International Journal of Advanced Science and Technology
Vol.115 (2018), pp.149-160
http://dx.doi.org/10.14257/ijast.2018.115.14

A Study on the Effects of Visualization Tools on Debugging Program and Extending Functionality

Woo-Chang Shin

Dept. of Computer Science, SeoKyeong University, 16-1 Jungneung-Dong Sungbuk-Ku Seoul, 136-704, Korea
wcshin@skuniv.ac.kr

Abstract

Nowadays, with the rapid change of the society with IT technology, programming education for students is becoming more important. However, it is difficult and time-consuming for students to master programming skills. In particular, object-oriented languages such as JAVA and C++ language are more difficult to understand, because the execution flow of the program is not intuitive and the program is executed by interactions between objects. In this context, program visualization is an effective tool to show the internal structure and behavior of a program. In the present paper, we introduce an object-interaction visualization method to help educate the object-oriented programming concept and the ObjectVisualizer system, which is a dynamic program visualization tool that implements the method. The results of our experiments also show how the ObjectVisualizer system works in two major tasks of programming: debugging and extending functionality.

Keywords: *Visualization Tools, Software Education, Object Interaction, ObjectVisualizer*

1. Introduction

In the era of the fourth industrial revolution, represented by major new technologies such as artificial intelligence, virtual reality, big data, and drones, the importance of software has increased, and programming education for students has been further expanded and strengthened [6][13]. In the UK, since September 2014, computing subjects have been compulsory for students aged from 5 to 16, and Japan has designated "information" as a required course in high school in 2012. Likewise, India has set computer science as a prerequisite for elementary and junior high schools since 2010, while the United States has included "Computational Thinking" courses in high school AP courses [13][21]. In order to keep pace with this global trend, in Korea, computer subjects will be implemented as compulsory education in elementary and secondary education courses nationwide from 2018 [10][11][13].

While the importance of programming education increases, it is difficult and time-consuming for students to master programming skills. One of the reasons behind this trend is that students do not fully understand the behavioral mechanisms that occur inside the computer when the program is run - such as variable states, stack memory changes, interrupt processes, and memory address references [15].

In general, college students study C language in the lower grades, and then study object-oriented languages such as C++ and JAVA. C language is still one of the most used languages, and object-oriented programming is the main paradigm of current programming language. According to TIOBE, which provides programming language

Received (December 10, 2017), Review Result (January 28, 2018), Accepted (February 2, 2018)

ISSN: 2005-4238 IJAST
Copyright ⓒ 2018 SERSC Australia

share ranking, eight out of the top 10 programming languages, including JAVA that has the largest market share (15.5%) as of April 2017, are object-oriented languages [23].

Procedural languages such as the C language are less difficult to understand and write programs in, as the program's execution flow is intuitive. By contrast, an object-oriented language such as JAVA or C ++ language dynamically changes the execution flow of a program. In addition, students experience more difficulties in learning the language, because they need to understand the objects that make up the program, acquire unique concepts such as Capsulation, Inheritance, and Polymorphism, and understand that the program is performed by interactions between objects. The latter is difficult to understand even for professional developers, as the function of the object called by Polymorphism is determined at runtime.

Program visualization is an effective tool to show the internal structure and behavior of a program. Rajala [18] defines program visualization (PV) as "a research area that is visually assisting learners in understanding behavior programs". PV is divided into Dynamic Program Visualization (DPV) and Static Program Visualization (SPV) [4]. Whereas the DPV tool shows the internal appearance of the dynamic execution of the program, SPV generally analyzes the program source code and provides the static structure or information to the user.

In the present paper, we introduce an object-interaction visualization method to help educate the object-oriented programming concept and the ObjectVisualizer system, which is a dynamic program visualization tool that implements the method. The results of our experiments also show how object interaction visualization methods and tools can be used to perform two major tasks of programming: debugging and extending functionality.

The remainder of the paper is organized as follows.

In Chapter 2, related works are reviewed. In Chapter 3, a system transformation model for visualization is described. Chapter 4 describes the implementation of the visualization system, and Chapter 5 examines the effectiveness of the proposed visualization method for debugging and extending the example program. Chapter 6 summarizes and concludes this paper.

2. Related Work

Many studies have been conducted to improve the effectiveness of object-oriented programming education using visualization methods and tools.

