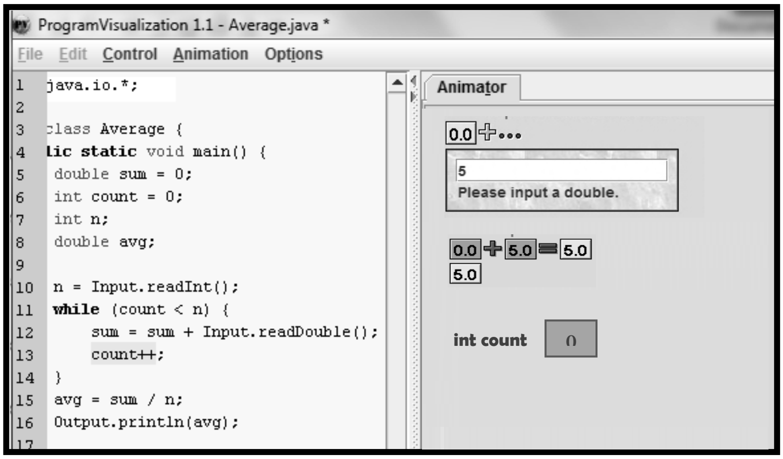
Figure 10. Users input added to the current value of *sum*


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator

```
java.io.*;

class Average {
lic static void main() {
 double sum = 0;
 int count = 0;
 int n;
 double avg;

 n = Input.readInt();
 while (count < n) {
      sum = sum + Input.readDouble();
      count++;
 }
 avg = sum / n;
 Output.println(avg);
```

```
0.0 + ...
5
Please input a double.
0.0 + 5.0 = 5.0
5.0
```

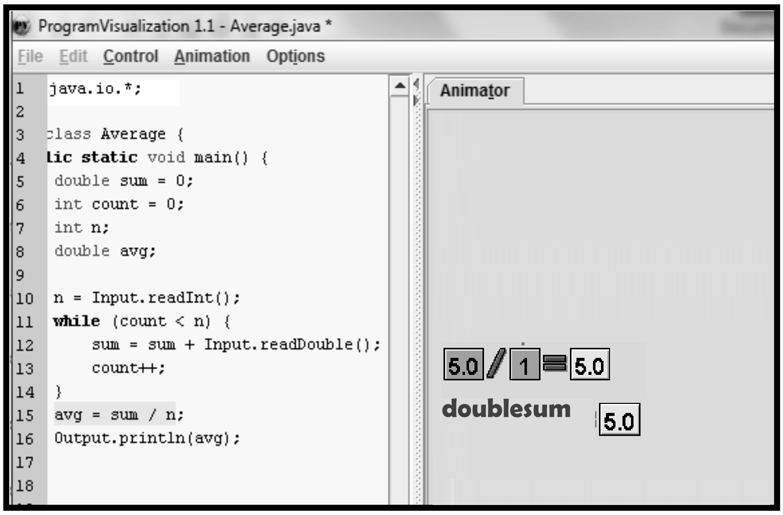
Figure 11. Assigned the value to field *sum*


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator
0.0 + •••
5
Please input a double.
0.0 + 5.0 = 5.0
5.0
int count
0

```
java.io.*;

class Average {
lic static void main() {
 double sum = 0;
 int count = 0;
 int n;
 double avg;

 n = Input.readInt();
 while (count < n) {
     sum = sum + Input.readDouble();
     count++;
 }
 avg = sum / n;
 Output.println(avg);
```

Figure 12. Increment *count* value by 1

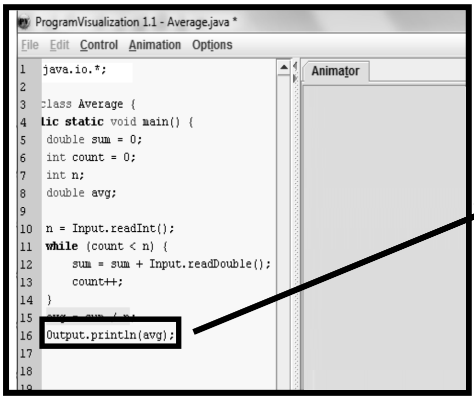
The result is then assigned to field *avg* of the value of *sum* divided by value of *n* which is shown in Figures 13 and 14.


