Understanding Software

Interaction Design Institute Ivrea

David A. Mellis

October 18, 2005

A discussion of various techniques for understanding the source code and execution of computer programs.

1 Algorithm/Program Visualization

Graphic representations of the execution of a program. Various parts of the state of the program are shown, with time represented as a spatial axis or through animation.

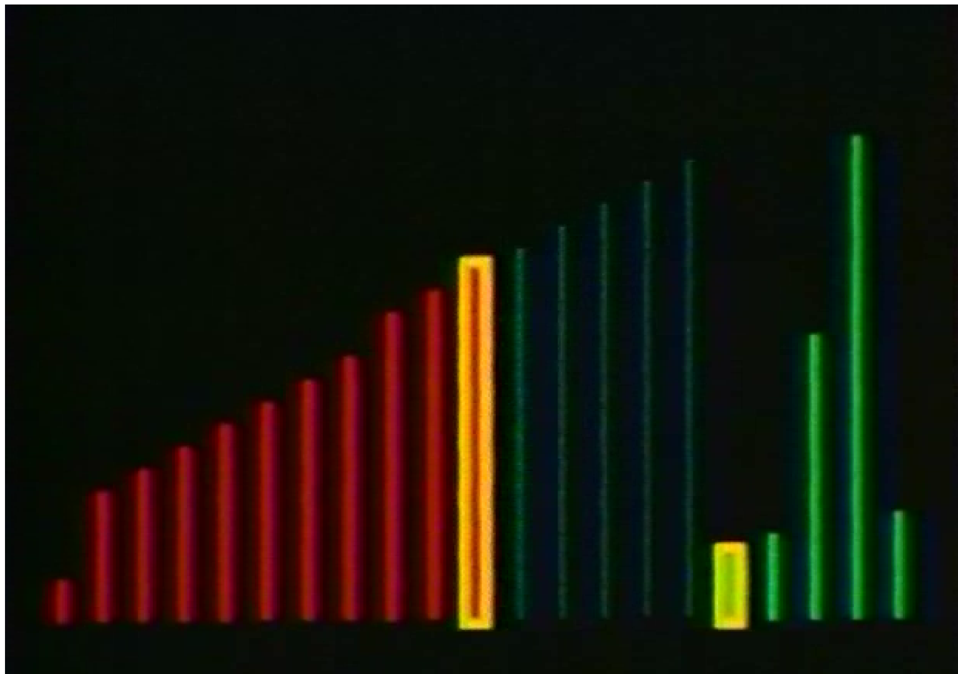

Figure 1: Algorithm Visualization from Sorting out Sorting.

For example, in the visualization of a sorting algorithm, the items to be sorted are often shown as a row of lines of varying lengths. Items being directly compared are highlighted and may be swapped. As the algorithm progresses, its working may be understand by noticing which lines are moved and which parts of the group are sorted first. Eventually, the lines are in order from shortest to longest.

Algorithm visualizations are often custom made for educational use to allow students to examine and compare the workings of various algorithms. The amount of time required to create a useful visualization makes them difficult to use as a general purpose tool.

Algorithm visualizations have become more interactive as the the educational benefits of allowing students to experiment have been realized. *What You See Is What You Code*, by Hundhausen and Brown describes a system in which the visualization and code are kept continuously in sync, allowing for easy manipulation of either.

A related technique is program visualization. This attempts to visualize program execution generically by displaying, for example, a color-coded view of the memory used during execution. Or a diagram may describe the relationship of various functions (e.g. the time spent in each, and which are called from which others). Because of their lack of specificity, these tools are only helpful in limited circumstances.

2 Visual Programming

A method for constructing programs visually instead of textually. Typically, various graphical symbols represent different features of a program, such as variables, control structures, etc.