First, there are studies that use educational visual programming language and tools. Lavonen [12] described Empirica Control (EC), a visual programming environment in which students can develop programs, in the form of flowcharts. Through experimentation with 34 students, EC proved to be a useful tool to train programming principles with minimum effort. Furthermore, Jung [8] presented visual programming methodology and curriculum to efficiently educate students on object-oriented concepts using Alice, an educational visual programming language. Next, Scratch, a visual programming language developed by MIT Media Lab, is widely used in basic programming education [5][14][19]. However, while learning object-oriented language or environment that is not used in the industrial field may be useful for elementary and middle school education, it is difficult to apply it to the university education.

Secondly, many studies have been carried out to improve the efficiency of learning by introducing visualization techniques for difficult areas in programming education, such as algorithms and data structures. For instance, Osman [17] introduced a visualized learning environment to help learn the data structure and proved the educational effect of the environment as an experiment. Furthermore, Byrne [2] found that the effectiveness of the visualization method was limited in learning algorithms. That is, visualization is useful for simple algorithms, but ineffective for complex algorithms. Likewise, Kehoe [9] showed that students can easily access the learning materials by reducing the fearful feeling about complex algorithms by showing the operation of the algorithm in animation.

 Copyright ⓒ 2018 SERSC Australia

Thirdly, extensive research is available that seeks to improve students' understanding of program operation and increase learning effect by showing them the internal structure and behavior of the written program using visualization tools. In one of these studies, Tek [22] divided college students into two groups and experimented with the effects of visualization tools. Programming training was performed using SPV (static program visualization) tool in one group and DPV (dynamic program visualization) tool in another group. The results of this study demonstrated that DPV is more effective than SPV. In addition, Ben [1] showed that applying the Jeliot 2000 system to the one-year curriculum, the animation function of the Jeliot system, helps students to learn the abstract concept of programming. Similarly, Moreno [16] showed that the Jeliot 3 system provides help to debug programs, though the results suggested that it is difficult to understand the semantics of program operation through animation.

In this context, the present study seeks to make the following contributions to the field:

(a) It presents a formalized visualization transformation model to show the internal behavior of object-oriented programs in real time.
(b) It implements a dynamic program visualization tool based on the model.
(c) It validates the effectiveness of the tool in debugging and extending functionality.

3. Transformation Model for Visualization

In order to visualize the object interaction, we developed the ObjectVisualizer tool which analyzes the source code of the existing program and automatically generates the source code with the added visualization function (see Figure 1).

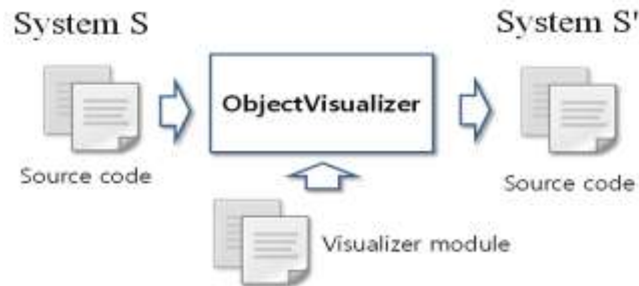


Figure 1. Input and Output of Visualization Transformation

A visualization module is added and a part of the source code is modified to display the interaction of the objects in real time while maintaining the existing program function. The Aspect-Oriented Programming (AOP) technique can be used to pass relevant information to the visualization module at the time the object's method is called or the object is created. However, in the present study, we designed a universal system model to apply the visualization method regardless of the AOP support in a specific programming language. This visualization transformation model was modified and extended based on the results reported by Shin [20]. In order to formalize the visualization transformation process, the object-oriented system model *SYS* was defined as follows.

[Definition 1] The object-oriented system model *SYS* is a structure consisting of the following elements.

$SYS = (CLA, \ll)$

- CLA : a set of classes
- $\ll$: a partial order to represent inheritance relation between classes

$\ll = \{ < c_1, c_2 > | c_1 \in CLA, c_2 \in CLA, c_1 \ inherits \ c_2 \}$

[Definition 2] Class *C* is a tuple constructed as specified below.

