DoodleDebug, Objects Should Sketch Themselves For Code Understanding

Niko Schwarz
University of Bern

ABSTRACT

Developers override `toString()` and `printOn:` methods to allow objects to display themselves. This is done to track object state while debugging. Although very popular, the technique breaks down when displaying complex, multi-dimensional objects. We propose an approach in which objects have two-dimensional visualizations at various levels of granularity. This makes it easier to compose visualizations from object parts, and enables "semantic zooming" of object visualizations while debugging. We have carried out an empirical study to understand how `printOn:` methods are used in practice, and we are developing DoodleDebug, a framework to support visualizable objects.

1. INTRODUCTION

In a survey on software visualization among 111 researchers in software maintenance, reengineering and reverse engineering, 40 % of the researchers found software visualization absolutely necessary for their work [9]. However, when 31 developers were observed, each for 70 minutes while performing software maintenance tasks, not a single one of them is reported to have used any kind of visualization whatsoever [7]. At the very least, current visualizations haven't become an integral part of the professional developer's toolkit, as Reiss [12] laments. On the other hand, the use of textually visualizing objects via the `toString` (in Java) or `printOn:` (in Smalltalk) method and equivalent facilities in other languages, is widespread.

We attempt to explain the popularity of the `printOn:` method in Smalltalk. We present the results from looking at 2145 open source projects, of which 590 overwrote the `printOn:` method.

A downside of the `printOn:` method is that it makes it hard to compose visualizations of objects. We outline our idea of DoodleDebug, our vision of a library that improves on the `printOn:` method by making it easy for developers to specify how their objects can be shown as 2-dimensional drawings. To delve deeper into an object's representation, we suggest using semantic zooming [14].

In section 2, we present the results of looking at 590 software repositories that use the `printOn:` method for object visualization. In section 3, we attempt to explain the success of this way of visualizing objects, and collect requirements that any improvement would have to meet. In section 4, we outline our proposed improvement over merely textually visualizing objects. In section 5, we list the related work and in section 6, we conclude.

2. CURRENT TEXTUAL VISUALIZATION

The textual visualization provided by the `printOn:` method is used in two sets of circumstances—in print statements to the transcript, or as a shortcut when viewing objects in a debugger, reducing the need to delve too deep into the object graph.

To get a feeling for how this feature is used, we downloaded all 2711 projects from the Squeaksource open source software repository[1]. Of these, we were able to extract the latest sources of 2145 of these repositories. The remaining repositories were either empty, or unreadable (some of the zip files holding the sources appear to be damaged).

Of the 2145 repositories that we analyzed, 590 overwrite the `printOn:` method, amounting to 28 % of all projects. In total, we found 2688 overwritten `printOn:` methods. We then extracted some basic properties from those 2688 overwritten `printOn:` methods, to get a feeling for what they were used for. We used a mix of scripts and manual inspection.

The average method length we observed was 7.1 lines, although a number of these methods seemed to call private methods in the same class to help with textually visualizing the class's objects.

Of the 590 `printOn:` methods,

- 44 already print a two-dimensional structure of themselves as ASCII graphics, separating the output into several lines which contain key-value pairs, or even tabular information[2]. Examples include classes that model matrices and addresses.
- 137 used parentheses to mark their contained instance variables. However, not all instance had to be printed. Also, we saw extra measures to ensure the brevity of every string's length.

[1] `http://squeaksource.com`
[2] We obtained this number by automatically selecting methods that print newlines, and then manually inspecting those. Other results were obtained similarly.

- 219 called the implementation of printOn: in their super class, which usually just prints the class name.
- 233 directly print their own class names.
- 452, or 77 % of all `printOn:` methods, included their class name. This suggests that the dominant and most important kind of information of interest about an object is its class, present in most instances.

From the study on the usage of `printOn:`, we learned that developers value concise textual descriptions (because of the care to only have short textual descriptions). And they value cheap to produce textual descriptions (since the lines of code were comparatively short).

