2014 Second IEEE Working Conference on Software Visualization

Templated Visualization of Object State with Vebugger

Daniel Rozenberg, Ivan Beschastnikh
Department of Computer Science
University of British Columbia
Vancouver, Canada
rodaniel@cs.ubc.ca, bestchai@cs.ubc.ca

***Abstract*—Software developers often need to inspect the state of objects during debugging. Existing debuggers display a textual representation of the state of selected objects. While these textual representations often contain enough information, they are also difficult to comprehend. For example, an object that represents a color is traditionally represented by listing the numbers that comprise its RGB values. This representation, while complete, is hardly comprehensible.**

We describe Vebugger, an IDE plugin for Eclipse that displays object state visually. Recalling the previous example, Vebugger displays the actual color that a `Color` object represents in addition to its RGB values. This representation is easier to understand. Vebugger visualizes object types using a set of extensible templates. These templates are written in HTML and CSS, and they are matched to Java types by inspecting the type hierarchy. We developed a dozen such templates for a diverse set of Java types to demonstrate the capabilities of the system. Vebugger is preliminary work, we also detail future research directions and our planned evaluation strategy.

I. Introduction

Non-trivial programs are challenging to debug and understand, and a foremost challenge in program understanding is the invisibility of software [5]. This problem is especially acute when attempting to understand the state of a running program. Developers rely on tools to augment their understanding of the runtime behavior of their programs and many of these tools are visual by design. These tools can be rudimentary, such as inspecting the program's console log, or advanced, such as formulating questions about the observed output [12] and inspecting state changes with a breakpoint debugger. Developers frequently use such debuggers to inspect variable state. For example, Murphy et al. [15] found that an overwhelming majority of participants in their study used the variables view while debugging.

In most breakpoint debuggers the state of various objects is reduced to a simple string, regardless of the amount of information contained in the object or its complexity. In other cases an object's representation might not contain the entire state, but rather a subset that consists only of its most essential parts. This is done to avoid overwhelming developers with an excessive amount of information. For example, in Java the textual representation of a `File` object is just the filename; it does not display whether the file exists, its access permission, or if it is a regular file, a directory, or another type of file.

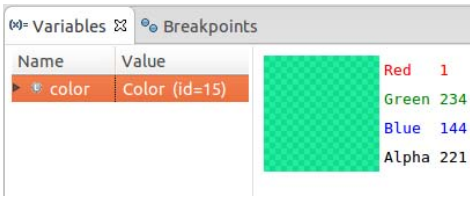

Figure 1: Vebugger's visualization of a `Color` class instance inside the Variables view of Eclipse's debug perspective. This visualization exposes the color represented by the object in a fashion more easily comprehensible compared to a textual representation.

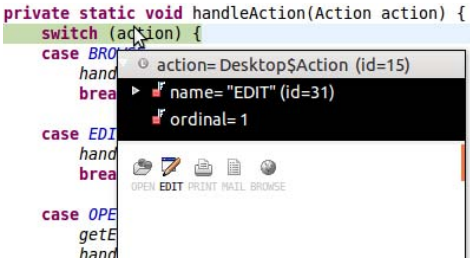

Figure 2: Vebugger's visualization of a `Desktop.Action` enum instance inside a runtime mouse-over tooltip. Unlike the textual representation, this visualization exposes the alternative visual states of the enum.

Furthermore, string representations can be difficult to comprehend. For example, in Java the textual representation of a `Map` object is a concatenation of the textual representations of every key and value pair in the map joined by an equals sign, and each pair joined by a comma, leading to long strings. Sometimes these textual representations are simply missing.

We describe a debugging aid tool called Vebugger that supplements objects with visual representations. Figure 1 demonstrates how Vebugger visualizes an instance of the `Color` class in a Java program. This representation, on top of displaying the numeric RGBA values of the color, displays the color itself, which eases the burden required to comprehend the values compared to the default `.toString()` representation of the same class. Figure 2 demonstrates how Vebugger visualizes an instance of the `Desktop.Action`

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.26

enum, a type that represent common actions that desktop application can perform. This representation provides a bird's eye view of the state of the enum along with the enum's other potential values, sparing developers from the need to explore the enum in a separate step. Vebugger enables developers to write templates for custom and built-in types. These templates are written in HTML and generate a visual output from object instances.