$C = < cname, ATTR, FUNC >$
- $cname$: class name
  - $cname \in ID$
- $ATTR$: a set of tuple $< name, sort >$ that represent attributes
  - $name \in ID$
  - $sort \in SORT_{Expr}$
- $FUNC$: a set of tuple $< name, sig, body >$ that represent member functions
  - $name \in ID$
  - $sig = < s_1 \times \ldots \times s_n, s >,\ s_1, \ldots s_n, s \in SORT_{Expr}$
  - $body = < st_1, \ldots, st_n >,\ st_1, \ldots st_n \in STA$
- ID : a set of all identifiers
- $SORT_{Expr}$: a set of all sort expressions
  - $\{boolean, byte, short, int, long, float, double, char\} \cup CLA \cup ENUM \subset SORT_{Expr}$
  - if $s \in SORT_{Expr}$ then $array(s) \in SORT_{Expr}$
  - $ENUM$ is a set of all enumeration type
- STA : a set of all program statements

The system transformation process for object interaction visualization using the system model *SYS* can be expressed by *TSYS*, *TFUNC*, and *TSTMT* functions defined in [Definition 3], [Definition 4], [Definition 5].

[Definition 3] The visualization transformation function *TSYS*, which transforms the system *s* into a visualization system *s'*, is defined as follows.

$TSYS(s) : SYS \rightarrow SYS$
- $TSYS(s) = (CLA', \ll')$ where $s = (CLA, \ll)$
- $CLA' = \{TCLA(c) \mid c \in CLA\} \cup \{v\}$
  - v : visualization class
- $\ll' = \{< TCLA(a), TCLA(b) > \mid < a, b > \in \ll\}$
- $TCLA(c) = < cname, ATTR, FUNC' >$ where $c = < cname, ATTR, FUNC >$
- $FUNC' = \{func' \mid func \in FUNC, func' = TFUNC(func)\}$
  - if there is no constructor in $FUNC$ then $< cname, <>, < entry_st, exit_st > > \in FUNC'$

If no constructor exists in *FUNC*, which is a set of member functions of a class, the default constructor is created in the modified class so that to convey the object creation information to the visualization module.

[Definition 4] The function *TFUNC*, which transforms a member function *f* into a visualization member function *f'*, is defined as follows.

$TFUNC(f) : FUNC \rightarrow FUNC$
- $TFUNC(f) = < name, sig, statements' >$ where $f = < name, sig, statements >$
- $name$: function name
- sig : method signature
- $statements$: a sequence of statements, $= < st_1, \ldots, st_n >$
- $statements'$: a sequence of statements

$$= \begin{cases} < entry_st, st_1', \ldots, st_n', exit_st > & \text{if } st_n \text{ is not a return statment.} \\ < entry_st, st_1', \ldots, st_{n-1}', st_n' > & \text{if } st_n \text{ is a return statment.} \end{cases}$$

  - $entry_st$: a statement that tells the visualization module that a function call has occurred.
  - $exit_st$: a statement that tells the visualization module that the function is to be returned.
  - $st_i' = TSTMT(st_i), 1 \leq i \leq n$

TFUNC adds *entry_st* as the first statement of the member function for visualization in the existing source code and adds *exit_st* as the last statement. When a return statement is in the middle or at the end of a function, *exit_st* is added to the return statement as defined in *TSTMT*. An *entry_st* statement added to the member function notifies the function call information to the visualization module, and an *entry_st* statement added to the constructor transfers the object creation information to the visualization module.

 Copyright ⓒ 2018 SERSC Australia

[Definition 5] The *TSTMT* function to analyze the statements, as well as to modify and insert the code for visualization is defined as follows.

$$TSTMT(st) : STA \rightarrow STA \bigcup \{ < st_1, \ldots, st_n > \mid st_1, \ldots, st_n \in STA \}$$

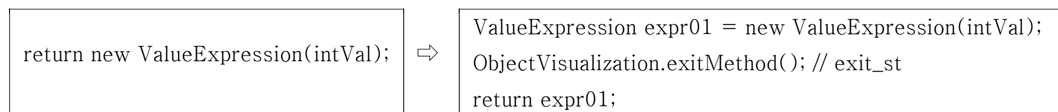
$$= \begin{cases} < exit_st, st > & if\ st\ is\ a\ simple\ return\ statement \\ < save_var_st, exit_st, simplified_st > & \\ & if\ st\ is\ a\ return\ statement\ that\ has\ a\ method-call \\ < st, link_st > & if\ isLinkStatement(st)\ is\ true \\ st & otherwise \end{cases}$$

- $isLinkStatement(st)$: check whether st is a statement that creates a connection with another object

$$= \begin{cases} true & if\ isAssignment(st) = true\ \&\&\ isMemberVar(l-value(st)) = true \\ & \&\&\ isReferenceType(l-value(st)) = true \\ false & otherwise \end{cases}$$

