A Visual Virtual Machine for Java Programs: Exploration and Early Experiences

Paul Dourish
Dept. of Information and Computer Science
University of California Irvine
Irvine, CA 92697-3425 USA
jpd@ics.uci.edu

Johan Byttner
Dept. Numerical Analysis and Computer Science
Royal Institute of Technology
Stockholm, Sweden

Abstract Software visualization is typically understood to be of value primarily to the developers of software systems. We believe that the same set of approaches offer promise for giving end users a more direct experience of the computation that they use. However, to do so involves solving a number of technical problems. We present some early experiences with VaVoom, a visual virtual machine for Java programs, which aims to address these problems. While VaVoom is also targeted primarily towards software developers, we believe that this technical approach generalizes beyond the domain of programming.

Index Terms Software visualization, programming education.

1. Introduction

As the cost of computing power has marched inexorably downwards, as described by Moore s Law, everyday computing platforms have become increasingly powerful engines for graphical display. As demonstrated by the rapid progress in computer games, graphical abilities that were available only in high-end, specialized devices are now incorporated into every computer sold. It might be expected that, as a result, the graphical experience of everyday computing would also have become richer, and that our interfaces would exploit the advances in graphical processing in conventional hardware platforms. However, as has been repeatedly remarked, most user interfaces continue to rely on a model of interaction that was designed thirty years ago. While it has been continually adapted and refined, today s graphical interface is largely the same as that of the first graphical interactive workstations.

However, not all aspects of computation have been so fixed. Conventional computing platforms certainly have, in some ways, evolved and incorporated more recent ideas. The incorporation of networking, the increasingly complexity of operating systems, the emergence of component-oriented approaches to application development, and other factors in the development of software systems, have been incorporated into modern operating systems and software environments. Our software systems are considerably more advanced than those on which the graphical interface was originally developed.

This leads to an odd conundrum, then. The software systems that people use are vastly more extensive, intricate, and complex than those that we formerly used, but the user interface through which they are to be controlled and understood are no richer or more developed than those of decades ago. User interfaces have failed to keep track of the increasing complexity of software systems.

Our work is directed towards those problems. In particular, we are interested in how to enhance the experience of computation in everyday interaction. How can computation be made visible and manipulable to end users, so that they can more directly perceive and understand the consequences of their actions in an interactive system?

This is a complex and multi-faceted problem, but approach that we have been exploring is to harness visualization techniques and apply them to the problem of visualizing computation. Rather than visualizing data sets or physical processes, our goal is to visualize the computation going on inside conventional computing systems.

1.1. Long-Term and Short-Term Focus

We are following two strategies concurrently, one long-term and one short-term.

The long-term strategy is concerned with providing visual accounts of the action of software systems to the users of those systems. Our intention is not to provide end users with a complete or detailed understanding of the operation of software systems; that would be impossible and inappropriate. However, we want to be able to convey to people some sense of the activity inside the system, hidden away behind abstraction barriers, where the configuration of that activity might have some consequence for what they do and how. For example, being able to gain a sense of the balance between network and disk activity involved in certain activities might provide a user with sufficient information to appropriately organize their activity around the availability of network bandwidth as they move between wired and wireless settings. The primary goal is to make computation directly available to end users, and the intuition — drawing on the experience of visualization in other domains — is that appropriate visual techniques can allow users to derive some degree of meaning from information that would otherwise be too low-level to be of value to them.

As a short-term strategy and technical proof-of-concept, we decided to focus on a particular set of users in the first instance who have a direct need to understand the operation of software systems — software developers, and especially novice software developers. Our short-term strategy is to develop software visualization tools to help developers understand the behavior and operation of the systems they develop. Software developers are, of course, a highly specialized group of users with a completely different set of needs and requirements than end users. However, our goal at this point is largely technical — to determine whether our basic technical strategy is sufficient to achieve real-time visualization of software system behavior.

To this end, we have been developing a visualization environment called VaVoom, for the Visual Virtual Machine. This paper describes some of our motivations, the technical challenges, and some initial experiences and reflections.

2. Our Approach

Again, our long-term goal is to be able to provide end users with a visual experience of computation. Although we are addressing a different audience in the short term, our future vision sets a number of constraints that have guided our short-term design.

