Visual Monitoring of Numeric Variables Embedded in Source Code

Fabian Beck*, Fabrice Hollerich†, Stephan Diehl† and Daniel Weiskopf*
*VISUS, University of Stuttgart, Germany
Email: {fabian.beck,weiskopf}@visus.uni-stuttgart.de
†University of Trier, Germany
Email: {s4faholl,diehl}@uni-trier.de

***Abstract*—Numeric variables are one of the most frequently used data types. During the execution of a program, their values might change often. Tracing these changes can be necessary for understanding specific behavior of the program or for locating bugs. However, using a breakpoint debugger requires tedious stepping, and logging changes implies analyzing large text files. To make the monitoring of numeric variables easier, this work introduces a visualization approach that augments the source code view of an IDE by small, word-sized graphics: the visualizations accompanying the declarations of monitored variables plot read and write accesses on a timeline; detail views can be retrieved on demand. As suggested by a case study, this approach might support program comprehension and debugging.**

I. Introduction

For understanding the runtime behavior of a program, it is important to follow the executed data transitions. While stepping through the execution using a breakpoint debugger provides insights into fine-grained transitions, an overview is hardly available. Only logging mechanisms, which usually need to be set up manually, provide a method for recording transitions over the full runtime of a program. However, large text output generated by logging needs to be further processed to finally obtain an overview. Hence, there is a need for novel approaches to monitoring data transitions and presenting these to software engineers in a meaningful and accessible way.

In this work, we focus on monitoring numeric variables, which are—e.g., as native data types in Java—one of the most frequently used types of variables. While it is complicated to understand the state of a complex data structure, the state of a numeric variable is described just by a single value. The main challenge is that this value changes over time. Developers might need to understand this time series in order to understand what the program is doing (wrong). In addition to assignments of new values to a variable, it is also interesting to see how often a specific value is accessed by the program after assignment.

This kind of time series can be presented in an additional view [1]. However, all numeric variables are already represented in source code by their declaration. Having a separate view requires introducing additional representations and, when multiple variables are displayed, may require the developer to search for the relevant ones. When trying to make sense of the evolution of a variable, the developer has to switch back and forth between the source code and the visualization. Even if both views—source code and visualization—are visible at the same time and synchronized, this causes at least an unwanted *split attention effect* [2].

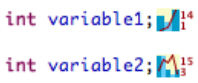

Fig. 1. Word-sized graphics for monitoring numeric variables.

To avoid these negative effects and unintentional complexities, we propose an in situ visualization as illustrated in Figure 1: small, word-sized graphics are embedded into the source code view of an integrated development environment (IDE). Each visualization accompanies a declaration of a numeric variable and plots related read and write accesses over time. Details can be retrieved on demand by tooltips showing full-sized visualization of the time series. Our prototype of the approach is implemented as an Eclipse plug-in for watching numeric class and instance variables in Java programs.

II. Related Work

Related research falls into two areas: augmenting the source code by embedded visualizations and the visualization of program behavior. While both areas have already been studied separately, our approach combines the two areas and is—to the best of our knowledge—the first to use embedded visualizations for monitoring data transitions.

A. Visually Augmented Code

Word-sized data visualizations are also known as *sparklines* [3] and gain more and more popularity as integrated parts of spreadsheet software such as Microsoft Excel. Often, line or bar charts are represented although the original definition is open to every kind of *"data-intense, design simple, word-sized graphics"* [3]. While sparklines were originally used for text and tables, they can be embedded into source code as well: Beck et al. [2] use variants of sparklines for encoding the runtime consumption of methods and how this consumption propagates through the call graph. To this end, method declarations and method calls are augmented with small graphics that provide a preview of the runtime consumption, calls, and threads within the code view.