$-isAssignment(st)$: check whether st is a assignment statement
$-l-value(st)$: return the left-value of st statement
$-isMemberVar(var)$: check whether var is a member variable
$-isReferenceType(var)$: check whether var is a reference type

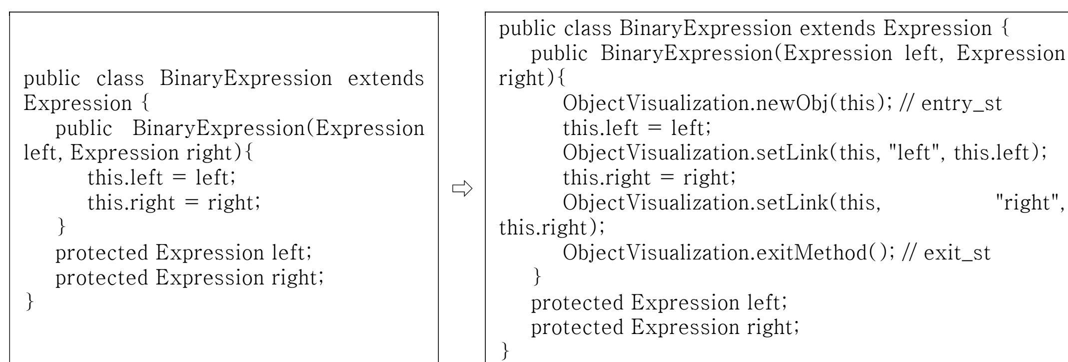
TSTMT reads the statements in the existing source code and adds or changes the code according to some visualization conditions. If the existing statement is a return statement, *TSTMT* adds an *exit_st* statement before the return statement that tells the visualization module that the function is to be returned. If there is a method call or an object creation expression in the return statement, *TSTMT* separates these expressions from the return statement and adds an *exit_st* statement in-between.



```
return new ValueExpression(intVal);
```
⇨
```
ValueExpression expr01 = new ValueExpression(intVal);
ObjectVisualization.exitMethod(); // exit_st
return expr01;
```

Figure 2. Conversion of a Return Statement Containing Method Calls

If the 'association' relationship between objects is generated as a result of executing a statement, it is transmitted to the visualization module. The *isLinkStatement(st)* in [Definition 5] confirms whether the 'association' relation is created. If *st* is an assignment statement that assigns a value to a member variable, and the value is a reference, an 'association' relation between the object of the member variable and the object specified by the reference value is created.



```
public class BinaryExpression extends Expression {
    public BinaryExpression(Expression left, Expression right){
        this.left = left;
        this.right = right;
    }
    protected Expression left;
    protected Expression right;
}
```
⇨
```
public class BinaryExpression extends Expression {
    public BinaryExpression(Expression left, Expression right){
        ObjectVisualization.newObj(this); // entry_st
        this.left = left;
        ObjectVisualization.setLink(this, "left", this.left);
        this.right = right;
        ObjectVisualization.setLink(this, "right", this.right);
        ObjectVisualization.exitMethod(); // exit_st
    }
    protected Expression left;
    protected Expression right;
}
```

Figure 3. Conversion of Statements that Create an 'association' Relationship

4. Implementation

The process of transforming source codes for visualization depends on the syntax and semantics of the programming language. In the present study, we implemented ObjectVisualizer for JAVA. The development environment of the tools is as follows.

Copyright ⓒ 2018 SERSC Australia

- Java Development Kit v1.8
- Eclipse Neon 2
- Jung Graph Library v2.1.1 [8]
- JavaSymbolSolver v0.5.1 [7]
- Sample Code: BWSCALC [3]

BWSCALC-VISUAL was created by converting the sample program BWSCALC using ObjectVisualizer. Figure 4 shows the visualization of the initial interaction between objects in the BWSCALC-VISUAL program.

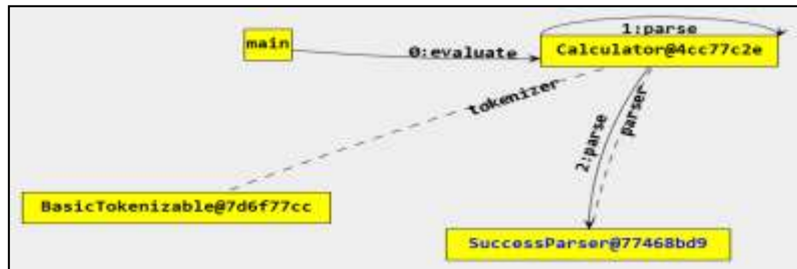


Figure 4. Visualization Screen of the Transformed Program

Although the *main0* function is not an object, it is objectified in terms of the program starting point and is expressed on the visualization screen. When an object has a reference to another object, it is represented by a dotted line between objects on the visualization screen, and the reference variable name is indicated by the line label.