3. ADVANTAGES OF SUMMARIZING OBJECTS IN TEXT

Especially in a programming environment like Smalltalk, which usually features a perfectly workable inspector, it is interesting to ask why textual visualization is still so popular. While visualizing an object as a short string of text does not use much of what is available to software visualization, it still reaps a number of benefits commonly attributed to more advanced software visualizations: it makes use of the pattern recognition abilities of the human brain, and can summarize a wide array of information in a concise format.

To help us shape DoobleDebug, we try to find the information needs that cause developers to overwrite the `printOn:` method. Ko et al. [5] observed 17 developers over the course of two months for a total of 25 work hours, noting the information needs that arose. He lists a total of 21 information needs, of which the following three can be attended to by overwriting the `printOn:` method:

1. What code caused this program state?
2. What does the failure look like?
3. In what situations does this failure occur?

"Program state" is explicit in the first question, and implicit in the other two. This leads us, combined with our experience, to the following hypothesis:
The strengths of overwriting the `printOn:` method as a textual visualization technique are the following:

1. Great numbers of snapshots of an object can be printed, so that the repeated printing of the same object visualizes the course of a program. The changes from one printout to the next can show the developer which stages occurred during computation. If the developer is not sure at which point in time an object may be interesting to inspect, he can print the object at all possible times, and then quickly scan over the transcript to find the stage that he is interested in.
2. Textual visualization integrates well into current IDEs. Default debuggers and inspectors are aware of the `printOn:` method and use it to visualize objects. To share a visualization, the developer only has to share code, which is part of his usual process. The overall effort for creating and sharing a visualization is therefore small.
3. Textual printouts can be easily exchanged, save for encoding problems. They are easy to compare, and can be used in bug reports, questions on community boards, or even in unit tests to assert correct behavior.

To understand these upsides, consider not overwriting the `printOn:` method and solely relying on a debugger that can only show basic data types. Since debugging race conditions without the help of print statements seems nearly unthinkable, print statements for debugging race conditions might have to become considerably longer, because they have to print all basic data types at every point of interest. If a developer is unsure at which point in time a certain line of code is interesting, setting a breakpoint would require him to proceed manually until the relevant moment is found. Once it is found, he has nothing to compare it to, since all previous moments have already passed without being displayed in the debugger. Unable to show the history of an object, current debuggers are unable to explain how these states came about. This is even though some research projects aim at making the past more accessible to the developer. [10, 6, 11]. However, none of these approaches provides what developers now get from printing to the transcript: a quick overview over a possibly large number of machine states, together with an easy way to compare those side-by-side.

4. DOODLEDEBUG

In this section, we propose an approach to satisfying the information needs attended to by the `printOn:` method, at a similarly low cost to the user, but with improved usefulness. Our approach is informed by the success of the `printOn:` feature and tries to add to it, without losing its upsides. We discuss directions in which this research could continue.

We intend to allow every object to overwrite the `drawOn:`, and `drawSmallOn:` methods, specifying its own 2D drawing, in a simple DSL. A default implementation is provided which simply inserts the current string representation.

When overwriting the `drawOn:` method, the user can therein choose from a set of simple data layouts, or draw directly on the canvas. However, the canvas must be quite unlike current graphical canvases, which are pixel-accurate. As we have found above, developers value visualizations that are cheap to produce. It is therefore imperative to allow the user to express his visualization wish in the simplest, most abstract terms, while still generating useful visualizations. The vocabulary of the canvas must thus be very coarse, as in: "draw i as an external resource", where i is an instance variable. It is then up to the canvas to choose a location and visual indication for external resources.

Since objects may be composed of other objects, they may need to draw these when drawing themselves. To summarize objects to what is needed from an abstract level, we provide the `drawSmallOn:` method, which can draw a shorter summary, which can then be zoomed in on by clicking it. This kind of zooming is called semantic zooming [14].

