Visual Debugger with a Customizable View

Rifqi Adlan Apriyadi, Hidehiko Masuhara, Youyou Cong

Visual debuggers provide more concrete representations of program behavior than what might normally still be abstract. For object-oriented languages, their object-centric components, such as objects and their relationships, are the information their visual debuggers aim to concretize. However, none of such debuggers provide the capability to customize their graphical view. Customization ought to facilitate better comprehensibility of the program behavior, which consequently aids in bug discovery, by allowing for a more focused program state view. This can be achieved by customizing the representation of objects of a class in accordance with the theoretical concept, or by abstracting away unnecessary information. This research proposes a visual debugger for Java with customization features for its graphical view that allows users to customize the visual representation of the program state through customization specifications. To provide flexible customizability, the proposed system would provide customization elements that function as building blocks for users to use in their specifications.

1 Introduction

The main goal of visual debuggers is to enhance program state understanding, and consequently, that of the program behavior. Visual debuggers display relevant information graphically to facilitate this purpose. This is fueled by the fact that visualizations of abstract information concretize it by displaying relationships and patterns [5].

For object-oriented languages, understanding the behavior of a program is significantly more difficult than understanding its structure or design [3]. Therefore, it is important to display the object-centric information of *program states* — the state of the program at a point in time or in program suspension — whilst debugging [4]. This includes the values of variables of objects and their relationships with other objects with regards to their references.

Moreover, this view is frequently displayed as a tree of objects where referenced objects are lower in the tree as their children. Tree structures depict items that could have children which could also have children of their own. However, each item could at most only have one parent and children of an item could not include any of the its parents. This means that a tree structure is not the most accurate nor concrete depictions of these relationships as objects could be referenced by multiple different objects while the referenced ones could, in turn, do the same to more, even the ones higher up in the tree.

To those efforts, visualizations are often in the form of object diagrams — or their variants — to show the aforementioned relationships between objects [11] [8] [7] [9]. In an object diagram, an object is a node or vertex and their references to other objects are represented by directed edges to the them. An simple example of an object diagram could be

Visual Debugger with a Customizable View

Rifqi Adlan Apriyadi, Hidehiko Masuhara, Youyou Cong, School of Computing, Department of Mathematical and Computing Science, Tokyo Institute of Technology.

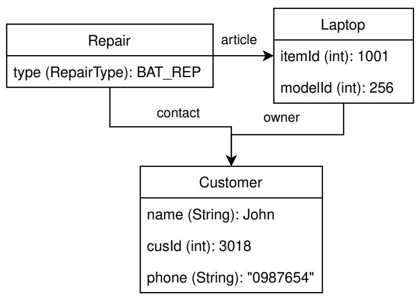

図 1 Example of an Object Diagram

seen in Figure 1, which shows a `Repair` object representing a repair order which has variables `article` and `contact` which reference a registered item and a customer, respectively.

However, although the main purpose of visual debuggers is to facilitate program understanding, they often come short in reaching this goal in some situations. Large program states often manifest this shortcoming by cluttering the debugger view with a large object diagram, even for modestly sized programs [4] [12]. Though some of these projects provide workarounds by allowing users to prune object nodes from the view to simplify it, this solution merely removes information relating to these objects completely when some parts of it might still be relevant [8].

Furthermore, it is not uncommon to have implementations of a concept to be different from how they are portrayed in theory. An example of this is the internal representation of lists when writing a heap sort algorithm. A visual debugger will normally display lists the way they are represented internally, which greatly impedes the programmer in understanding the current program state while debugging, given that they would most likely need to visually observe the behavior of their program working on a heap structure. Even though the purpose of the visualization is to facilitate the understanding of concepts and their behaviors [10] [6] as represented by the visualized objects, they are unequipped to cover this practice, causing the process to become counter-intuitive.

Given this, the main problem causing these two shortcomings is the rigidity of the graphical views in terms of how they display program states. Users cannot control the visualization of their program states to match their situations, harming users' understanding of program states.

To alleviate this problem, this research proposes a visual debugger for a chosen object-oriented programming language, Java, where users are able to customize the object diagram view by means of a customization specification to fit their current needs. The proposed system is to be built on Visual Studio Code utilizing its various available APIs. A comparative experiment is to be conducted to verify the effectiveness of the debugger with regard to the problem it aims to solve.

The aim of this research is to investigate an approach to the development of a visual debugger with a feature that allows for flexibility in displaying these program states. Section 2 details the proposal and key features relevant to the problem at hand. Furthermore, Section 3 presents the specifics of the implementation of the prototype of the tool aimed to represent this research whose success is to be validated as described in Section 4. Finally, similar work will be mentioned in Section 5 followed by this paper's conclusion in Section 6.