Vebugger currently supports visualizations of Java types[1] and its design focuses on Object-oriented programming languages, though it can be extended to visualize stack state of programs written in functional languages. Many popular and general purpose languages, such as C++, Java, Python, and Ruby are OOP languages, making the current design of Vebugger already broadly applicable.

To demonstrate the capabilities of Vebugger we implemented templates for a dozen types in the java.* package that fall into three general categories. There are templates for *GUI elements* like class Font, templates for the *multi-property* classes like File, and for *data structure* classes like LinkedList. We discuss these categories and templates in greater detail in Section IV. Vebugger is free software and is available for download [17].

The remainder of this paper is organized as follows. In Section II we discuss the criteria that motivated us to design Vebugger. In Sections III and IV we discuss implementation details and example uses of Vebugger that we explored. In Section V we discuss the future direction in which we intend to take Vebugger. We review related work in Section VI and conclude in Section VII.

II. Design criteria

Vebugger's design was directed by four criteria:

Typed visualizations. Type systems are an important feature in many modern programming languages; each variable or watch expression that a developer explores is associated with a type in the language. Vebugger must support visualizations based on type.

Extensibility through customizable templates. Developers create and use custom types in their programs and Vebugger must support these. We chose to use a template-based approach to make the visualization creation process accessible. We discuss future ideas to make the process even more accessible in Section V-B. Templates also enable Vebugger to provide context-sensitive object visualizations. We discuss this idea in greater detail in Section V-C.

IDE integration. Developers tend to debug on the platform where they develop the code. Existing debuggers are tightly integrated with IDEs and Vebugger must be as well.

[1]Java type refers to classes, interfaces, and enums

Do no harm. Vebugger must not degrade a developer's experience: Vebugger must be able to fall back to existing behavior of using string representations for objects.

III. Implementation

We implemented Vebugger as an Eclipse plugin. The plugin creates a new value details pane for the debugger (e.g., Figure 1). Our implementation requires users to add a helper library to their Java project. Templates are classes that extend the VebuggerTemplate class. Templates must implement two methods, getType and render, to signal which type the template matches and to generate HTML from an object instance, respectively.

Vebugger uses Java's built in runtime type matching to pair the object's type to a template that returns the first matching class via getType. The first matching template is the template class that handles the object's class directly, through its super classes, or through one of its interfaces, in that order. Once a matching template is found Vebugger invokes the template class's render method. Vebugger displays the .toString() value of the instance if it does not find a matching template. In this way Vebugger does not harm developers' comprehension in case of a missing template.

We use HTML and CSS for implementing templates due to their ubiquity and the support for these languages in Eclipse. Custom types require new templates. We chose to match each template with a single Java type for simplicity. One limitations of this choice is the difficulty in sharing templates between similar classes. We plan to mitigate this by replacing the templating backend with an industry standard engine, such as JSP or Razor. Such an engine would also support nested templates.

IV. Example categories

We now present four categories of types in the Java standard library that are representative of the kinds of types that we would like to support in Vebugger. We included a dozen templates in the initial release of Vebugger which are described below in their respective categories.

GUI elements and media. Types such as buttons, text-input widgets, graphics-related types, audio, and videos are a natural fit for visualization. These classes can be visualized to approximate their runtime appearance. As these types represent visual elements, the visual representation is bound to be more useful than a textual representation. We included three templates in this category:

Class Font: Represents a single font and some formatting properties. The .toString() representation of this class displays the font's name and properties such as size and whether the font is bold or italicized. Vebugger displays a famous English pangram using the font that the object represents.

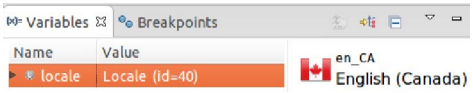

Figure 3: Vebugger's visualization of a Locale instance.

Class Point2D: Represents a single point in 2D Cartesian space. The .toString() representation of instances of this class display the coordinates. Vebugger shows the position of the dot on a Cartesian plane.

Class Color: Represents a single color with an optional alpha component. The Vebugger display of this class is demonstrated in Figure 1, explained in Section I.

