Understanding Code

David A. Mellis
Interaction Design Insitute Ivrea

Draft: March 20, 2006

Abstract

Understanding Code

David A. Mellis

Current methods for debugging and reading source code impose too much cognitive burden on programmers. *Understanding Code* presents the design of a tool for exploring a program's behavior. It provides a unified interface that exposes an overview of the code's execution, connections between and relative importance of different pieces of the program, and the exact, step-by-step computations performed by the software. By making it easier to understand code, an implementation of this interface would increase the maintainability and extensibility of existing programs.

Contents

1 Introduction **2**

2 Background Research **4**

3 Explanatory Prototype **13**

4 Analysis **16**

5 Interface Design **20**

6 Feasibility **28**

7 Evaluation **29**

8 Conclusion **30**

A Elements of Programming **32**

Chapter 1

Introduction

Writing a computer program is like trying to assemble a grandfather clock, blindfolded, without being sure that one has all the parts. That is, it requires the coordination of countless intricate pieces with no good way of observing the functioning of the whole. Put one tiny part in the wrong place and everything stops, but you can only discover the error by probing each part of the system in turn, memorizing the numerous linkages, combinations, and movements. It's a wonder any programs ever work.

Software needn't be so. No physical constraints govern the arrangement of its components. Nothing need be hidden from view behind a decorative covering. Only our ingenuity limits the number of ways we can recombine different pieces, the tools we can use on them, the ways we look at their operation. As Fred Brooks has said, "the programmer, like the poet, works only slightly removed from pure though-stuff. He builds his castles in the air, from the air, creating by exertion of the imagination."

It falls to us, then, the makers and users of programming languages, libraries, environments, to decide which tools we need, how they should function, and what they should look like. The ones we have now are just starting to adjust their forms to the problems we are trying to solve and the behaviors we are trying to understand. Originally, they were primitive, general-purpose instruments: the text editor knew nothing of the programming language; the operating system cared little for one's source code. Now connections are beginning to form: editors display variables in a different color from strings, comments with less saturation than function calls; a program crash often comes with a list of the lines of code immediately preceding the disaster; syntax errors in a source file get a squiggly red underline as you type; one can edit the code of a running program; even record every function called, variable modified, input received.

And yet, there is no equivalent to opening the case of the clock and watching it tick out the seconds, swaying pendulum letting rotate a gear, that regulating the revolutions of another, slower, one, and that a third, and a fourth, chains driving the hands from behind, the whole ballet powered by a slowly descending weight. We cannot watch a whole program at work.

Of course, there are difficulties. Software is orders of magnitude more complex than even the most intricate clock. It is made of text - words that become meaningless at a distance. It runs inconceivably fast, so that we cannot possibly examine each of its actions individually. It is made of heterogeneous parts, written by different people in different languages with varying degrees of secrecy. It is written under pressure, changed often, and required to work under wide-ranging conditions, with a menagerie of accessories and managers.

Still, reasons exist for hope. As computers get faster and the complexity of their software increases, so too do the resources it offers us to understand and assemble our code. We constantly find new abstractions that allow programmers to work at higher and higher levels, and to reuse more and more mature technology. As we learn that programmers are people too, we can successfully apply to them many of the principles that help ordinary people use all types of software.

This thesis attempts to do just that: show how an understanding of the goals and mindsets of programmers can be used to design tools to help them understand the dynamic behavior of their programs. It attempts to reduce the cognitive burden on programmers by suggesting how knowledge and reasoning could be shifted from their heads into a tool. By making it easier to read as well as write code, it hopes to ease the reuse and debugging of existing source.

It firsts reviews background research, discussing the limitations of current tools for understanding programs and drawing inspiration from related areas of work. Chapter 3 presents a prototype created early on to delimit and explain the focus of this thesis. Chapter 4 analyzes the process of understanding and debugging software, both as an aid for non-technical readers and as an essential step in determining the requirements for the design of the software tool presented in Chapter 5. This interface is the primary product of thesis, a proposal for understanding the complexity of the behavior of an executing piece of software. Chapter 6 discusses the technical feasibility of such a tool and Chapter 7 evaluates its usefulness. Finally, Chapter 8 discusses the lessons learned from this thesis process.