First, we are concerned with providing *real-time* information about the behavior of software systems. Our goal is to be able to provide users with an understanding of the behavior of their systems as those behaviors unfold. In particular, in our initial focus on software system developers, this means that our system should provide for concurrent analysis. This is in contrast to the more conventional style of post-mortem analysis supported by standard tracing and profiling tools, or even more advanced tools such as Jinsight [1]. This is especially important when using VaVoom at the development stage, and with interactive systems, since it allows the programmer to observe directly the consequences of particular patterns of interaction with the target program.

Second, we want to focus on the *dynamic behavior* of software systems. Many tools, especially those aimed at novice programmers, aim to provide visual representations of the static structure of programs. This is especially important when teaching object-oriented programming, in which the structural organization of a software system is an even more important component than in procedural or functional programming. Similarly, some visualization tools aimed at large-scale software development, such as

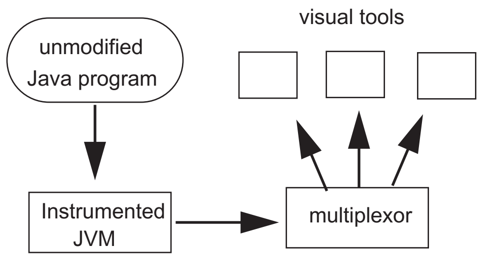

Fig. 1. The basic architecture separates visualization components from the JVM execution engine.

those of Eick and his colleagues [2], are built on top of the static structure (or even the lexical structure) of the system under investigation. Our goal has been to complement existing representations of the static structure of a system with a set of visualizations that can expose aspects of its dynamic structure — how those static features operate in a running program, the relationship between components, the performance of the system while running, etc. These dynamic properties often have a major impact on software performance, but are invisible to software developers.

Third, our approach is based on the use of *multiple, linked visualizations*. Rather than present a single visual representation of all aspects of the software system s behavior, we provide multiple specialized representations which each focus on one particular aspects of what s going on. Users can switch between these at will, or call up new ones at any point. In addition, these visualizations are linked so that different aspects of the system s behavior are visualized concurrently. This is an important characteristic, since it allows users to understand the correlations between different aspects of the system s behavior.

Fourth, and perhaps most importantly for our long-term vision, we set out to visualize *unmodified code*. Our goal is to be able to derive visual representations of existing software systems without access to the source code, and without requiring programmers to annotate or transform their code in any way. This is in contrast with traditional approaches to algorithm animation (e.g. [5]).

3. The Visual Virtual Machine

In order to address these requirements, we have been tackling our problem at a fairly low level. In particular, our strategy has been to visualize the behavior of Java programs by instrumenting and visualizing the actions of the virtual machine on which they execute [3]. We have dubbed our system VaVoom, or the Visual Virtual Machine.

The basic architecture of our system is shown in Figure 1. Java programs run on an instrumented virtual machine, which is configured to send reports of its dynamic behavior to a second process (which may be located on a different machine). This is the visual multiplexor. The multiplexor routes information about the virtual machine to one or more visualization tools. Each visualization tool provides a visual display of some aspect of the system s behavior. A single event reported by the virtual machine might be of interest to multiple visualization tools. Tools can be started and stopped as the program is running. The tools are not simply passive displays, but are interactive, allowing the user to drill down to explore specific features of the behavior of their program.

3.1. Implementation Details

Our starting point was an open-source implementation of the Java virtual machine, called Kaffe. Kaffe can operate both as an interpreter and as a Just-In-Time compiler; our work has focused on the interpreter. Since Java programs are typically subject to dynamic compilation and other advanced adaptive techniques, using an interpreter imposes a significant performance penalty over conventional execution; in addition, of course, our modifications make programs slower still by inserting additional steps in the execution of Java bytecodes. We will discuss performance issues later.

Separating the visualization multiplexor and visualization tools from the virtual machine itself allows us to minimize the impact on the execution engine, to migrate visualizations to other machines, and to isolate us from the specifics of any given virtual machine. Although Kaffe is written in C, the multiplexor and the visualization tools are all written in Java. (In fact, it is impossible for the visualizer to visualize its own behavior, but this is done more for novelty value than for utility.)

One current limitation is that, using a pure interpreter, our implementation cannot handle programs that use the native code extensions of Java (JNI). Unfortunately, recent implementations of Java s Swing UI toolkit rely on some native code (not in Swing itself, but elsewhere in the system). We use a Swing library implementation from an older version of Java (1.1.8), which allows most Swing programs to run under VaVoom without modification. Some recent features such as drag-and-drop, however, are not supported.