In *Aesthetics of Computation: Unveiling the Visual Machine* (2001), Jared Schiffman describes several of his excellent visual programming interfaces, including Plate, in which traditional textual constructs are placed on plates, with designated holes with which to fill in values or other statements; and Pablo, a data-flow language in which operations are visually linked together into programs, through which values flow during execution. Schiffman also offers several important principles for visual programming environments. He places primary importance on continuity, including an unified visual space, single visual language, continuity of composition and execution, integration of machine (i.e. code) and materials (values), and continuity of animation. My research seeks to discover what these principles mean for text-based languages.

3 Traditional Debuggers

Traditional debuggers keep track of the correspondence between the source code of a program and the machine code it generates. Thus, they can, for example, halt the execution of a program when a particular line of code (called a breakpoint) is reached. Then the programmer can examine the state of the program's memory, which the debugger can map back to variables in the code.

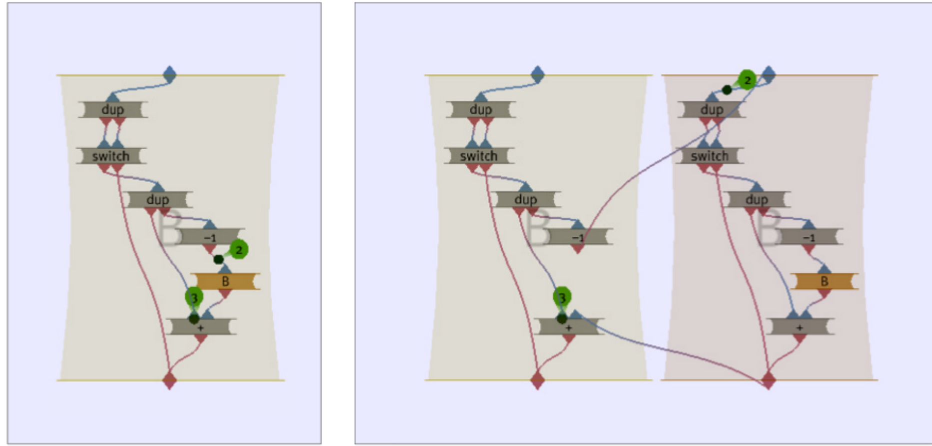

Figure 2: Pablo in the midst of evaluating a function. On the right, a function call has been expanded through the creation of another box.

Lines of code can be executed one at a time (stepped through), or function call can be stepped into. Some debuggers allow breakpoints to be specified for certain conditions (e.g. using an undefined variable) as well particular lines.

Debugging using a traditional debugger can be a very awkward and time consuming process. The most important step is locating the bug. This usually requires guessing many possible circumstances which could create it, stopping the debugger at each one (which might mean repeatedly stepping through a piece of code until the desired condition appears), examining the contents of many different variables (often in a difficult to read form), and slowly advancing through the code to see if the bug appears. Click the wrong button and execution can skip right past the area of interest, requiring a restart of the entire process.

Another problem is the number of distinct pieces of information that must be integrated by the programmer. A debugger shows the values of variables in one window, the program's output in another, the current stack of function calls in a third, program threads in a fourth, with only a small amount of room left over for the source code itself, whose repair is the object of the whole process. Recently, debuggers have begun integrating more information into the source code window, by, for example, displaying the value of a variable when the mouse cursor hovers over it. My research furthers this process, revealing programmers of the cognitive burden of combining many small facets of the program's state.

4 Tracing Debuggers

These debuggers insert into a program code to keep track of various events in its execution, such as a function call or variable assignment. The resulting