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator
5.0 / 1 = 5.0
doublesum
5.0

```
java.io.*;

class Average {
lic static void main() {
 double sum = 0;
 int count = 0;
 int n;
 double avg;

 n = Input.readInt();
 while (count < n) {
     sum = sum + Input.readDouble();
     count++;
 }
 avg = sum / n;
 Output.println(avg);
```

Figure 13.Assigned to field *avg* of the value of *sum* divided by value of *n*

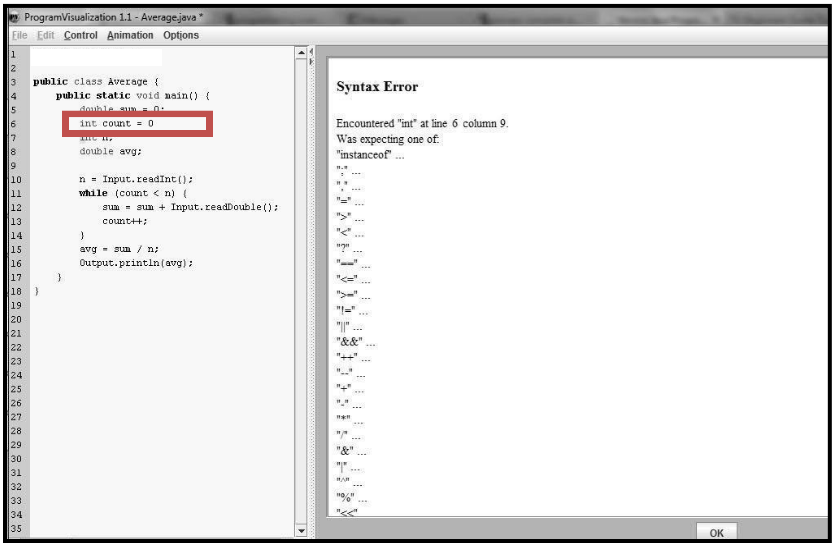
Figure 15 shows an example of a Java program that has a syntax error. Syntax errors are usually typing errors. Misspelled command in Java or


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
Animator
Console

```
java.io.*;

class Average {
lic static void main() {
 double sum = 0;
 int count = 0;
 int n;
 double avg;

 n = Input.readInt();
 while (count < n) {
      sum = sum + Input.readDouble();
      count++;
 }
 avg = sum / n;
 Output.println(avg);
```

```
1
5.0
```

Figure 14.Value of field *avg* displayed in the console window


ProgramVisualization 1.1 - Average.java *
File Edit Control Animation Options
OK

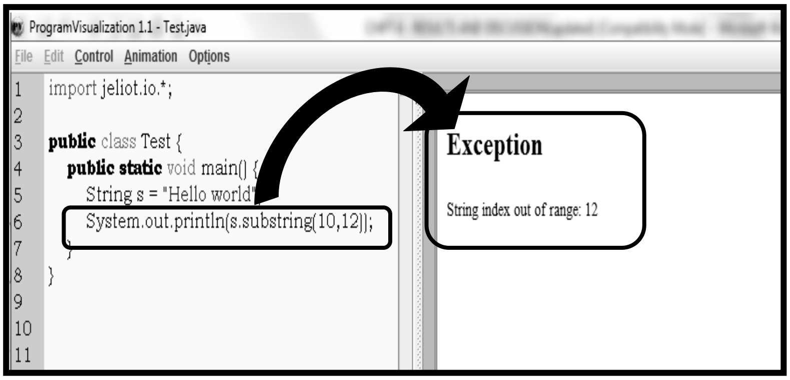
```
public class Average {
    public static void main() {
        double sum = 0;
        int count = 0
        int n;
        double avg;

        n = Input.readInt();
        while (count < n) {
            sum = sum + Input.readDouble();
            count++;
        }
        avg = sum / n;
        Output.println(avg);
    }
}
```

```
Syntax Error

Encountered "int" at line 6 column 9.
Was expecting one of:
"instanceof" ...
";" ...
"," ...
"=" ...
">" ...
"<" ...
"?" ...
"==" ...
"<=" ...
">=" ...
"!=" ...
"||" ...
"&&" ...
"++" ...
"--" ...
"+" ...
"-" ...
"*" ...
"/" ...
"&" ...
"|" ...
"^" ...
"%" ...
"<<"
```

Figure 15. Program visualization of a java program with a syntax error

forgot to write to semi-colon at the end of a statement. Java attempts to isolate the error by displaying the line of code and pointing to the first incorrect character in that line. However, the problem may not be at the exact point. Other common mistakes are capitalization, spelling, the use of incorrect special characters, and omission of correct punctuation. Syntax errors occur upon the compilation of a java program before the visualization. If such kind of error occurs the visualization does not take place until corrected. In the example, shown in the figure, wherein the semicolon was intentionally omitted in one of the statement. The error message displayed in the Visualization View Area suggests that something is expected in line 6, displaying a list of possible missing symbols or characters. According to Ben-Ari (2003), if you encountered a lot of error messages, try to correct the first mistake in a long list and try to complete the program again. Doing so may reduce the total number of errors dramatically.