4. The Visual Tools

Since our user population, in the short term, is Java programmers, we have been able to design visualizations that rely on standard models and metaphors for the execution of high-level languages. This means that the visualizations are somewhat more literal than would be appropriate for an end-user population. Visualizations must always be adapted to the needs of specific communities, so naturally we do not anticipate using these same models for other groups. However, we are interested in determining, first, the feasibility of our approach, and second, the properties of visual representations that afford different experiences of computation.

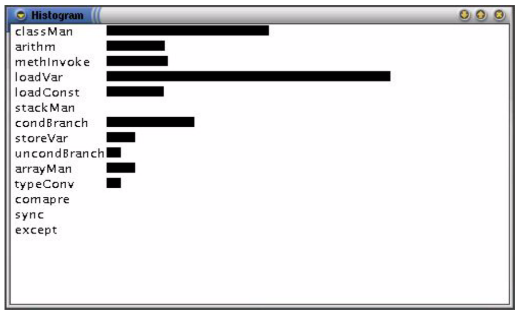

Fig.2. Instruction histogram, showing moment-by-moment distribution of excuted bytecodes.

Our current implementation offers five views of the execution of a program. Each view is outlined below.

4.1. Instruction Histogram

The simplest and most basic visualization, shown in figure 1, is a dynamic instruction histogram. It clusters basic Java bytecodes into simple groups (e.g. load operations, store operations, arithmetic operations, etc.) and displays a dynamic histogram showing the number of each class of instructions executed in the last fraction of a second. Like an audio spectrum display, the pattern of the histogram bars varies according to the nature of the instruction cycle at any given moment.

This visualization was our first demonstration, and is of little value for analyzing real programs. It does have one interesting feature which is invariably remarked on; it demonstrates that an idle program is not idle at all. When a Java program is in an idle state, the histogram nonetheless shows a characteristic pulsing due to the behavior of background threads.

4.2. Method Stack Depth

One of the primary visual displays is shown in figure 2 (left), and displays the method stack depth. Basically, this displays a simple measure of the program s behavior at any given point. Users can watch the stack depth grown and shrink as calls are made and returned.

In addition to the basic display of method depth, a number of other features are incorporated:

1. Each method call entry is color-coded to indicate the degree of locality. One color indicates recursive method calls within a single object; a second indicates non-recursive calls within a single object; a third indicates

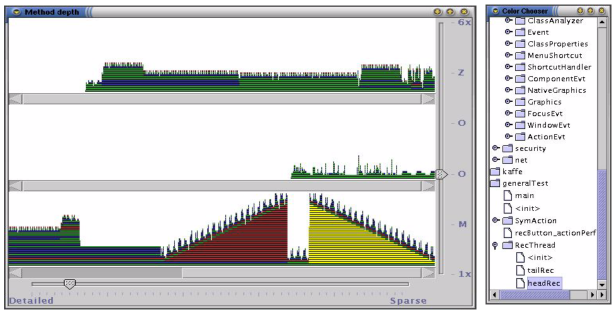

Fig.3. Method stack depth (left) and method call list (right) display the pattern of method calls. The lower thread in the graph window shows tail and head recursion.

calls within different objects of the same class; and a fourth indicates calls between different classes.

2. A filtering mechanism allows for faster or slower display by regulating the frequency with which the data is sampled. This allows programmers to trade off speed for detail.
3. As threads are forked off, the display is split. This allows both comparison between different threads and visualization of thread scheduling.
4. The visual objects representing each call are active objects. Selecting one has two effects. First, it highlights that specific call and all other calls to the same method, allowing programmers to see how calls are distributed through the execution trace; second, it highlights the same call in the method call display (see below) so that the specific method can be identified.

Due to the richness of the display, and its direct relationship to the underlying execution model, the method stack depth graph is probably the central visualization in actual use. The shape of the graph reveals various salient patterns of behavior. For example, the illustration in figure 2 shows three threads; in the lower of the threads, two recursive calls (one tail recursive and one head recursive) can be seen directly.

4.1. Method Calls

The method call display, shown in figure 2 (right), is a simple display that exists largely to accompany the method stack depth display, although it also has some independent functionality. It displays a dynamic list of called classes and maintains measures of the frequency of each method call on those classes. Again, this gives a measure of the program s dynamic behavior rather than its static structure. The method call display and the method stack depth display are linked so that selection in one also highlights the same information in the other, so that programmers can move conveniently back and forth between them.