As shown in Figure 4, the Calculator object has references to the *BasicTokenizable* object and the *SuccessParser* object, and the reference variable names are 'tokenizer' and 'parser', respectively. Function calls are represented by solid lines. In Figure 4, *main()* invokes *evaluate()* of the *Calculator* object ("0: evaluate") and the *Calculator* object calls its own function *parse()* ("1: parse"), then *Calculator.parse()* Calls the *parse()* function of the *SuccessParser* object ("2: parse"). With the execution of the program, the interaction between these objects is animated in real time (with some time delay) with the execution of the program.

When an object is rendered on the visualization window, its name is basically "class_name@address". However, while this makes it possible to distinguish objects from each other, their meaning is difficult to understand. On the visualization window, the name of an object is found by using the *Object.toString()* function. Thus, the developer can override the *toString()* function to change the name of an object into a more understandable string.

As shown in Figure 5, there is a *SubstractionExpression* object with two *ValueExpression* objects as left and right member variables.

When displaying *ValueExpression* objects on the visualization screen, it is more convenient for the user to understand the objects, such as "ValueExpression(7)" and "ValueExpression(2)", which include the value of the object in the name, rather than display it in the form of "ValueExpression@3564a6bd", "ValueExpression@3564a6d0".

Figure 5. Changing the Name of an Object through toString() Overriding

Copyright ⓒ 2018 SERSC Australia

5. Experiments

In this section, we will examine the effects of object interaction visualization methods on two major tasks of programming, debugging and extending functionality, through experiments.

A total of 30 computer science students in programming classes were randomly divided into two groups (N= 15), the normal group and the visualization-tool group. The normal group was tested on two tasks without visualization tools. By contrast, the visualization-tool group was tested using the visualization tools. Before starting the test, the visualization-tool group was briefed on how to use visualization tools.

5.1. Experiment on Debugging

For the debugging experiments, we modified the sample program BWSCALC to insert a logical error. That is, when the expression "-5-3" was calculated, a wrong calculation value "2" (rather than the correct answer "-8") was output.

The normal group was given the source code of BWSCALC, and the visualization-tool group was given the source code of BWSCALC-VISUAL generated by ObjectVisualizer. Both groups had no prior knowledge of the source code and were instructed to resolve the error within a set time frame. The normal group solved the problem using code review and debugging-mode in Eclipse, the JAVA development environment. By contrast, the visualization-tool group solved the problem using visualization tools along with code review and debugging-mode.

When the students in the visualization-tool group executed BWSCALC-VISUAL, the interaction of the program's objects was shown in animation in the visualization window (see Figure 6). For the expression "-5-3", *SuccessParser* parsed the values used in the expression into *ValueExpression* objects.

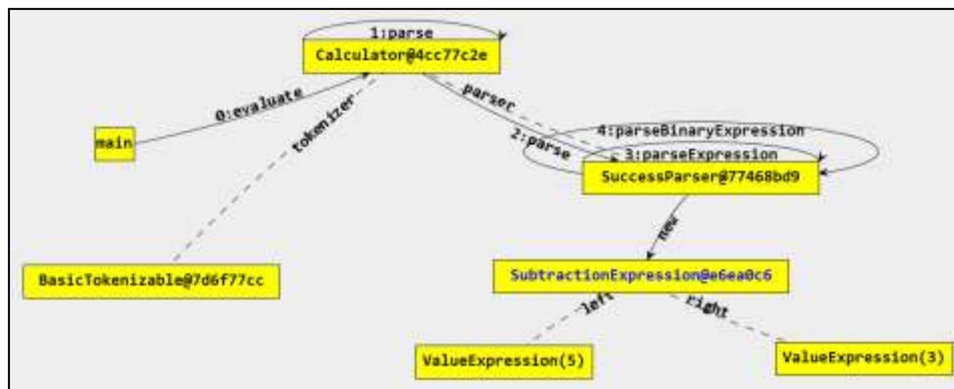


Figure 6. Visualization Screen for the Debugging Experiment

Many students in the visualization-tool group noted that, in Figure 6, the first value of the two *ValueExpression* values included in the *SubstractionExpression* was 5 instead of -5. They said that they were able to identify the problem with parsing the *ValueExpression*.