Code can also be visually augmented in other ways: for instance, Murphy-Hill and Black [4] integrate a glyph-based visualization of software metrics into a code editor for detecting so-called *code smells*; Harward et al. [5] present a framework for mapping software metrics to various visual properties of the source code view. Swift et al. [6] discuss the use of in situ visualization for the purpose of live programming. In general, we have been witnessing that IDE developers introduce more and more small features enriching the source code view, for instance, warnings or previews of search results.

B. Visual Debugging

Analyzing the state changes of a program recorded during execution is a post-mortem debugging technique known as *tracing* [7]; in turn, visualizing this data can be considered a dynamic program visualization technique [8], which is a specialization of visualizing time-dependent data [9]. There exist some techniques that provide an overview of a complete execution of a program: for instance, web service transactions [10], stack traces [11], or method call graphs [12] can be represented on a timeline. Specifically for tracing numeric variables, Alsallakh et al. [1] visualize the variable history as a timeline view in the IDE.

Other visual debugging techniques concentrate on visualizing a single program state such as the current memory state [13]. In breakpoint debuggers of IDEs, the value of numeric variables can be accessed in variables views or as tooltips hovering the variable. One step further, Mellis [14] proposes to blend in the values directly into the source code. The Whyline debugging tool [15] helps answer the question why a specific value was assigned.

III. Visualization Approach

When monitoring the state of a numeric variable, of course, its value and how it changes over time is of interest. The changing value creates a time series for each instance of a variable. Computing a new value for a variable, however, is not an end itself, but mostly just represents an intermediate result as a part of a more complex computation. Considering this, accesses of the variables are also important. The accesses create an additional series of points in time for each instance of a variable. Comparing these to the original time series of changing values, each access can be mapped to a specific value; the number of accesses accumulates per variable value. Hence, variable changes and the number of their accesses form two aligned time series that will be visualized at each declaration of a numeric variable as described in the following. For simplification, we first assume that only one instance of a variable exists; multiple instances are discussed later.

A. Sparkline Visualization

The two time series are visualized as two sparklines in the same graphics as depicted in an enlarged and annotated example in Figure 2. Both are mapped to the same timeline (from left to right) and overlaid. The timeline represents logical time instead of physical time, i.e., each change of a value counts as a new time step and time steps are equally spaced on the timeline. The reason for this kind of design is that variable changes usually are distributed very unevenly across the runtime of the program—using physical time for the timeline would produce visual clutter restricting the readability of the diagrams.

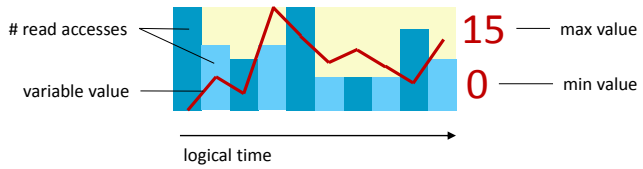

Fig. 2. Sparkline visualizing the changes and accesses of a numeric variable.

The sparklines are drawn on word-sized rectangular areas having a bright yellow background. To make the two time series discernible, different colors as well as different visual metaphors are used for their representation:

- The change of the variable value is visualized as a **red line graphic**. The minimum value and maximum value are aligned with the top and bottom of the graphics; the two extreme values are additionally shown as small-font numbers at the right side of the graphics.
- In contrast, the number of variable accesses are represented as **blue bar charts**. The bar chart metaphor is suitable because accesses are counted in natural numbers while variable values can take any real-valued number. To make neighboring bars more discernible, the color of the bars is varied in alternating brightness. The bars are aligned at top and bottom of the diagram, but on a scale independent of the scale for the variable values.

This sparkline visualization is added after the last character of a line of the respective declaration of the variable. Although it is currently assumed that only one variable is declared per line, the approach is extensible to multiple declarations per line when placing the visualization directly behind the identifier instead.

B. Details on Demand

A sparkline can only act as a preview of the precise time series: visual elements are too small for reading details such as specific values; further, there is no space for adding labels. Hence, we allow users to retrieve details on demand: we propose showing a larger version of the graphic in a tooltip when the user hovers over the sparkline with the mouse. The enlarged version (see examples in Figures 3, 4, and 5) may additionally include labeled axes that enable the user to read the specific variable values and numbers of accesses.