Aggregate types. As the importance of each object property depends on the context in which the object is used, we believe all properties should, by default, deserve equal representation. Doing so with text results in representations that are difficult to comprehend, which is why many of these types have their .toString() method return a unique identifier. Vebugger can expose these properties in a way that balances the developer's cognitive burden. We included three templates in this category:

Class File: represents a file. The .toString() outputs the file's name. Vebugger displays all the file properties in a table, such as access permissions and the file type.

Class Currency: represents a single currency. This object includes information like the currency's code, symbol, region, and name. The .toString() representation is the currency's code. Similar to the File template, Vebugger displays all fields in a table structure.

Class Locale: represents a specific geographical, political, or cultural region. The .toString() representation displays the locale code. Vebugger adds the region's flag as well as the full language and region names (see Figure 3).

Small finite state. Types such as enums can be displayed by exposing all potential states grayed out, and the active state emphasized. We included one template in this category:

Enum Desktop.Action: represents common actions in desktop applications. The .toString() representation displays the active state of the enum. Vebugger displays all available states, as well. It also displays a familiar icon to represent the value (see Figure 2).

Data structures. Types that represent data structures are also a natural fit for visualization. They are often represented visually in textbooks (e.g. [6]) and drawn by developers on whiteboards. We included five templates in this category:

Class Object[][2]: represents a collection of elements with a predefined size. The .toString() representation displays this format: [a, b, . . . , n] where a, b, n are the .toString() values of the elements. Vebugger embeds the visualizations of the elements of the list in a table with the associated index number paired with each element.

[2]Object[] is an array of objects, the base class for all non-primitive arrays in Java.

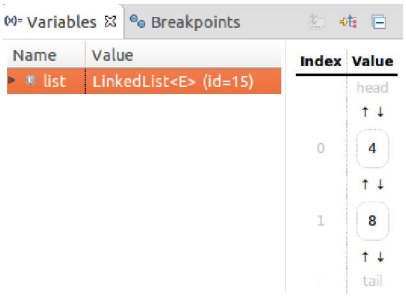

Figure 4: Vebugger's visualization of a LinkedList instance.

Interface List: represents a collection of elements with basic list actions (add element, remove element, empty the list, etc.). This interface is represented the same way as Object[], both with and without Vebugger.

Class LinkedList: an implementation of List that stores elements in a linked list. The .toString() representation is identical to the other List implementations. Vebugger displays a table similar to that of List but also displays the connections between values as arrows (see Figure 4). This follows textbook conventions regarding linked lists.

Interface Map: represents a dictionary of key-value pairs. The .toString() output looks like this: {Ka=Va, Kb=Vb, . . . , Kn=Vn} where Ka, Kb, Kn and Va, Vb, Vn are the .toString() of the keys and values respectively. Vebugger displays a table similar to List, but in place of the indices it embeds the visualization of the keys.

Interface Set: represents a set of unique elements. Its .toString() output is identical to that of List. Since the elements in a Set have no order, Vebugger displays each element in the set in a random scattering of circled nodes.

This list of categories is not comprehensive. Each type has unique constraints and developers should consider these when creating a new template. We believe that these categories begin to capture a taxonomy of possible templates. Vebugger can use these categories to provide a baseline template for developers who are creating custom templates.

V. Future Work

A. User studies of Vebugger

We intend to design and conduct a survey to determine the set of type templates to support natively in Vebugger. The survey will ask developers to either sketch a visual representation of types in varying states or declare that a type is better represented textually. The results will be aggregated to find recurring themes. We will categorize types based on whether many participants preferred to visualize a type in a similar manner, in many different styles, or if the type was repeatedly marked as unvisualizable.

By creating visualization templates for types based on the results of the survey, we can next conduct a user study involving developers and students. We will ask the participants to debug programs with and without Vebugger and test the participants on how well they understand the state of various objects based on their visual/textual representations. The results of this study will help us understand whether developers perform better with Vebugger.

B. Better support for custom templates

Developers who wish to use Vebugger must create a template for each custom type or super class of the type that they wish to visualize. This is a barrier to developers who need to visualize many custom types. Vebugger can be modified to automatically create templates for a custom type. One option is to create tabular representations based on getter properties, like the methods named $.\mathrm{getX}()$. Getter methods do not have side effects and return values that are deemed interesting. Vebugger can generate a table with the name and value of each of these getter methods for types that do not have a template.