The results of the students' tests were scored from 0 to 100 points according to the progress of the debugging.

Two students in the normal group and five students in the visualization-tool group successfully solved the task. The null hypothesis and the alternative hypothesis for the test are as follows.

- H_0: $\mu \leq \mu_0$
  H_a: $\mu > \mu_0$
  - μ : mean score of the visualization tool group
  - μ_0 : mean score of the normal group

Thereafter, independent sample t-test analysis was performed on the test results of the two groups (see Table 1 and Figure 7).

Table 1. One-tailed t-test Results for the Debugging Experiment

Group	N	Mean	Std. Dev.	df	t	*p*(T<=t)
Normal group	15	30	32.2933	26	-1.7225	0.0484
Visualization-tool group	15	53.33	41.3464			

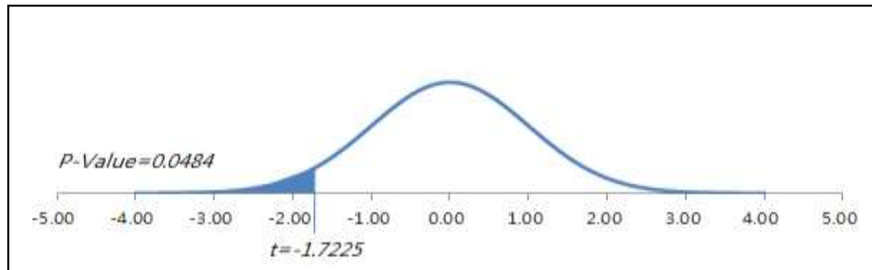


Figure 7. The Obtained t-value on the Debugging Experiment

The average score of the normal group was 30, and the average score of the visualization-tool group was 53.33. The degree of freedom was 26, and the critical value of t was -1.7225.

As shown in Figure 7, the left tail area at t=-1.7225 was 0.0484 ($p<0.05$). Therefore, the null hypothesis H_0 had to be rejected. In other words, the results support the conclusion that using the visualization tool at the significance level of 0.05 is more effective in debugging.

5.2. Experiment on Extending Functionality

The sample program BWSCALC supports '+' and '-' operations in arithmetic operations, but does not support '*'(multiply) and '/'(divide) operations. After informing the normal group and the visualization-tool group about this issue, the two groups were assigned the task of extending the function so that the program could perform '*' operation.

When the BWSCALC-VISUAL program calculated "5*3", a message informing the occurrence of the exception was displayed. In addition, the visualization screen showed what error occurred at a certain point (see Figure 8).

In Figure 8, after creating a *ValueExpression* object by parsing '5', the problem occurred in the *parse()* function of *SuccessParser* when reading the operator '*'.

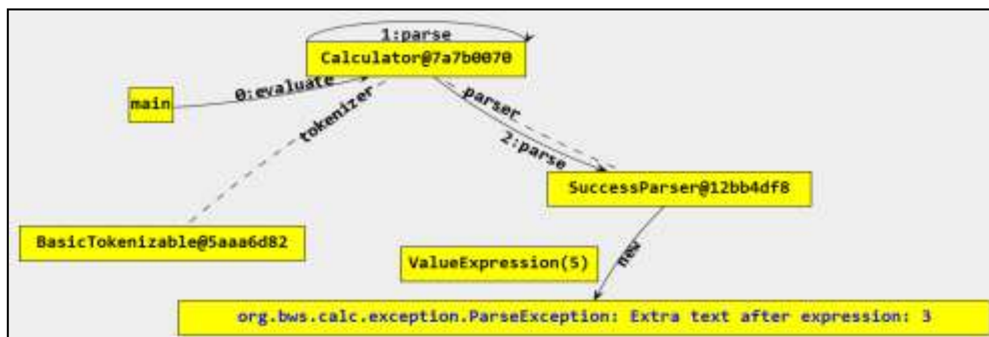


Figure 8. Visualization Screen when Performing the Multiplication Operation

 Copyright ⓒ 2018 SERSC Australia

In the sample program, *BinaryExpression* is a class that represents binary formulas. There are *AdditionExpression* (that represents addition formulas) and *SubstractionExpression* (that represents substraction formulas) classes inherited from *BinaryExpression*.