C. Multiple Instances

For class variables (i.e., fields declared as `static`) or instance variables of *singleton* classes [16] (i.e., classes that are designed to only have one instance), indeed only a single copy of the variable exists. For general instance variables and local variables, however, usually multiple copies are created

during runtime. Hence, for a declared variable in the source code, more than one sparkline may be generated. A simple and straightforward solution for representing those multiple instances is to just display a single (or a few) sparkline(s) in the code and provide the complete list of instances only on demand. As shown in Figure 3, this can be easily combined with the enlarged view of the diagrams in the same tooltip where the user may switch between the instances. The sparklines in the list view help quickly find the most interesting instances.

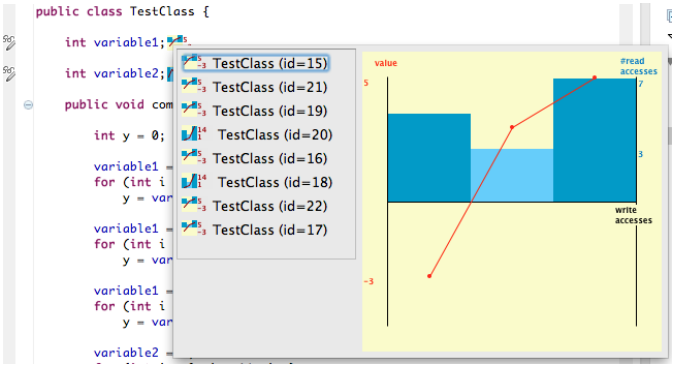

Fig. 3. A list of multiple instances of a variable available on demand.

An extension of the approach, which is not yet implemented in our prototype, would be to use techniques from temporal data mining [17] to analyze the time series data before visualization. In this way, a more compact visual representation that might filter out uninteresting raw data is conceivable. In particular, clustering of time series [18] would allow us to find a few representative diagrams that could be used to show the full set of raw time series plots. In the related field of time series data in (geo)-spatial context, clustering and aggregation is successfully applied to the visualization of many and complex time series [19]. Similar strategies might be useful for the time series of numeric variables. Other interesting approaches from data mining include pattern recognition and outlier detection. Oftentimes, (hidden) Markov chain models, Markov field models, or variants thereof serve to model time series. Such models can be used for the visualization of outliers and anomalies [20]. Similarly, anomalous time series could be extracted and visually highlighted in our sparkline representation to support the user in detecting potential bugs in the code.

IV. Implementation

A first prototype of the approach is implemented as a plug-in for the IDE Eclipse for monitoring Java programs. It uses the already available functionality of Eclipse to set watchpoints for class and instance variables; watchpoints for local variables are not available in Eclipse, hence not considered so far by our tool. Normally, watchpoints are used for debugging: they pause the program whenever the value of the variable is changed (or accessed). Deactivated watchpoints do not stop the program but still can be traced by a background process. Catching those events, our plug-in stores all changes to numeric variables currently covered by (deactivated) watchpoints similar as in EclipseTracer [1]. After completing a run of the analyzed program, the described visualizations are blended into the source code view of Eclipse.

V. Application

While our sparkline visualization can be used for any analysis of a program execution, we see two particularly promising applications as described in the following.