2 Proposal

This research proposes a visual debugger with customization features in its graphical view. Among other things, these features ought to cover the aforementioned common flaws present visual debuggers for object-oriented languages. This section first describes the customization features which will then be followed by the description of how customization is a potential solution.

Though the purpose of the customization feature is to avoid rigidity, it is also important for the feature itself to not stumble onto the same pitfall. In this effort, the debugger provides users with customization elements that act as building blocks. With these building blocks, the users should be able to specify their own customizations to the debugger view in various components of the language, such as for a specific class, class field, method, method parameter, etc. Through a specification, the debugger view will display program states accordingly once debugging starts.

Types of customization elements include:

(a) Invisible Edge: The removal of an edge relating two object nodes from the view

(b) Imaginary Edge: The addition of an edge relating two object nodes to the view

(c) Invisible Object: The removal of an object node and all its incoming and outgoing edges from the view

(d) Imaginary Object: The addition of an object node to the view

(e) Node Description: Allows each object to have their own descriptions which could tailor to its current state.

For example, Figures 2 and 3 show sample mock-up object diagrams of the visual debugger. Both subfigures display a `Single` object — representing a *Single* move made by a player involving only one card in a card game — that has variables `moveMaker` of class `Player` and `card` of class `Card` representing the player making the Single move and the card within the move, respectively. Figure 2 shows the uncustomized view and Figure 3 shows it with some customizations applied. The customizations used in Figure 3 are:

- Declaring the Card class as an invisible node
- Declaring the card variable in the Single class to be an invisible edge
- Adding a custom description to object nodes of the Single class to show the description of the card variable.

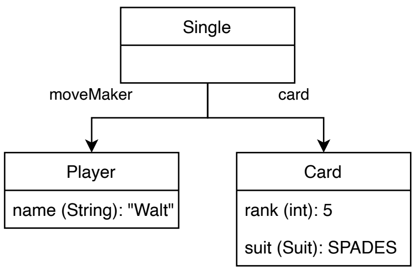

図 2 Uncustomized

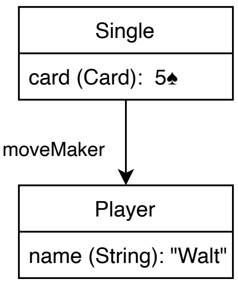

図 3 Customized

Furthermore, it is anticipated that it would be useful for customizations to be accompanied by optional conditions which allow users to specify when a customization is applied. For example, to specify the list to heap customization as discussed in the previous section, conditions would prove useful to indicate where to create imaginary edges and where to remove ones that would normally exist. It is also envisioned that some combinations of these elements will be used frequently for higher-level customizations, such as in the use case of merging two nodes of objects where one wraps the other in the implementation side. Therefore, a few higher-level customization elements that are predicted to be of frequent use that serve as shorthands for combinations of the building block customization elements shall be provided as well. In addition to specifying the customization before debugging, the debugger would also allow for the user to customize the view while debugging, which would serve as a *quality-of-*

life feature for small customizations or to preview how the customization that the users have in mind.

For the case of the inconsistency between the concept and its representation in runtime, users could customize the view to closer resemble the representation of the concept in theory despite how the data is actually stored internally. For example, users could customize the view to show a node with a textual description of a matrix instead of showing an array that refers to more arrays, the former allowing for a more comprehensible view.

On the other hand, for the case of visual clutter, users could specify customizations to the view that abstract information that is irrelevant to the current situation [13]. For instance, users could reduce visual clutter by making immutable objects of a class as node elements of objects that have them as reference and removing any of their nodes from the view completely, if the behavior of this class is currently of no importance.

3 Implementation

The tool is to be developed as a debugger for Java due the support it is given by the software used for the development. Furthermore, it is a convenient choice given that most research projects taken as reference for this research [7] [8] [9] [1] build their tools for Java, as well.