Another way to assist developers is to equip Vebugger with a drag-and-drop based editor with a WYSIWYG interface. The draggable components can be a type's public fields and getters. For arrays or $\mathrm{Iterable}$ values the interface can generate a list or a table. For values that are complex objects the interface can inline the entire sub-object's visualization or allow developers to choose from one of the sub-object's own values, from values of the sub-object's own sub-object's, and so on. We intent to conduct a user study to learn how to improve the template creation process.

C. Support for context-specific templates

In general, visual representations of types like String or subclasses of Number would not provide any benefit over their textual representation. However, the *context* in which variables of these types are used can benefit from visualizations. For example, an int is a numeric value but it could also represent a stock price. In this context this variable can benefit from coloring the number green or red and adding a plus or minus sign, depending on whether the value has recently increased or decreased.

Vebugger can be equipped with an ability to switch templates for specific instances of a type depending on the context. For this to be usable the template creation process must be streamlined so developers can create on-the-fly visualizations to support their context. Additionally, Vebugger could automatically infer the context based on static and dynamic analysis of the code. For example, the values of a numeric variable that changes often can be visualized with a line chart to display the sequence of values; this is similar to [4]. Alternatively, values of a variable that changes often but uses a limited set of values can be visualized with a histogram. By analyzing the data flow, Vebugger can recommend similar templates based on connections in the data flow graph. Related variables are likely to be used in similar contexts.

D. Scalable and recursive visualization and navigation

Types with dynamically-sized content may become unmanageably large to visualize with default Vebugger templates. For example, a list with 10,000 elements will be represented as a sequence that displays all elements. Templates can be extended to support dynamic resizing, based on the size of the container and user interactions. One trivial option is to collapse most elements and let the user expose them by clicking an "expand" button. Another option is to generate a histogram over the elements to expose statistics instead of individual elements, to provide zoom controls to facilitate quick exploration, or to display the elements with a fish-eye scrolling interaction.

In the case of complex classes that contain references or pointers to other fields, visualizations can potentially become too large to fit inside the Vebugger pane. The referenced types can instead be visualized using a hyperlink. Clicking that hyperlink will change the selection to the related object and display the template. Facilitating navigation from the template can potentially ease the exploration of the relationship between various object instances. The same concept can be applied to large data structures. The hyperlinks themselves could be generated in a way that uniquely represents each object, so a smaller template would be required for objects that are expected to be displayed as hyperlinks as part of the visualization of other objects.

E. Supporting animations

Animation of changes to data structures have been shown to increase comprehension of algorithms by students [11]. Vebugger can be modified to support animation in templates to visualize transitions between states. The benefit of animating state transitions is not limited to data structures. For example, a template for a $\mathrm{BufferedOutputStream}$ instance can be in the form of a pipe with a transition animations that changes the pipe's width depending on whether the volume of content waiting inside the buffer has increased or decreased since the previous observation. Adding animation to Vebugger poses several challenges and questions, such as what types and transitions can be animated, when to display these animations, and how this will impact the template creation process.

VI. Related work

Many projects generate object visualizations and algorithm animations. Earlier tools [16], [14], [19] were limited to visualizing data structures in box forms, showing the values of basic data types like numbers as boxes, and pointers as arrows between the boxes.

Hendrix et al. introduced extensible visualizations for data structures in a lightweight Java IDE called jGRASP [10]. Their work focuses on the educational value of the visualizations, while Vebugger is a general-purpose tool targeting developers. For example, Cross et al. [7] conducted a study on university students using jGRASP, showing that students learn data structures and algorithms more efficiently using a visual debugger. We believe that the same result applies to developers. Demetrescu et al. introduced MONNALISA [8], a toolkit to create charts and other related visualizations for program runtime, either event-driven or data-driven. MONNALISA is more focused on visualizing statistical information about the state of the world. Alsallakh et al. [3] introduced an Eclipse plugin to visualize the content of arrays and collections in Java by exposing the objects' fields in a table, displaying line charts and histograms of selected fields. We used Vebugger to reproduce the same visualizations. However, as discussed in Section V-D, Vebugger lacks the interactivity that exists in Alsallakh's tool.