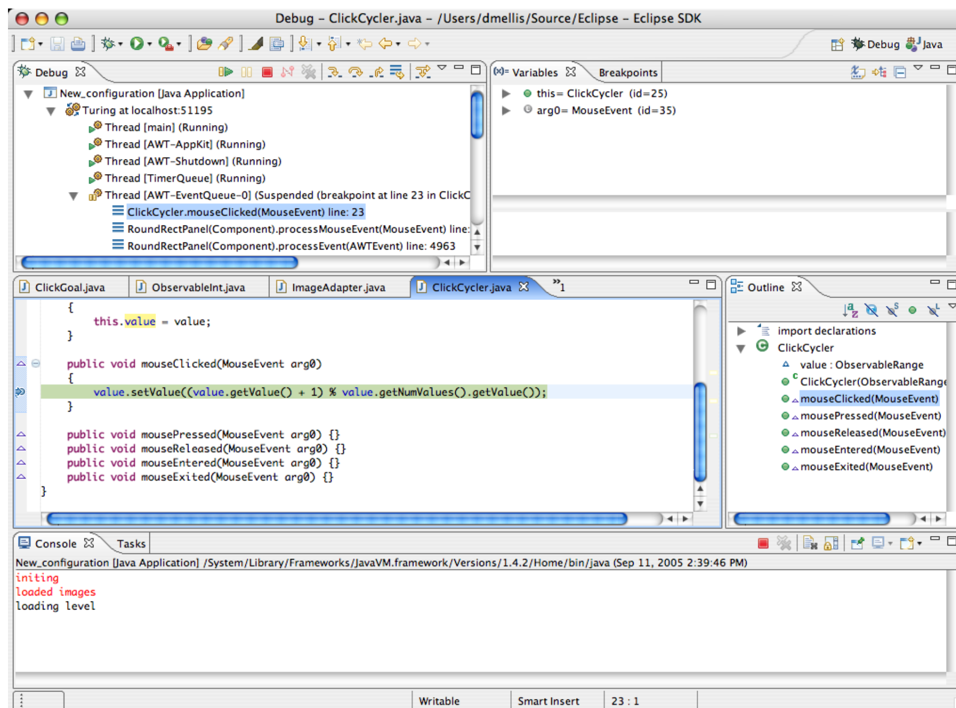

Figure 3: The default debugging perspective in the popular Jave IDE Eclipse. Notice the small portion of the screen devoted to source code.

record is called a "trace." Increasing processor speed, hard drive capacities and higher level languages are beginning to make it practical to record practically every significant occurence in the execution of a program, allowing the programmer to explore backwards and forwards in time. For example, the Omniscient Debugging project has released a tracing debugger for Java, and they also exist for functional languages such as Haskell and OCaml.

The availability of such large amounts of data demands careful attention to the design of the method for exploring it. Goldsmith, O'Callahan, and Aiken, in *Relational Queries Over Program Traces*, describe a method for building a querying a database of function calls using a SQL-like language. They provide examples of how this technique can be used to detect performance problems and answer other programmer questions.

5 Language-Aware Editing

Previously, source code was mainly edited with generic text-editors. That is, the program had no specific knowledge of the structure or syntax of the programming language or the purpose or form of the code. Now many tools can