Chapter 2

Background Research

Need to add spreadsheets, Code Profiles by W. Bradford Paley, ZStep95

The screenshots in this section need to be annotated to identify individual components of the various interfaces in order to facilitate comparison with the interface developed for this thesis.

Traditional Debuggers

Need to describe print-statement debugging here.

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. 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.

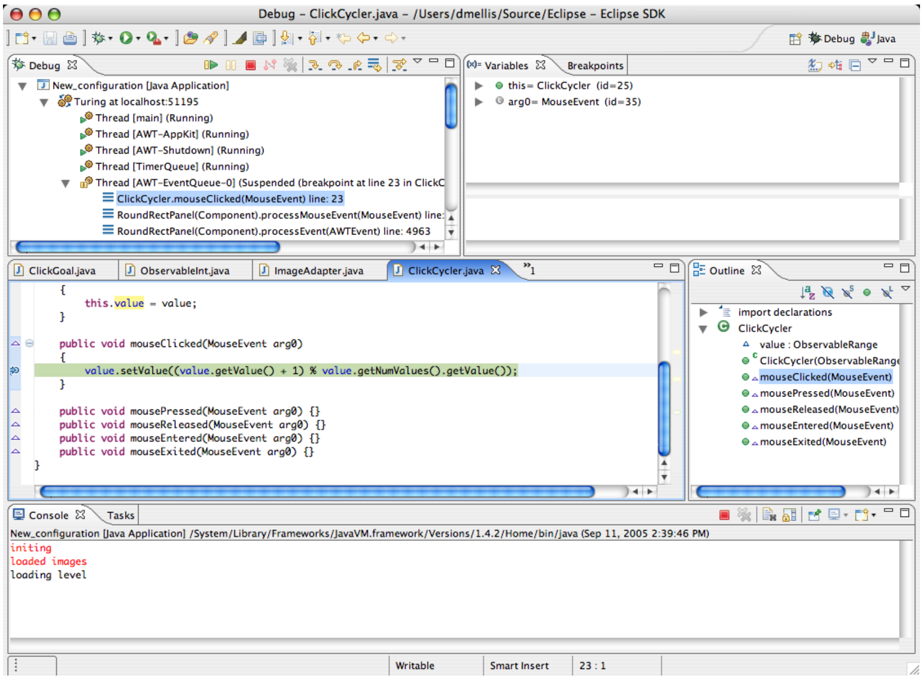


Figure 2.1: The default debugging perspective in the popular Jave IDE Eclipse. Notice the small portion of the screen devoted to source code.

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 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.

Language-Aware Editing

Include Xcode screenshot here.