Mellis [13] introduced Tangible Code, an environment that, among other features, includes "live functions", a feature that inlines the values of variables in the source code view. Similar to Vebugger, Tangible Code assists in exposing the developer to information faster, albeit in a completely different manner. Ko et al. [12] introduced a more complex program analysis tool called Whyline; a debugging tool that allows developers to debug code by asking questions about the output rather than tracing the flow of data manually. Like Vebugger Whyline eases the cognitive burden on developers by exposing information that a developer would otherwise need to infer by slowly exploring the state of the world and its relation with the code.

The field of object visualizations is related to the field of algorithm animations, with many papers discussing the two fields interchangeably. Beck et al. [4] introduced a method to visually monitor changes to numeric variables embedded inside the source code view. Stasko described Tango [18], a framework for iterative creation of animations on arbitrary programs. Tango provides developers with an expressive way to define visualizations for transformations in their program's state to help them increase their comprehension of their algorithms. Codea [2] is an iPad application that uses many software visualization and mobile interaction techniques to make game programming more accessible and fun. Visual Studio [1] includes Visualizers, a method to provide debug-time object visualizations similar to Vebugger. A distinguishing feature is that Visualizer templates are defined using high level languages such as C#. We think Vebugger's choice of using HTML, a presentation language, is more suitable for exploring auto-generation of templates and lowers the bar for custom user-developed templates.

Eisenberg et al. [9] proposed presentation extension, which is distinct from semantic extension and provides a general mechanism to extend the IDE through a metaobject protocol. Vebugger is a kind of a presentation extension.

VII. CONCLUSION

Software is difficult to develop and maintain. One important reason for this is the invisibility of software, yet most IDEs continue to represent software state textually. Developers often think of program state more abstractly. To support developer software comprehension we designed Vebugger, an IDE plugin that displays object state visually. Vebugger uses HTML and CSS templates to visually represent Java types. Besides describing Vebugger we also touched on the various promising research directions for future work. Vebugger is available for download [17].

ACKNOWLEDGMENTS

We thank Eric Wohlstadter who supported Vebugger as a class project. We also thank the reviewers for their feedback along with Yuriy Brun, Gail C. Murphy, and Marc Palyart who have offered advice on earlier drafts. This work was partially supported by NSERC and the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

REFERENCES

[1] Visual Studio. http://www.visualstudio.com/, Accessed July 30, 2014.
[2] Codea – iPad. http://twolivesleft.com/Codea/, Accessed July 8, 2014.
[3] B. Alsallakh, P. Bodesinsky, S. Miksch, and D. Nasseri. Visualizing Arrays in the Eclipse Java IDE. In *CSMR*, 2012.
[4] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. In *VISSOFT*, 2013.
[5] F. P. Brooks Jr. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, 1987.
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.
[7] J. H. Cross II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *TOCE*, 9(2):13, 2009.
[8] C. Demetrescu and I. Finocchi. A data-driven graphical toolkit for software visualization. In *VISSOFT*, 2006.
[9] A. D. Eisenberg and G. Kiczales. Expressive Programs Through Presentation Extension. AOSD, 2007.
[10] T. D. Hendrix, J. H. Cross II, and L. A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. *ACM SIGCSE Bulletin*, 36(1):387–391, 2004.
[11] C. Kehoe, J. Stasko, and A. Taylor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *IJHCS*, 54(2):265–284, 2001.
[12] A. J. Ko and B. A. Myers. Debugging reinvented. In *ICSE*, 2008.
[13] D. A. Mellis. Tangible code. Master's thesis, Interaction Design Institute Ivrea, 2006.
[14] S. Mukherjea and J. T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *TOCHI*, 1(3):215–244, 1994.
[15] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *Software, IEEE*, 23(4):76–83, 2006.
[16] B. A. Myers. INCENSE: A system for displaying data structures. In *SIGGRAPH*, 1983.
[17] D. Rozenberg and I. Beschastnikh. Github – Vebugger. https://github.com/daniboy/vebugger, Accessed July 8, 2014.
[18] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, 1990.
[19] A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.