4.1 Example

Let us suppose that the developer is constructing a number of affine transformations, which are then concatenated. Since the concatenated affine transformations are not as the developer expects, he wishes to see all of them, at various stages of the course of the program. In the Pharo Smalltalk IDE, he could now choose to open an inspector for the array of affine transformations in an appropriate line of code in his program. This would open a plethora of inspector windows with no clear ordering, making it impossible to use them for tracing the program state. As an alternative, he could set a break point in the same line, looking at the array of affine

transformations on each step. Or, he could write the array to the transcript on every iteration. Since an affine transformation is best described as a 2×3 matrix, and since it seems the least awkward option to use the transcript, he may there find output as seen in Figure 1.

```
an Array()

an Array(
MatrixTransform2x3(
    2.0 0.0 0.0
    0.0 2.0 0.0
))

an Array(
MatrixTransform2x3(
    2.0 0.0 0.0
    0.0 2.0 0.0
) MatrixTransform2x3(
    0.707107 -0.707107   0.0
    0.707107 0.707107    0.0
))
```

Figure 1: Textual visualization of an array of affine transformations. They can only be listed vertically. The affine transformations represent a scale by 2, followed by a rotation by $\pi/4$

The default visualization of arrays and affine transformation in Pharo Smalltalk shows matrices in their 2-dimensional structure. However, it cannot show the two matrices next to each other, since when nesting strings that contain newline characters, 2-dimensional structures cannot always be nested. Instead, they are shown vertically aligned. Thus, when we print the course of our program by printing three stages of the construction of the array, as seen in Figure 1, there is almost no visual clue as to where one step of the program ends and the next one starts. Even worse: comparing states of the program is not easy, because the visualization does not help us map the individual matrices from state to state of the program.

In contrast, if we allow matrix objects to visualize themselves non-textually, several matrices can be fit on one line. Our developer could, for similar effort, see the visualization in Figure 2. In this new visualization, with every program state being exactly one line, it is visually clear where each print begins and ends. Since the scaling matrices appear vertically aligned, we can easily see that from the second to the third state, the scaling matrix was not changed, but a rotation matrix was added.

To achieve this result, all we need is the composition of two simple data layouts. One horizontal layout, so that the array can print its contents truly horizontally, even if it contains 2-dimensional entries. And a table layout, which displays nested arrays as tables. We are hopeful that similar improvements will be cheap for developers in a wide range of situations.

4.2 Future directions

Having objects that specify their visual layout opens the path for new research possibilities. Once objects know how

```
{}

{ 2.  0.  0.
  0.  2.  0.
  0.  0.  2. }

{ 2.  0.  0.
  0.  2.  0.
  0.  0.  2. ,  0.707107  -0.707107  0.
                0.707107   0.707107  0.
                0.         0.        1. }
```

Figure 2: If objects defining their own visualization are not restricted to text, nesting 2-dimensional structures becomes possible.

to sketch themselves, we can use the instructions that define the drawing to determine a raw outline of what matters for the developer. Using it, we can construct a simplified data model of an object, which may be helpful for comparing different versions of that object, so helping understand the history of a program's state.

The next step is to make the visualizations more interactive. For example, right-clicking on an object should open its class, so we can navigate from visualization to code. Dragging an object into a visualization should assign it to the slot represented by the location where it was dropped.

We have learned in Section 3 that one key advantage of using `printOn:` is the easy shareability of the textual output of objects. We intend to use the drawing instructions to create textual descriptions of the object to be visualized to improve shareability.

If properly exploited, we believe that the zooming metaphor has the potential to get rid of the need for a separate inspector altogether. If, for example, every doodle is accompanied by a list of instance variables at the current zoom level, then zooming in is just like diving into the object graph using an inspector, making the ladder superfluous. Some experimentation will be needed to answer whether doodles can entirely take over object inspection.

5. RELATED WORK

GNU DDD, the Data Display Debugger [15] uses visual metaphors to give insight into the state of a program. To that, LIVE [1] adds metaphors with animations, although empirical results as to the helpfulness of animation in understanding algorithm is mixed [4]. JGRASP [3] and Travis [8] provide automatic mappings between data and its visualization.