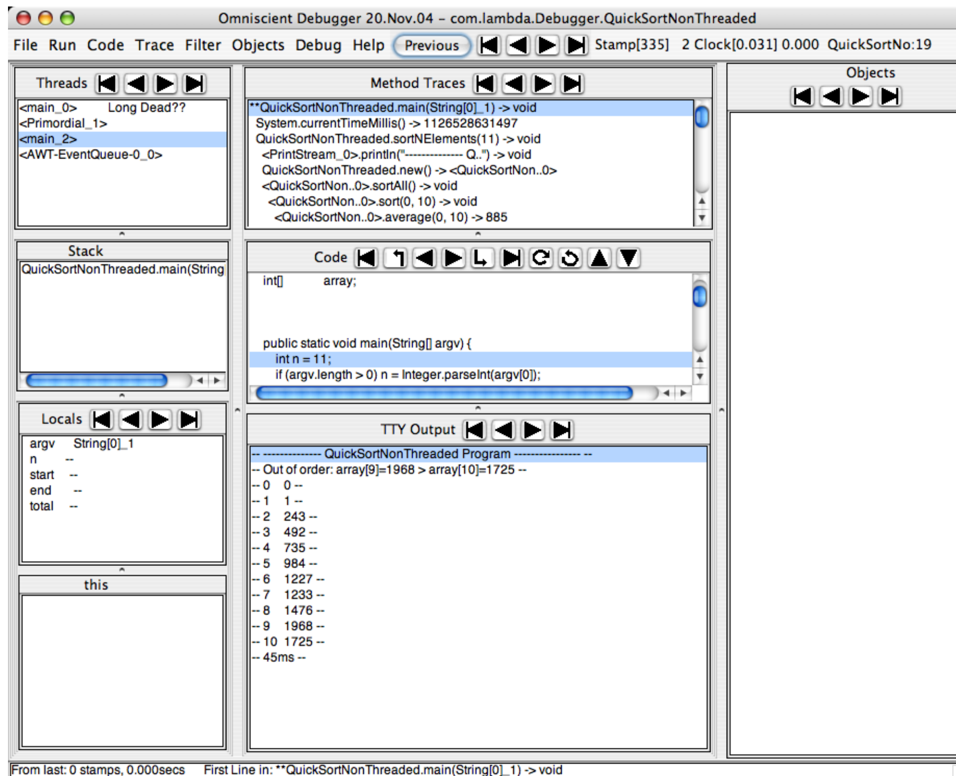

Figure 4: Omniscient debugging. The left and right arrows step forwards and backwards through the execution of the program.

offer services based on an understanding of the program being edited. These include:

- White-space: most compilers ignore most white-space; it is, however, crucial to the legibility of code and many editors will help keep it consistent.
- Comments: ignored by the compiler, but may be auto-generated or organized by the editor.
- Syntax highlighting: the display of different pieces of a program in different colors. For example, comments might be displayed in gray, strings in red, keyword in blue.
- Version history / diffs: an editor might display lines changed by the programmer, or the amount of editing a section of code has undergone.
- Error highlighting: some environments (e.g. Eclipse) will incrementally compile code as it is edited, highlighting syntax errors as they occur (e.g.

with a red underline).

- Command completion (dropdown lists): editors can automatically complete partially-typed names, or display a list of possible options.
- UML and auto-generated class diagrams.
- Links/related sections of code: for example, the place in which the currently selected variable was defined.
- Refactoring: the ability to perform simultaneous, distributed edits to large bodies of code (e.g. renaming a variable or reordering the arguments to a function).

In 1997, Baecker, DiGiano, and Marcus argued that editors could use visual display and organization to provide even more assistance (*Software Visualization for Debugging*):

"A large real program is an information narrative in which the components should be arranged in a logical, easy-to-find, easy-to-read, easy-to-remember sequence. The reader should be able to quickly find a table of contents to the document, determine its parts, identify desired sections, and find their locations. Within the source text, the overall structure and appearance of the page should furnish clues regarding the nature of the contents."

Managing Duplicated Code with Linked Editing (2003) by Toomim, Begel, Graham presents Codelink, a tool for creating, maintaining and editing linked sections of code (i.e. unrefactored sections of code which have much text in common but also include differences). Allows programmers to make consistent changes across related sections of code without the cognitive overhead of restructuring or abstracting them. Drastically lowers the time required to relate sections of code (vs. abstraction). Most code bases have lots of duplication (e.g. 15-25% in the Linux kernel; 9% in GCC; 21-29% in Sun's JDK).

Clones are created by selecting a block of text, then selecting similar blocks of text while holding the Control key. Equivalent sections of the clone are shown with blue backgrounds, differences with yellow backgrounds. A checkbox ("Linked Editing") toggles between linked and individual editing. During linked editing, the cursor becomes a block and ghost cursors (in blue) appear at the corresponding sections of the other clones. During individual editing, the cursor is a bar and ghost cursors disappear. Shared sections of clones can be elided so that only the differences are visible.

The authors would like to add support for moving back-and-forth between linked coded and higher level language abstractions as well as for the automatic creation of linked sections through copy-and-paste or automatic clone detection tools.