3. 1 Visual Debugger User Interface

The UI of the debugger will be implemented as an extension for Visual Studio Code, hereafter referred to as VSCode. WebView [2] will be used to display the UI due to the fact that extensive flexibility in designing the UI is provided and that VSCode has its own dedicated API for it. This flexibility will also allow for the development to be facilitated by a visualization library that lies in abundance. The UI would extend the debugger UI of VSCode. In either case, the architecture of the tool to be built

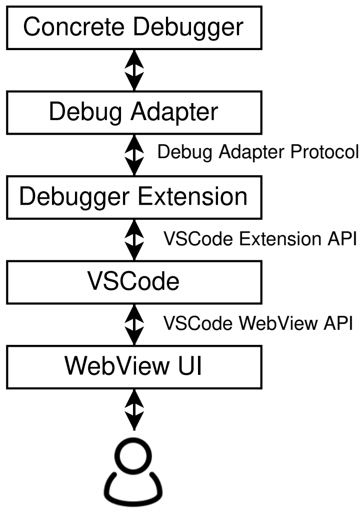

図 4 The Architecture of the Visual Debugger

can be seen in Figure 4 and is described as follows:

- The user interacts with the WebView UI
- The WebView UI exchanges information with VSCode using the VSCode WebView API
- VSCode communicates the visual debugger extension that is to be built using the VSCode Extension API
- The extension manages the debugger by communicating with VSCode's *Debug Adapter* (DA) for Java through the *Debug Adapter Protocol* (DAP)
- The DA handles the communication with Java's concrete debugger.

3. 2 Debug View Customizability

The main concerns in the implementation of the view customization features of the visual debuggers are how users would specify their customizations and how the visual debugger would understand them. For the former, there are a few options available.

One option would be to provide a dedicated *Domain Specific Language* (DSL) that users would use to write in to specify their customizations in a separate file. Requiring users to specify their customizations in a separate file allows them to be as expressive as need to be in their customization specification. However, this would require them to specify which customization applies to which part of the program, which might involve cumbersome identification. This is especially troublesome for large projects where paths to classes are often long.

Another option would be to provide annotations users could use in different parts of the program, which would also require a DSL, albeit not necessarily one as expressive as the previous option. Users would have greater convenience in specifying which customization applies to which parts of the program given that annotations could generally be placed on virtually most of the language's components, resolving the identification problem from the previous option. However, as specifications along with their conditions would be made in these annotations, it is questionable how expressive these specifications could be given that heavily expressive specifications may litter the file. Additionally, this option would not allow users to write customizations on parts of the program in read-only files.

Finally, the third option is to incorporate both of the previous options, taking the best of both worlds. If the user is not to provide lengthy specifications for each annotation or if they are not to be written in a read-only file, then these annotations could remain and would not disarrange the program file. Otherwise, they could be made written into a separate file. The downside to this option is the development effort required to implement both of these functionalities.

Regardless of the specification method option chosen, specifications are to be compiled into an XML file, which the debugger could more easily parse and create its internal representation from.

4 Validation

The main aspect to be validated in this research is how much the use of view customization in a visual debugger improves the understandability of program behavior, or lack thereof. Since the view of the visual debugger will be a variant of an object diagram, the component whose understandability is to be measured is the object diagram in the view and certainly not the code used for the validation itself. However, although there are already studies on measuring understandability of code, there is yet to be a devised metric for the measurement of diagram understandability.

Therefore, a comparative experiment ought to be conducted which aims to compare the average time it takes for participants to understand the behavior of provided programs in two different settings. The control group would be tasked to understand the behavior of programs using the visual debugger without any customizations applied on the debugger view. In other words, this group would only use the basic view of the debugger. The treatment group would be tasked to do the same with the same set of programs provided, except they will be using the visual debugger with customizations already applied. To have the provided tools be uniform to all participants in the treatment group, the customization specification will already be provided for them. For both groups, they will be asked to do their tasks using the debugger as much as possible. It is expected that the average time taken to understand each program in the treatment group to be smaller than that of the control group.

This experiment will certainly have extraneous variables present that may influence the results of each participant. The biggest of these variables would be the amount of experience and knowledge participants already have in the experimentation environment. A participant may have more expe-

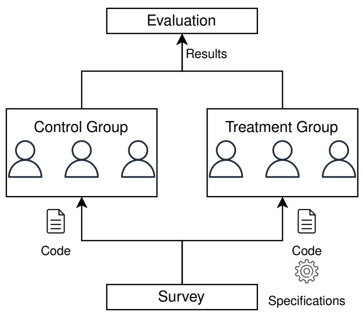

図 5 **Flow of the Experiment**

rience in the programming language, in program comprehension and debugging, or in the theory the programs emulate. Though these variables could not be avoided, they should have the capacity to be minimized through careful distribution of participants. For example, the distributions of the two groups could be done in such a way that the groups are balanced in terms of their participants' experience and knowledge, the information of which could be obtained by a survey some time before the experiment is conducted. Figure 5 illustrates the flow of the experiment.

5 Related Work

This section discussed the more well-known tools for program visualizations. Though not all of them have the same concept as the visual debugger discussed in this paper, they all provide visualizations aimed at concretizing program behavior compared to their predecessors.