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

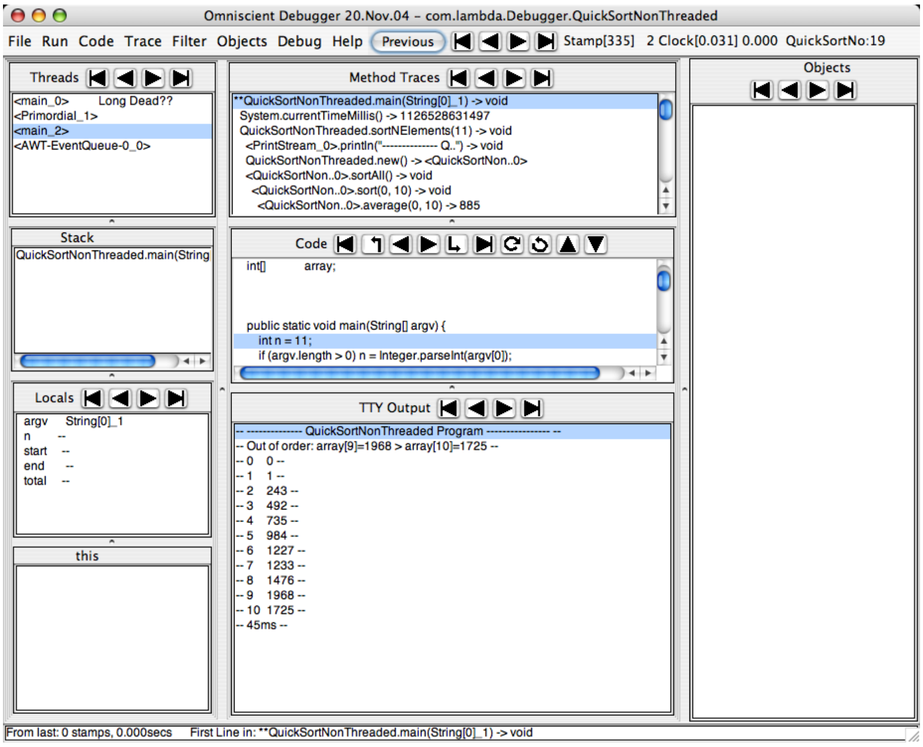


Figure 2.2: Omniscient debugging. The left and right arrows step forwards and backwards through the execution of the program.

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 below, 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.

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.

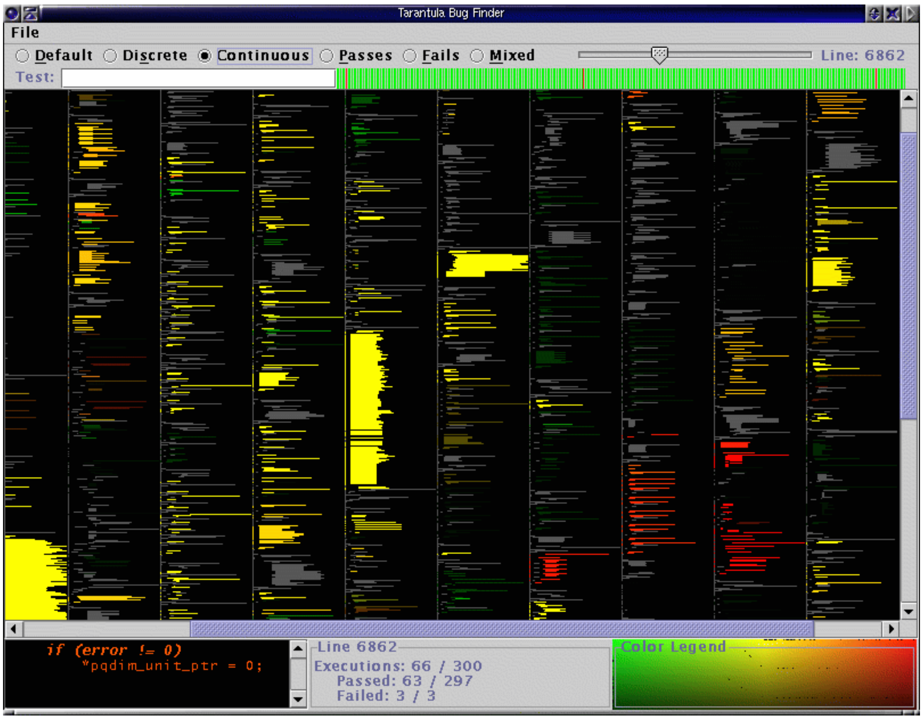


Figure 2.3: Visualization of test results from Tarantula. Lines of source code that pass tests are shown in green, failing lines in red.

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 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 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."

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.

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

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.