A. Program Comprehension

Our approach can be considered a generic algorithm visualization technique focusing on numeric variables. As such, it can be applied to understand the inner workings of algorithms for educational purposes or for familiarizing with an unknown piece of code. The example provided in Figure 4 shows two numeric variables used for an execution of a bubble sort implementation:

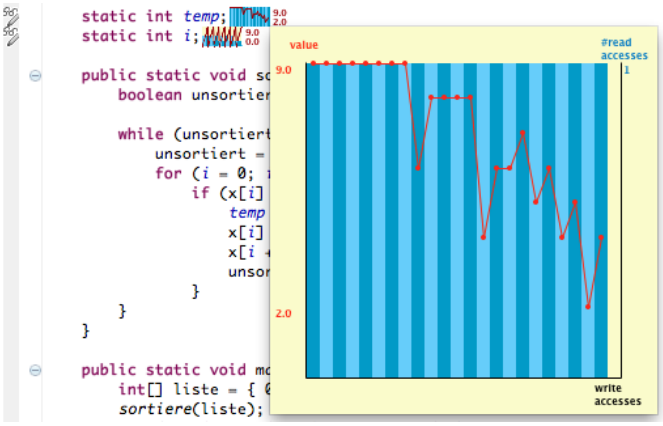

Fig. 4. Monitoring numeric variables for understanding a *bubble sort*.

- Variable `temp` is used for swapping the values that should be sorted. The visualization shows that, by chance, the maximum value had been in the first position of the array in the initial order. The value was swapped to the end in eight transitions. In the second iteration, at first, a smaller value was swapped until a new maximum was discovered. Hence, moving the current maximum in a *bubble* toward the end of the array creates a jagged curve: every rising flank represents a new maximum value; every falling flank marks the beginning of a new iteration.
- Variable `i` (only depicted as a sparkline in Figure 4), being the loop counter, shows the iterations even more clearly. While six iterations can be counted for the `temp` variable, we observe seven iterations for `i`—the last iteration is needed to confirm that everything is sorted already. The number of read accesses of `i` decreases in each iteration. Moreover, the visualization reveals that the bubble sort implementation is not yet optimal: `i` is always increased from 0 to 9 although it can be assumed for later iterations that the last elements are already in the right position and do not need to be checked anymore.

For this example, the visualization not only helps understand the execution of the algorithms but also supports detecting weaknesses in its implementation. All relevant information can be already retrieved from the sparklines, but the enlarged versions are easier to read.

B. Debugging

In the second example, we address the application of debugging. Assumed that a defective behavior of the program was observed for a specific execution, our visualization approach helps check whether the numeric variables of the program showed illegal states or suspicious behavior. The program depicted in Figure 5 was intended to count the frequency of a given character and a given trigram in a string using the variables `counter` and `counter2`. The final state of `counter`, however, always is 0 independent of the input. Analyzing the respective diagram shown enlarged in Figure 5, we observe that the variable seems to be increased correctly but is set to 0 in the end. Moreover, the previous value, which is the correct value, was not accessed at all. Looking at the code, it can be confirmed that the final assignment of 0 is superfluous; deleting it fixes the bug.

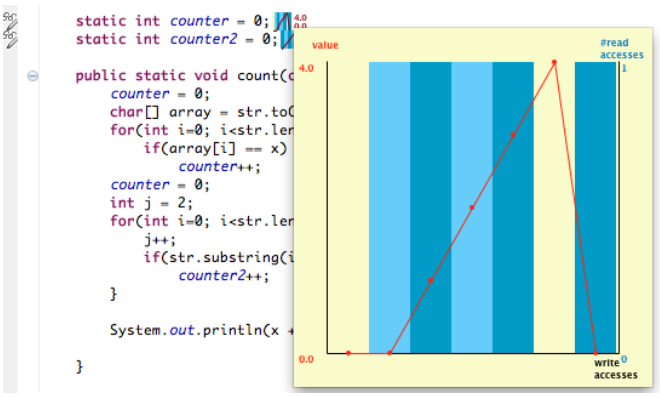

Fig. 5. Debugging a program to count characters and trigrams in a string.

This is only an illustrating example. However, we assume that comparable errors are often hidden in more complex program code and might be discovered quickly with the help of visualization. In particular, programs that handle complex numeric data such as simulation software (e.g., for computational physics or engineering) are promising areas of application for our approach.