Java Interactive Visualization Environment (JIVE) [8] is a visual debugger built as a plug-in for the Eclipse Java debugger. It features an object diagram view, a sequence diagram view, backwards stepping, and query-based debugging — though without the flexibility this research aims to achieve.

BlueJ [7] is an integrated Java development environment designed for introductory teaching to object-oriented programming. Its main property is its simplicity and pedagogy. BlueJ provides an interactive environment in which students could interact directly with classes and objects, allowing them to learn the behavior of the object-oriented programming paradigm.

Similarly, Jeliot3 [9] is a pedagogic tool aimed also at novice students for object-oriented programming, as well. It provides a fully or semi-automatic visualization of the data and control flows of the program. In addition to supporting visualizations of object instances and inheritance, this version of Jeliot also introduces object-oriented concepts.

Velázquez-Iturbide et al. [13] developed WinHIPE, an environment providing students the capability of customizing the visualization of expressions. The customization areas it provides are in the choice between textual or graphical visualizations, or a mixture of both, the typographic styles used, and visualization simplification.

6 Conclusion

Though the addition of visual debuggers and similar tools has been acclaimed, the very nature of visualizations is that they mirror abstract information in a finite space, leading to rigidity. The research this paper describes aims to discover the methods, considerations, and results of developing a visual debugger that overcomes the trap of rigidity through customizability.

The visual debugger proposed is to allow users to specify their customizations which will be automatically applied once they start debugging. Basic customization elements are provided that serve as building blocks for any customization. The debugger will be developed as an extension to Visual Studio Code due to the fact that it provides convenient APIs for the view, the extension development, and the debugger back-end.

The validation of the success of the tool is planned to be done by means of a comparative experiment, where participants are to be separated into control and treatment groups. They are to be separated accordingly such that the programming experience and proficiency of the two groups are as balanced as possible. This is important given the fact that the rate in which participants can understand program behavior through the use of the tool is the variable of interest, which may vary immensely.

参考文献

[1] Cardillo, G., Schürmann, P., and Lagadec, A.: *Visual OO Debugger*, PhD Thesis, OST Ostschweizer Fachhochschule, 2022.

[2] Cockburn, A., Greenberg, S., McKenzie, B., Jasonsmith, M., and Kaasten, S.: WebView: A graphical aid for revisiting Web pages, *Proceedings of the OZCHI*, Vol. 99, 1999, pp. 15–22.

[3] De Pauw, W., Lorenz, D. H., Vlissides, J. M., and Wegman, M. N.: Execution Patterns in Object-Oriented Visualization., *COOTS*, Vol. 98, 1998, pp. 16–16.

[4] Gestwicki, P. and Jayaraman, B.: Interactive visualization of Java programs, *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, IEEE, 2002, pp. 226–235.

[5] Ho, S. Y.: Seeing the value of visualization, (2010).

[6] Keahey, T. A. et al.: Using visualization to understand big data, *IBM Business Analytics Advanced Visualisation*, Vol. 16(2013).

[7] Kölling, M., Quig, B., Patterson, A., and Rosenberg, J.: The BlueJ system and its pedagogy, *Computer Science Education*, Vol. 13, No. 4(2003), pp. 249–268.

[8] Lessa, D., Czyz, J. K., and Jayaraman, B.: JIVE: A pedagogic tool for visualizing the execution of Java programs, *Bericht, Univ. of New York, Buffalo*, (2010).

[9] Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M.: Visualizing programs with Jeliot 3, *Proceedings of the working conference on Advanced visual interfaces*, 2004, pp. 373–376.

[10] Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rößling, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., et al.: Evaluating the educational impact of visualization, *Acm sigcse bulletin*, Vol. 35, No. 4(2003), pp. 124–136.

[11] Oechsle, R. and Schmitt, T.: Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi), *Software visualization*, Springer, 2002, pp. 176–190.

[12] Sinha, V., Karger, D., and Miller, R.: Relo: Helping users manage context during interactive exploratory visualization of large codebases, *Visual Languages and Human-Centric Computing (VL/HCC'06)*, IEEE, 2006, pp. 187–194.

[13] Velázquez-Iturbide, J. Á. and Presa-Vázquez, A.: Customization of visualizations in a functional programming environment, *FIE'99 Frontiers in Education. 29th Annual Frontiers in Education Conference. Designing the Future of Science and Engineering Education. Conference Proceedings (IEEE Cat. No. 99CH37011*, Vol. 2, IEEE, 1999, pp. 12B3–22.