Codelink was developed on top of Harmonia, a flexible, extensible system for creating language aware tools.

Some projects have taken an extreme perspective on language-aware editing, creating environments that do not allow any text editing. These include Pablo,

discussed above, and Subtext. Subtext is an environment with no distinction between editing and running a program. The value of a variable is determined by its links to other variables and functions; whenever those links are changed, the values of all variables are immediately updated. Thus, all results of the program are visible as it is being modified. Additionally, all edits are made via links, that is, essentially, to the structure of the code rather than its textual expression. This means that the program is always syntactically correct and can evaluated at all times, and that names are not needed to execute, and are free to be used or not by the programmer as descriptive devices. Also, by keeping track of copies and pastes in the code, Subtext allows duplicates to be managed in similar ways to Codelink.

6 Testing

A newly popular technique for checking program correctness is known as "unit testing." A unit test checks the correctness of a single unit of code in a self-contained manner. This makes them easy to run quickly and an excellent source of examples with which to attempt to understand code. My research seeks to exploit unit tests for programmers trying to understand how code works instead of simply checking it for correctness.

Saff and Enrst have developed a technique (described in *Reducing wasted development time via continuous testing*) for continuously running tests as a program is edited, saving developer from wasting time waiting for tests to run or remaining ignorant of program bugs for long periods of time (making them harder to fix). This suggests that it may be feasible to display state information from program execution while code is being edited. For a given unit test, a programmer could see, perhaps, the values of the variables in the code being edited, metaphorically flowing over the program in a method similar to that of Pablo.

7 Code Evolution

The way a program evolved can be invaluable in understanding how it works. The following papers offer ideas for making use of the revision history of a piece of software.

CVSSearch: Searching through Source Code using CVS Comments (2001)

A tool for searching a project's source code and associated CVS comments. Initially displays a list of matching files with number and types of match (source or CVS comment) for each. Clicking a file summons a vertically-split view: lines matching the search on the left, full source on the right. Clicking a line on the left scrolls the right to the corresponding location and displays the associated CVS comment at the bottom of the window. Lines that more strongly match the search query are displayed with progressively darker backgrounds.

Also presented are techniques for associated comments with lines of code through multiple revisions and building a database from the CVS history, as well interesting statistics on the size, number of revisions per file, number of