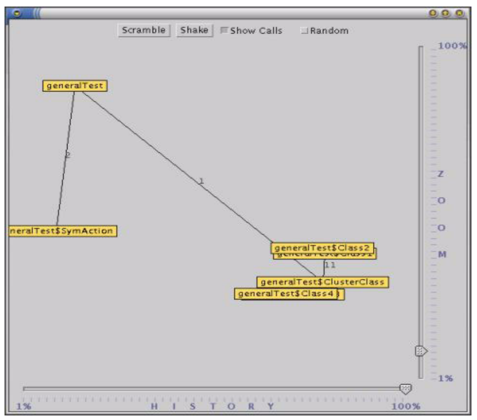

Fig. 4. Class cluster uses a spring model to show the pattern of inter-call relationships between instances of different classes.

4.2. Class Clusters

The class cluster display, shown in figure 3, gives a very different display of the program s behavior, and one less literal than the others. The class cluster display shows an object for each currently instantiated class, laid out in a 2D space. The distribution is controlled by a spring-model simulation in which the strength of the attraction between any two classes is proportional to the frequency with which objects of one class call methods on objects of the other. The result is a display in which classes move together or apart depending on the degree to which they inter-call at any given moment. The effects of inter-calls fade with time, and the slope of this curve can be controlled interactively.

4.3. Instance Count

Finally, a fifth display indicates memory usage, and in particular the current instance count of each class. Using a collapsible tree display, it allows users to select either particular classes or entire packages. The instance count tool is shown in figure 4.

Although this is a seemingly straightforward display, it opens up a number of conceptual difficulties in a language like Java. The first the behavior of the garbage collector, which automatically reclaims unreachable objects. Our initial implementation displays both the current instance count and a high water mark indicating earlier counts which have been reduced by garbage collection. More problematic is the notion of object encapsulation. Earlier incarnations of this tool displayed both instance count and memory footprint, so as to distinguish more easily between a small number of large objects and a large number of small objects. However, this creates a problem in accounting for the ownership of different portions of the process memory. When an instance class A incorporates an instance of class B, should the memory allocated to the instance of B be accounted to class B or to class A? In different circumstances, either may make sense, depending on the program under examination or even the class being examined. Our intention is to examine this issue empirically.

4.4. Controlling the Visual Tools

The visual tools are controlled through a simple interface (not shown here) attached to the multiplexor. This allows users to open or close particular visual tools at will, including multiple instances of the one tool (useful when different instances have been configured to show different aspects of a program s behavior). The emphasis is on allowing the user to explore the visual representations, in the course of the program s execution. In the case of interactive programs, this can be particularly valuable.

5. Experiences with VaVoom

We have not yet (at the time of writing) conducted a formal evaluation of Vavoom. We have conducted a number of informal feedback sessions with programmers of different degrees of experience, in order to get some general feedback. A number of interesting issues arose in the course of these procedures, which influence our future work.

One of the first issues that we encountered was one of program size and complexity. Since our initial focus was on novice programmers, our first contact was with students enrolled in introductory programming classes. However, in general, we found that their programs were often too small to be effective in our visualizations. There are a number of reasons for this. The first is that, since VaVoom visualizes all Java behavior, the noise introduced by start-up procedures, concurrently running threads, and other aspects of the background Java virtual machine behavior can dominate small programs. Another related reason is that our visualizations concentrate on dynamic effects, whereas novice programmers often struggle more with basic structural features of programs rather than more complex or

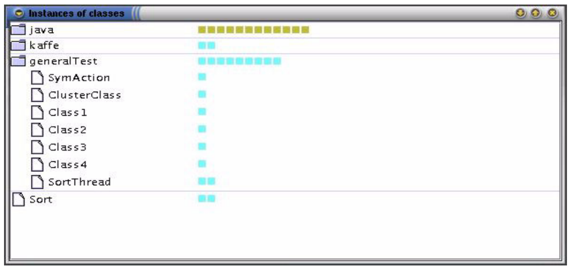

Fig. 5. Instance count display showing the number of allocated instances by package or by class.

long-lived dynamic behaviors. Although we had some successes using VaVoom with the novices (such as using the method depth graph to illustrate the different recursion patterns exhibited by different sorting strategies), we generally found that larger programs were more effectively visualized under our system.

As a result, for our second round of informal evaluations, we looked for more advanced programmers with more complex systems. Examples of systems that we explored include a simple interpreter for a Lisp-like language, a parser for an Architectural Description Language, a log file analysis system, and a collaborative spatial hypertext environment.