Figure 16 shows an example of a program that has a run-time error. Run-time errors are errors that will not display until the program will run or execute the program. Even program that compile successfully may display wrong answers if the programmer has not thought through the logical processed and structures of the program. The sample java program is trying to extract a substring of the string *s* but the upper index *12* is not within the string. Executing the program cause an exception to be thrown, thus a message is displayed in the Visualization Area informing the programmer that *String index is out of range: 12.*


ProgramVisualization 1.1 - Test.java
File Edit Control Animation Options

```
import jeliot.io.*;

public class Test {
    public static void main() {
        String s = "Hello world";
        System.out.println(s.substring(10,12));
    }
}
```

Exception

String index out of range: 12

Figure 16. Run-time error

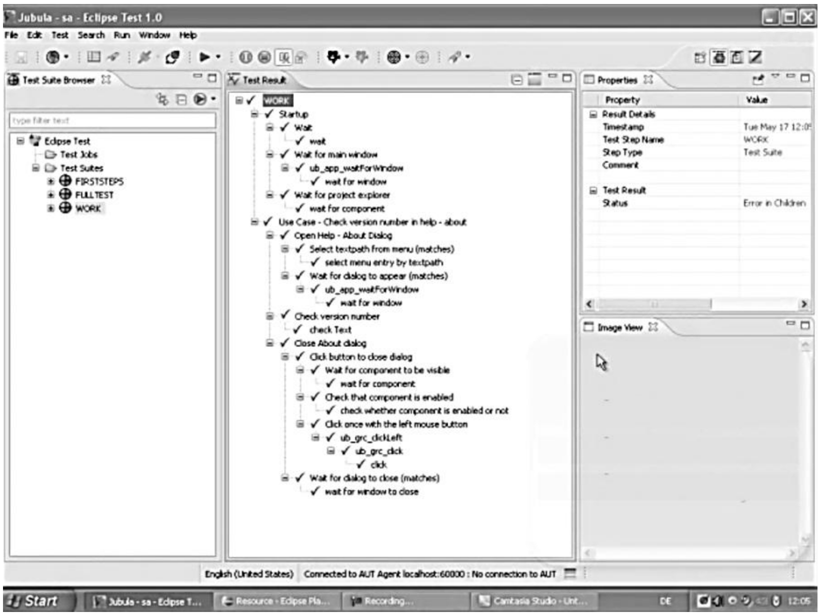
The prototype’s functionalities is tested and verified if it has performed and functions correctly according to its design and specifications using Jubula-Automated Functional Testing Tool.

The Jubula functional testing tool is based on the premise that automated acceptance tests are just as important as the project code, and should adhere to the same best practices (modularity, reusability, and readability) without requiring that any code be produced. This places the power of testing in the hands of the testers and improves accessibility for users who may want to monitor the tests. The code-free approach keeps test maintenance to a minimum and allows acceptance tests to be written from the user perspective.

Test creation in Jubula is achieved using a library of actions which can be combined using drag and drop. This library has been successfully used in diverse projects and already consists of the vast majority of actions necessary.

Using Jubula can be split into three sections: test creation, test execution and test analysis.

The software prototype has undergone the three major processes: test creation, test execution and test analysis. Figure 17 shows the test result of the program visualization prototype. The test result verifies that the prototype performs and functions correctly according to its design and specifications. It is clear that it has satisfied all constraints with no errors encountered during the entire test execution.



```
Eclipse Test
  Test Jobs
  Test Suites
    FIRSTSTEPS
    FULLTEST
    WORK
```

```
✓ WORK
  ✓ Startup
    ✓ Wait
      ✓ wait
    ✓ Wait for main window
      ✓ ub_app_waitForWindow
        ✓ wait for window
    ✓ Wait for project explorer
      ✓ wait for component
  ✓ Use Case - Check version number in help - about
    ✓ Open Help - About Dialog
      ✓ Select textpath from menu (matches)
        ✓ select menu entry by textpath
      ✓ Wait for dialog to appear (matches)
        ✓ ub_app_waitForWindow
          ✓ wait for window
    ✓ Check version number
      ✓ check Text
    ✓ Close About dialog
      ✓ Click button to close dialog
        ✓ Wait for component to be visible
          ✓ wait for component
        ✓ Check that component is enabled
          ✓ check whether component is enabled or not
        ✓ Click once with the left mouse button
          ✓ ub_grc_clickLeft
            ✓ ub_grc_click
              ✓ click
      ✓ Wait for dialog to close (matches)
        ✓ wait for window to close
```