Cheng et al. provide the xDIVA tool [2], which provides a number of mappings from data types to "visualization metaphors" which render the data. Since data types can be nested, visualization metaphors can be composed. for a number of reasons, however, the visualizations are difficult to specify. First, xDIVA is a separate tool, rather than being integrated into the IDE, like `printOn:`. Second, the visualizations are configured in a graphical interface, rather than in the code of a class, and thus can not easily be exchanged with members of the same team. Third, the visualizations are a great deal more complex than we think is necessary. The visualization in xDIVA is 3D, allowing for circling around objects of interests. This complexity likely increases the complexity of creating new visualizations.

Mathematica [13] ships with some data layouts for tables, matrices, and curves. However, as Mathematica is not object-oriented, objects cannot define their own behavior, and using any but the predefined commands for viewing data is uncommon. Further, Mathematica does not allow data layouts to be composed.

6. CONCLUSION

Developers overwrite the `printOn:` method because they need to understand what caused a certain program state, what caused a failure, and what exactly a program failure looks like. They use textual visualizations because they give a quick overview over a possibly large number of machine states, together with an easy way to compare those side-by-side. Textual visualization of objects helps in using the debugger and inspector. It simplifies printing program states to the transcript, as is useful, among other things, for debugging race conditions. Textual visualizations integrate well into the development processes, are cheap to produce and allow easy exchange with developers. We thus account for their tremendous adoption among developers.

We outline DoodleDebug, a suggested research prototype which allows the developer to better compare program states than `printOn:` allows. DoodleDebug sacrifices some of the easy sharing of program states that strings allowed, in exchange for a denser view of information, by allowing the nesting of 2-dimensional structures. By allowing the user to zoom into doodles, we hope to receive a new way of exploring the data during the running of his program. We hope to achieve the same degree of integration into the IDE and the same cheap production cost as textual visualization tools.

Acknowledgements

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper. We thank Adrian Kuhn, Cedric Reichenbach, Oscar Nierstrasz and Edouard Tavinor for their input to this paper.

7. REFERENCES

[1] Alistair E. R. Campbell, Geoffrey L. Catto, and Eric E. Hansen. Language-independent interactive data visualization. *SIGCSE Bull.*, 35:215–219, January 2003.

[2] Yung P. Cheng, Jih F. Chen, Ming C. Chiu, Nien W. Lai, and Chien C. Tseng. xDIVA: a debugging visualization system with composable visualization metaphors. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA Companion '08, pages 807–810, New York, NY, USA, 2008. ACM.

[3] James H. Cross and T. Dean Hendrix. jGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. *J. Comput. Small Coll.*, 23:170–172, December 2007.

[4] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of algorithm visualization effectiveness.

[5] Andrew J. Ko. Information needs in collocated software development teams. In *in International Conference on Software Engineering (ICSE 2007*, pages 344–353, 2007.

[6] Andrew J. Ko and Brad A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[7] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng.*, 32(12):971–987, December 2006.

[8] J. L. Korn and A. W. Appel. Traversal-based visualization of data structures. pages 11–18.

[9] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *J. Softw. Maint. Evol.: Res. Pract.*, 15(2):87–109, 2003.

[10] A. Lienhard, S. Ducasse, and T. Girba. Taking an object-centric view on dynamic information with object flow analysis. *Computer Languages, Systems & Structures*, 35(1):63–79, April 2009.

[11] Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. Executing code in the past: efficient in-memory object graph versioning. *SIGPLAN Not.*, 44:391–408, October 2009.

[12] S. P. Reiss. The paradox of software visualization. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–5. IEEE, 2005.

[13] Stephen Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, 5th edition, August 2003.

[14] Allison Woodruff, James Landay, and Michael Stonebraker. Goal-directed zoom. In *CHI 98 conference summary on Human factors in computing systems*, CHI '98, pages 305–306, New York, NY, USA, 1998. ACM.

[15] Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.