There are benefits and costs to this approach. The benefit of dealing with more complex programs is that they have more complex behaviors, and so the advantages of being able to provide developers with rich visual displays are greater. On the other hand, for larger programs, the performance penalty imposed by running the code on a modified interpreter rather than a more conventional JIT or adaptive virtual machine is that much greater. For some programs that we encountered, the performance degradation was simply too great; interactive programs may be especially affected. However, other experiences did suggest that, for some programs, programmers could derive new insights into the behavior of their programs. So, for example, dynamic and emergent behaviors (such as the recursive search through environments in the Lisp interpreter, or xxx... something else... xxx) become more immediately visible, especially when they can be displayed as direct responses to particular patterns of interaction with the visualized program.

Our current set of visual tools reflect some of our early experiences in looking at our own code and working with colleagues on their own. For instance, the method call display is the most recent tool to have been added, since we repeatedly found that people needed to be able to relate elements in the method stack depth display to particular aspects of their program, such as the particular methods being called. The bidirectional link between the method call display and the stack depth tools allows for an interactive exploration of the relationship between the program s static and dynamic structures.

6. Conclusion

As indicated at the outset, our work with VaVoom was intended to offer a proof-of-concept for a longer-term strategy. In the long term, we are interested in the use of dynamic visualization as a mechanism to convey aspects of system behavior to end users, especially in the case of distributed or component-oriented systems. An initial question to be resolved, then, is whether it is possible to derive information from running systems out of which dynamic visualizations can be constructed. Focussing on the Java virtual machine as our first target, we developed VaVoom as a means to explore the extent to which meaningful visual displays could be constructed from low-level execution information.

We regard these initial explorations as successful. While the audience for these program visualizations is quite different from our anticipated long-term user community, we are pleased nonetheless that the visual depictions of programs can be made meaningful. In addition, a number of design features have emerged that we feel will be important to further explorations.

One of these is the use of multiple, linked representations. The fact that we can provide multiple representations which proceed in lock-step allows users to triangulate behaviors, examining them from different perspectives, refining the views and so narrowing down the range of behaviors of interest.

Another is the direct coupling of visualization and execution, so that the programs response to patterns of input action can be directly apprehended. This is especially important for interactive systems, and given that our eventual audience is expected to be the users of interactive systems, this is a significant observation for our future work.

We have deliberately constructed VaVoom so as to support extension and revision. By decoupling the virtual machine from the visual tools, and the visual tools from each other, we have attempted to design an experimental framework in which we can develop and test a range of visual schemes. This allows us to explore a range of future directions. On the practical side, we are interested in porting the environment to different virtual machine implementations which might help us overcome some flaws in the Kaffe virtual machine implementation. In the short-term, we want first to conduct more formal evaluations and then to explore the relationship between visual displays and styles of software systems in more depth. In the longer term, our goal is to explore the opportunities for extending our approach to end users. We intend to do this through two approaches. In the first, we will focus on domain-specific areas of activity, such as the behavior of web search engines, where both engine performance, database coverage, and query reformulation are largely hidden from end users [4]. In the second, we intend to broader our JVM approach to encompass issues such as network behavior that may be more relevant to end user investigation. We feel that the initial experiences with VaVoom support this general approach. Above all, we believe that visualizing the behavior of software systems is an approach which has much broader applicability than has previously been explored.

Acknowledgment

This work was carried out while Johan Byttner was a visiting scientist at UC Irvine. We would like to thank the students who gave time to assist us with informal evaluations, and David Kay, Norm Jacobson, George Leuker, Alex Thornton and Andre van der Hoek for advice and assistance.

References

[1] W. De Pauw, D. Kimelman, and J. Vlissides, Visualizing Object-Oriented Software Execution, in J. Stasko, J. Domingue, M. Brown, and B. Price (eds), *Software Visualization*, Cambridge, MA: MIT Press, 1997.

[2] S. Eick, J. Steffen, and E. Sumner Jr., Seesoft: A Tool For Visualizing Line Oriented Software Statistics, *IEEE Trans. Software Engineering*, pp. 957-68, 1992.

[3] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley, 1999.

[4] J. Muramatsu and W. Pratt, Transparent Queries: Investigating Users Mental Models of Search Engines, Proc. ACM Conf. Information Retreival SIGIR 01, New York: ACM.

[5] J. Stasko, TANGO: A Framework and System for Algorithm Animation, *IEEE Computer*, 23(9), 27-39, Sept. 1990.