Property	Value
Result Details	
Timestamp	Tue May 17 12:0
Test Step Name	WORK
Step Type	Test Suite
Comment	
Test Result	
Status	Error in Children

```
English (United States)   Connected to AUT Agent localhost:60000 : No connection to AUT
```

Figure 17. Program visualization - Jubula test result analysis

4. Conclusion and Recommendation

In this study, the visualization model and prototype using program visualization, for interactive java programming simulation is proposed. From my review of research on teaching introductory programming, it has been determined the need for an educational tool that specifically targets novices apparent fragile knowledge of elementary programming which manifests as difficulties in tracing and writing even simple programs. I therefore conclude that a tool that tightly integrates programming tasks with visualizations of program execution and allow novices to practice writing code and easily transition to visually tracing it in order to locate programming errors would help address such problem.

Program visualization is a tool with a low barrier to entry that lets effortlessly transition between writing programs and observing program behavior with the help of automatically generated visualizations. Using such tool, novices are better equipped to move on to learning program design and higher level problem solving skills.

For future work, it is suggested to further enhance the present model to improve the efficiency of the program visualization software that would not just cater novice programmers but higher-level as well, catering large subset of Java programs.

Incorporating visual editor of classes such offered by BlueJ would create a helpful tool for novices. Integrating the system into a more adaptive environment would guide novices through their first step in programming. The material would consist of lessons, corresponding exercises, and interactive visualization of the examples with audio features. The environment would be self-adapting in order to cope with the user’s progress. Visualization would be integrated in a modified environment that can give more helpful specific descriptive details to errors encountered. In such enhancements and modifications the shift from novice to expert programmer would be easier to accomplish.

Lastly, a study of the program visualization software’s effectiveness in terms of its intended purpose such as the learning impact of the said software to novice programmers will help spring up many ideas for refinement of its visualizations and user interactions.

5. References

Allen, Eric E., Cartwright Robert and Stroler, Brian (2002). Drjava: A Lightweight Pedagogic Environment for Java. Sigcse, 137-141.

Ang, Marianne P., Sioson, Allan A. (2009) Enriching Programming Instruction using Visualization. Philippine Information Technology Journal. Vol.2, Pp. 31-34.

Baecker, Ronald M. (1998) Sorting Out Sorting: A Case Study of Software Visualization for Teaching Computer Science. Software Visualization: Programming as A Multimedia Experience, 369-381.

Ben-Bassat Levy, Ronit, Ben-Ari, Mordechai, and Uronen, Pauliina(2003). The Jeliot 2000 Program animation System.Computers &Education,40, 15–21.

Birkheim, Alexander (2002) Automatic Visualization of Java Programs to Be Used In Thejava Teaching. Master's Thesis, University of Applied Sciences Cologne,Cologne, German.

Bluej (2003) Bluej — The Interactive Java Environment.Http:// Www.Bluej.Org (Accessed 2004-13-02).

Domingue, John B., Price, Blaine A. A., and Eisenstadt, Marc(1992)A Framework for Describing and Implementing Software Visualization Systems. Proceedings of Graphics Interface '92. Canadian Information Processing Society, 53–60.

Gestwicki, Paul V., And Jayaraman, Bharat. (2002) Methodology and Architecture of Jive. Acm Symposium on Software Visualization.Acm Press.95-104.

Hudson, Scott E., Flannery, Frank, Ananian, C. Scott., Wang, Dan., And Appel, Andrew W. (1999)"Cup Parser Generator For Java", http://Www.Cs.Princeton.Edu/~Appel/Modern/Java/Cup/, (Accessed 2010-18-02).

Kölling, Micheal And Rosenberg, John (2003) ThebluejSystem And Its Pedagogy.Journal Of Computer Science Education,13 (4).

Oechsle, Rainer Andschmitt, Thomas (2002). Javavis: Automatic Program Visualization qith Object and Sequence Diagrams Using the Java Debug Interface (Jdi). In: Diehl, S. (Ed.), Software Visualization.Lecture Notes In Computer Science.Springer-Verlag, Vol. 2269, pp. 176–190.

Stratton, David (2001) A Program Visualisation Meta-Language Proposal.In: Lee,C.(Ed.), Proceedings of the 9th International Conference on Computers in Education Icce/Schoolnet2001.Soeul, South Korea,601–609.