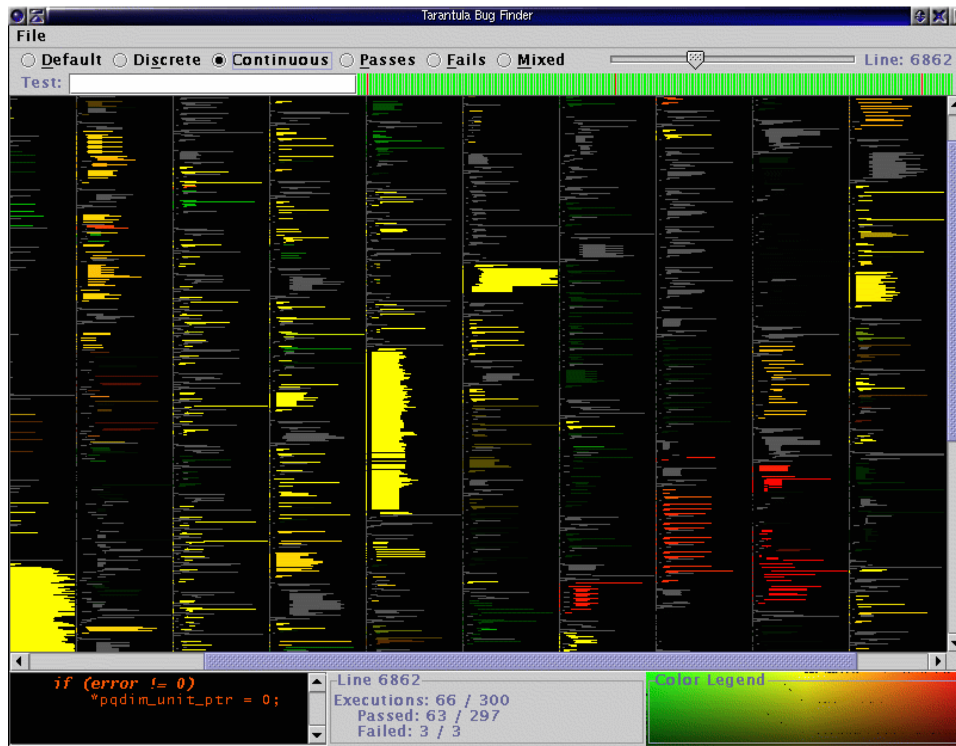

Figure 5: Visualization of test results from Tarantula. Lines of source code that pass tests are shown in green, failing lines in red.

CVS comments per line of multiple KDE projects.

The interface is rudimentary, but the idea good. How can CVS comments be used to provide a more conceptual and historical overview of a line or section of code? What about a tool that simply displays a log of CVS revisions and comments with links to or summaries of the corresponding changes?

Version Sensitive Editing: Change History as a Programming Tool (1998)

David L. Atkins discusses VE, a tool which displays version history during editing of source code. This interface uses simple visual characteristics to highlight the most important aspects of the revision history of a line. In particular, changes made to the working copy of the code (i.e. since the last committed version) are shown in bold, previously-deleted lines, when requested, are underlined, and non-approved code is shown in italics. The date and comment associated with the latest revision of a line are shown in addition to the line number. The programmer can adjust the criteria which determine which lines receive a particular appearance.

The paper provides two realistic examples of the usefulness of this tool to the programmer: finding a bug by correlating dates of revisions of lines with the

```
String FindSource(String base, String dir) {
   DIR * dirp = opendir(dir);
   String result;    // The filename, if found
   for (int i = 0; i < NS; ++i) {      // Loop over suffix list
      String tmp = base + suffix[i];   // Target name to find
      for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
         if (tmp == de->d_name) {     // We found it, stop looking
            result = tmp;
            break;
            return tmp;
         }
      rewinddir(dirp);
   }
   closedir(dirp);
   return result; // Return the found name (may be null)
   return "";     // No match was found
}

Deleted by MR 595 by vz,97/11/15,approved [Stop source search at 1st match]
MR 467 by dla,97/9/21,integrated [Find source using list of suffixes]
"findsource.c", line 15 of 23
```

Figure 6: VE, which shows revision information as code is edited. Lines changed by the programmer are bold, deleted lines are underlined.

occurence of the bug, and coordinating edits with a programmer whose changes have not yet been approved. How might such a system work if not constrained by the limitations of Emacs and Vi?

From *In Search of a Simple Visual Vocabulary (1995)*

"So we can visualize program executions as a series of space-maps. But a series of space-maps is itself just a space-map – a space-map being any arrangment of regions in a space of arbitrary dimensions. We can manipulate this history using the same construction, deconstructions and evaluation rules that we use for any other data object. A program history can be used as data for visualizing program execution, debugging and communication."

Questions

How to display an overview of large amounts of code? What do you see half-way between code and diagram?

How to distinguish static code from dynamic values?

How to show a change in the state of the program (e.g. assignment of a value to a variable)? How to watch the progression of a variable or other aspect of program state?

How to show nested function calls? How to show repeated execution of a block?

How to specify an “interesting” event (e.g. call to a particular function with certain arguments, particular exception, execution of a specific statement)? How to specify the useful context of an event?

How to show (revision) history of code? The programmers explanation is invaluable to understanding a piece of code. How do I incorporate it?

Tools for programmers are made by their users; why then are they badly designed?

Why do programmers think that graphical/visual programming tools dont scale to large programs? What work has been done in this area?

Are there any small, precise visualizations that could be useful, perhaps something like sparklines or thread arcs?