After the students figured out this program's structure, they could add a multiplication ability to the program by creating a *MultiplicationExpression* class and modifying the program to interpret '*' symbols in *SuccessParser.parse()*. While students easily understood that the *MultiplicationExpression* class was needed, it was difficult to modify the code so that '*' symbols could be interpreted. Actually, the code modification was not necessary in *SuccessParser.parse()*, but it was necessary to modify the code in *BasicTokenizable.tokenize()* that generates tokens.

The results of the students' tests were then scored from 0 to 100 points according to the progress of the task of extending the function. Overall, 5 students in the normal group and 6 students in the visualization-tool group successfully solved the task.

The null hypothesis and the alternative hypothesis for the test were set up identically to the description in Section 5.1. The results of the independent sample t-test analysis for the two groups are shown in Table 2 and Figure 9.

Table 2. One Tailed t-test Results for the Extending Software Features Experiment

Group	N	Mean	Std. Dev.	df	t	*p*(T<=t)
Normal group	15	42	44.59	28	-1.0340	0.1550
Visualization-tool group	15	58.67	43.69			

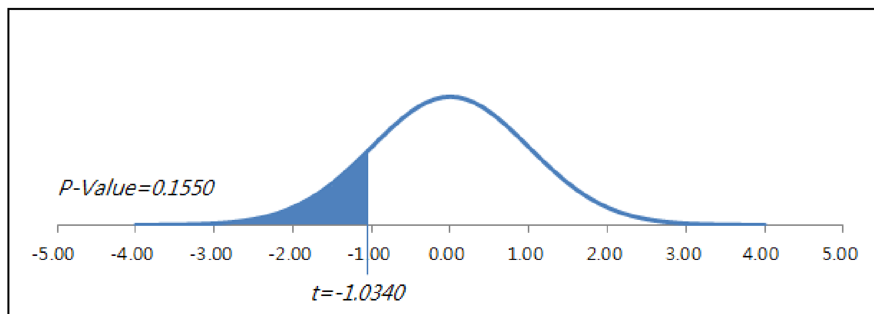


Figure 9. The Obtained t-value on the Extending Software Features Experiment

The average scores of the normal group and the visualization-tool group were 42 and 58.67, respectively. The degree of freedom was 28, and the critical value of t was –1.0340. As shown in Figure 9, the left tail area of t=-1.0340 was 0.1550 ($p<0.05$) Therefore, the null hypothesis H_0 could not be rejected. In other words, it is difficult to conclude that using the visualization tool at the significance level of 0.05 is more effective in extending program functionality.

Although the visualization-tool group had a higher average score than the normal group, the reason(s) behind this pattern can be explained as follows.

- While it is important to find out where the problem is occurring in debugging, it is important to integrate the new code into the existing code for the function extension.
- The sample program source code is so small that it does not pose any difficulty to understand the code with the code review.

In conclusion, while the proposed visualization tool shows great strengths in debugging work to find out the problems of the existing code, it does not show significant strength in adding new functionality in small programs.

6. Conclusions

In the present paper, we introduced object visualization method that helps to educate object-oriented programming concept and ObjectVisualizer system, which is a DPV tool that implements it. The ObjectVisualizer animates the behavior of the program's internal objects simultaneously with the execution of the program. In our experiments, we also demonstrated that the visualization tool is effective in debugging and extending functionality. Specifically, our results suggest that the proposed visualization tool is helpful in debugging tasks to find problems, but is less effective in terms of providing meaningful help in extending new functionality in small programs.

Future research would involve is changing the visualization system implemented in a stand-alone system into a plug-in module of eclipse, so that the visualization function can be used in real time without the transformation process.

Acknowledgments

This Research was supported by SeoKyeong University in 2016.

This paper is a revised and expanded version of a paper entitled "Development of an Object Interaction Visualization Tool" presented at 2017 1st International Workshop on Cultural and Technological Exchange and Mutual Development of The Pacific Rim Countries, Cheonan, Korea, December 16-17.