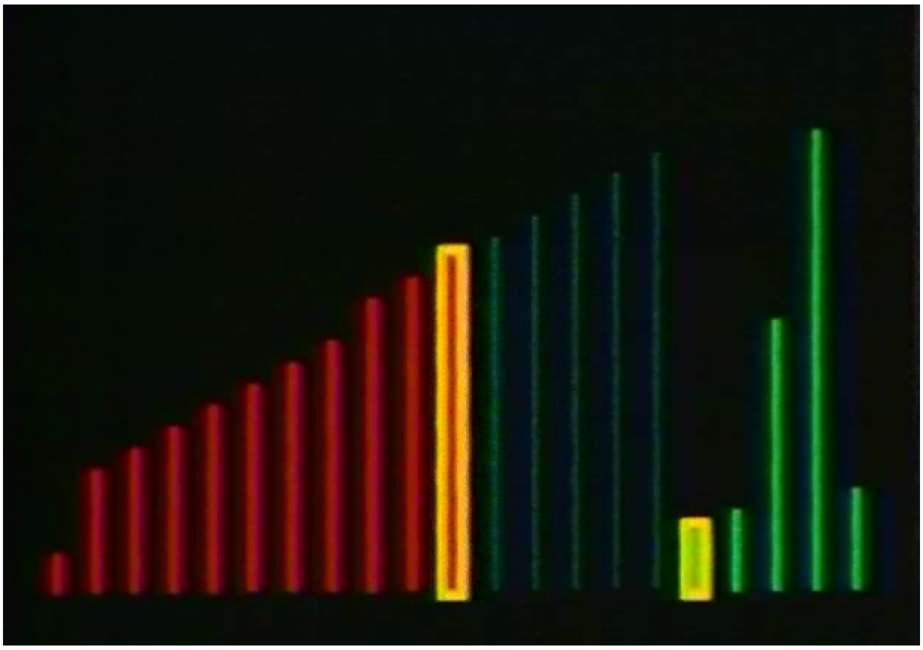


Figure 2.5: Algorithm Visualization from Sorting out Sorting.

Visual Programming

Need to talk about spreadsheets somewhere.

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.

Custom/Complex Interface Elements

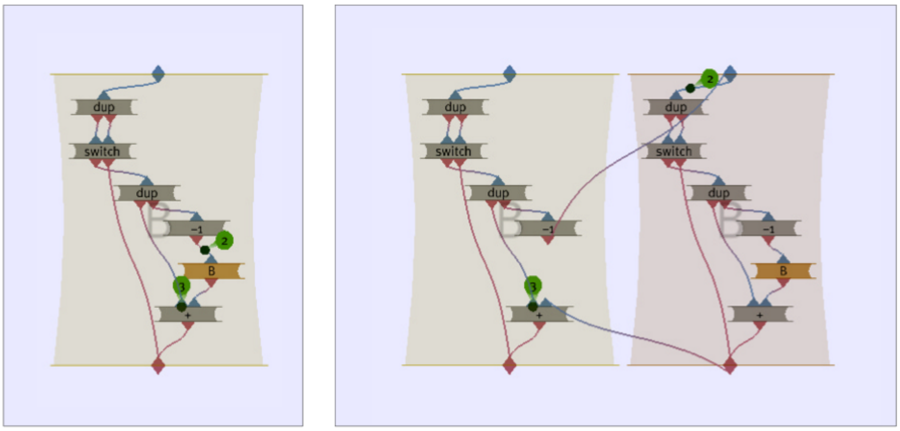


Figure 2.6: Pablo in the midst of evaluating a function. On the right, a function call has been expanded through the creation of another box.

Chapter 3

Explanatory Prototype

In an attempt to ground my thesis and explain its domain to a non-programming audience, I began by creating an explanatory prototype. It provides an example of a non-graphical visualization of program behavior: using code (the medium of programming) as an interface to explore computation.

Introduction to Scheme

The prototype took the form of a debugging interface to the Scheme programming language (a dialect of Lisp). I choose Scheme because it uses the same simple (if a bit strange-looking) syntax for every operation: nested, parenthesized expressions, each containing an operation followed by its arguments – e.g. `(+ 1 2 3)` adds the numbers one, two, and three to get six. This uniformity meant that I could use a single technique to provide interactions with any code.