VI. Conclusion and Future Work

We have presented an in situ visualization technique augmenting the declarations of numeric variables in source code views. For an execution of the analyzed program, the visualization plots write and read accesses of the variables as two overlaid sparklines. Details and information on multiple copies of a variable are available on demand. As suggested by our case study, this approach can help monitor numeric variables in context of program comprehension and debugging.

As part of future work, the approach needed to be evaluated in a user study: a controlled experiment comparing the augmented code to a non-modified version of the IDE as well as to a modified version where the time series are presented in an additional view; the hypothesis is that the embedded visualizations help understand algorithms and fix bugs more efficiently. Further extensions of the approach could be explored such as the proposed clustering techniques or comparing multiple runs of a system. Maybe, it is also possible to develop similar time series visualizations for more complex data types like arrays, collections, or tree structures.

References

[1] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch, "Visual tracing for the Eclipse Java Debugger," in *CSMR '12: Proceedings of the 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 545–548.

[2] F. Beck, O. Moseler, S. Diehl, and G. D. Rey, "In situ understanding of performance bottlenecks through visually augmented code," in *ICPC '13: Proceedings of the 21st IEEE International Conference on Program Comprehension*. IEEE, 2013.

[3] E. R. Tufte, *Beautiful Evidence*, 1st ed. Graphics Press, 2006.

[4] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *SOFTVIS '10: Proceedings of the 5th International Symposium on Software Visualization*. ACM, 2010, pp. 5–14.

[5] M. Harward, W. Irwin, and N. Churcher, "In situ software visualisation," in *ASWEC '10: Proceedings of the 21st Australian Software Engineering Conference*. IEEE Computer Society, 2010, pp. 171–180.

[6] B. Swift, A. Sorensen, H. Gardner, and J. Hosking, "Visual code annotations for cyberphysical programming," in *LIVE '13: Proceedings of the 1st International Workshop on Live Programming*, 2013.

[7] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *CASCON '04: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 2004, pp. 42–55.

[8] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*, 1st ed. Springer, 2007.

[9] W. Aigner, S. Miksch, W. Müller, H. Schumann, and C. Tominski, "Visual methods for analyzing time-oriented data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 1, pp. 47–60, 2008.

[10] W. De Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. F. Morar, "Web services navigator: Visualizing the execution of web services," *IBM Systems Journal*, vol. 44, no. 4, pp. 821–845, 2005.

[11] J. Trümper, A. Telea, and J. Döllner, "ViewFusion: Correlating structure and activity views for execution traces," in *Theory and Practice of Computer Graphics*. Eurographics Association, 2012, pp. 45–52.

[12] F. Beck, M. Burch, C. Vehlow, S. Diehl, and D. Weiskopf, "Rapid serial visual presentation in dynamic graph visualization," in *VL/HCC '12: Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing*. IEEE, 2012, pp. 185–192.

[13] A. Zeller and D. Lütkehaus, "DDD—a free graphical front-end for UNIX debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[14] D. A. Mellis, "Tangible code," Master's thesis, Interaction Design Institute Ivrea, 2006.

[15] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. IEEE, 2008, pp. 301–310.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[17] T. Mitsa, *Temporal Data Mining*. Chapman & Hall/CRC, 2010.

[18] T. W. Liao, "Clustering of time series data–a survey," *Pattern Recognition*, vol. 38, no. 11, pp. 1857–1874, 2005.

[19] G. L. Andrienko, N. V. Andrienko, S. Rinzivillo, M. Nanni, D. Pedreschi, and F. Giannotti, "Interactive visual clustering of large collections of trajectories," in *VAST '09: Proceedings of the IEEE Symposium on Visual Analytics Science and Technology*. IEEE, 2009, pp. 3–10.

[20] Z. Liao, Y. Yu, and B. Chen, "Anomaly detection in GPS data based on visual analytics," in *VAST '10: Proceedings of the IEEE Symposium on Visual Analytics Science and Technology*. IEEE, 2010, pp. 51–58.