References

[1] R. B.-Bassat Levy, M. Ben-Ari and P. A. Uronen, "The Jeliot 2000 program animation system", Computers & Education, vol. 40, no. 1, **(2003)**, pp. 1-15.
[2] M. D. Byrne, R. Catrambone and J. T. Stasko, "Evaluating animations as student aids in learning computer algorithms", Computers & Education, vol. 33, no. 4, **(1999)**, pp. 253-278.
[3] https://github.com/brian-w-smith/bwscalc, **(2017)** June 5.
[4] S. Diehl, "Software visualization: Visualizing the structure, behaviour, and evolution of software", Springer, Berlin, German, **(2007)**.
[5] B. M. Hill and A. Monroy-Hernández, "The cost of collaboration for code and art: Evidence from a remixing community", Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW '13). San Antonio, Texas, USA, **(2013)**, pp. 1035-1046.
[6] S.-m. Hong, Y.-j. Lee, J.-h. Kim, K.-h. Son and M.-n. Jung, "A study to Train Human Resources and to Support Their Employment and Entrepreneurship for New Future Industries", Science and Technology Policy Institute, Korea, **(2016)**.
[7] https://github.com/javaparser/javasymbolsolver, **(2017)** June 5.
[8] http://jung.sourceforge.net, **(2017)** June 5.
[9] C. Kehoe, J. Stasko and A. Taylor, "Rethinking the evaluation of algorithm animations as learning aids: an observational study", International Journal of Human-Computer Studies, vol. 54, no. 2, **(2001)**, pp. 265-284.
[10] S. Kim and J. B. Chae, "Trend Analysis of Educational Programming Language and Teaching-Learning Examples", KERIS Issue Report RM 2014-25. Korea Education and Research Information Service, **(2014)**.
[11] K.-h. Kim, "2015 Revised Curriculum The right direction of software education and its case study", Seoul Education, Seoul Education Research & Information Institute, vol. 226, **(2017)**.
[12] J. M. Lavonen, V. P. Meisalo, M. Lattu and E. Sutinen, "Concretising the programming task: A case study in a secondary school", Computers & Education, vol. 40, no. 2, **(2003)**, pp. 115-135.
[13] C.-y. Lee, "Special Report: Obligatory of elementary and middle school s/w education and foster digital human resources, Promotion and prospects", Edzine, Autumn, Korean Educational Development Institute, vol. 41, no. 3, **(2014)**.
[14] M. Marji, "Learn to Program with Scratch", San Francisco, California: No Starch Press, **(2014)**, pp. xvii, 1-9, 13-15.

Copyright ⓒ 2018 SERSC Australia

[15] Y. Miyadera, K. Kurasawa, S. Nakamura, N. Yonezawa and S. Yokoyama, "A real-time monitoring system for programming education using a generator of program animation systems", Journal of Computers, vol. 2, no. 3, **(2007)**, pp. 12-20.
[16] A. Moreno and M. S. Joy, "Jeliot 3 in a demanding educational setting", Electronic Notes in Theoretical Computer Science, vol. 178, **(2007)**, pp. 51-59.
[17] W. I. Osman and M. M. Elmusharaf, "Effectiveness of combining algorithm and program animation: A case study with data structure course", Issues in Informing Science and Information Technology, vol. 11, **(2014)**.
[18] T. Rajala, M.-J. Laakso, E. Kaila and T. Salakoski, "Effectiveness of program visualization: A case study with the ViLLE tool", Journal of Information Technology Education: Innovations in Practice, vol. 7, **(2008)**, pp. 15-32.
[19] M. Resnick, J. Maloney, A. Monroy-Hernάndez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman and Y. Kafai, "Scratch: Programming for All", Communications of the ACM, **(2009)** November.
[20] W.-c. Shin, "A Study on Object-Oriented Programming Education using Visualization Method", Journal of The Korean Association of Information Education, vol. 21, no. 5, **(2017)** October, pp. 557-565.
[21] J.S. Sung and H.C. Kim, "Analysis on the International Comparison of Computer Education in Schools", The Journal of Korean Association of Computer Education, vol. 18, no. 1, **(2015)**, pp. 45-54.
[22] Tekdal, "The Effect of an Example-Based Dynamic Program Visualization Environment on Students' Programming Skills", Educational Technology & Society, vol. 16, no. 3, **(2013)**, pp. 400-410.
[23] TIOBE Index, https://www.tiobe.com/tiobe-index/, **(2017)** June.

Author

Woo-Chang Shin, he received the Ph.D. degree in computer engineering from Seoul National University in 2003, Republic of Korea. Currently he is a Professor at Department of Computer Science, SeoKyeong University. His research interests include software development methodology, software modeling, software testing, and formal specification.

Copyright ⓒ 2018 SERSC Australia