Part of the power of Scheme comes from its interactive environment (known as a REPL: "read-eval-print loop"), which allows a programmer to type in a Scheme expression and immediately view its resulting value, without the need for any intermediate steps such as compiling or running the program. This rapid evalution allows a programmer to quickly try out a library of code or test a newly written function.

```
> (define (factorial x)
>   (if (< x 1)
>       1
>       (* x (factorial (- x 1)))))
> (factorial 3)
6
```

Figure 3.1: Transcript of an interactive session with a Scheme interpreter. Lines prefixed with a "`>`" were typed by the programmer; others are responses by the interpreter. In this case, the programmer defined a function to compute the factorial of a number. Then the programmer used it to compute the factorial of 3 (which is 6).

If, however, the programmer doesn't understand how the expression typed yielded the value returned, Scheme environments provide no easy method for digging deeper into the code's execution. My prototype reveals the entire chain of computation to the programmer for exploration.

How it Works

A basic set of interactions allows for exploration of the computation performed by a program. Clicking a value expands the code that calculated it. The clickable values are underlined, similar to hyperlinks on a web page. Values of variables are shown in blue, and hovering over them with the mouse shows the variable's name. Code that wasn't executed is grayed out.

For example, the initial result of 6 expands into `(factorial 3)` (the function call that calculated it), followed by `(if (= 3 1) 1 (* 3 2))` (the body of that function). Here, the 2 is also the result of a calculation, and clicking it will reveal that code.

```
> (factorial 3)    > (factorial 3)    > (factorial 3)
6                  6                  6
                   |                  |
                   (factorial 3)      (factorial 3)
                   (if (= 3 1)        (if (= 3 1)
                       1                  1
                       (* 3 2))           (* 3 2))
                                                |
                                                (factorial 2)
                                                (if (= 2 1)
                                                    1
                                                    (* 2 1))
                                                          .
                                                          .
                                                          .
```

Figure 3.2: Expanding the function calls. **Show mouse cursors?**

Implementation

Initially, the prototype consisted of a simple, hand-coded webpage. Hyperlinks triggered short Javascript functions that expanded or collapsed the relevant sections of code. HTML offers a straightforward, precise control of typography, layout, and behavior that's difficult to achieve with either graphic design software like Adobe Illustrator or by programming a desktop application.

Additionally, the use of HTML eased the transition into the second, working prototype. This version allows a programmer to enter arbitrary Scheme code (with some limitations) and navigate the resulting computation in the same way as in the first, canned prototype. The interface consists of similar webpages, but in this case, they are dynamically generated by a Scheme program. Here, a second property of Scheme was also essential – namely, the existence of simple, freely available Scheme programs to parse and execute Scheme code. I modified one from Abelson and Sussman's *Structure and Interpretation of Computer Programs* to record the program's computations and output them as HTML.

Discussion

This process is distinct from those of writing and running code. Like reading a map, it involves following trails, finding connections, and seeing how different parts fit together. As with an online map, it allows one to zoom in on different pieces. In short, it is a mirror or visualization of the actual computation performed by a program that is:

- *specific* to a particular execution on particular values,
- a *passive* recording of past activity, and
- *structured.*

The code itself, in contrast, is abstract, requiring more effort by the programmer to deduce its behavior for a given input. Traditional debuggers suspend a program's execution, meaning that in order to view a different part of a computation, the programmer must run more of the program. This makes it impossible to look back at a previous program state or jump between two sections of interest. Debugging logs output by the program give a specific, passive view of the execution, but an unstructured one: the logs cannot be navigated or cross-referenced.

Additionally, the continuity between the language used to edit a program and the one used by the prototype to display its behavior avoids the additional cognitive load imposed by tools which use different interfaces for each task.

Questions

While this prototype delimiited and communicated the subject of the thesis, it also identified some questions for the remainder of the thesis investigation.

- How does this approach scale to larger programs?
- How does this functionality integrate into a complete user interface?

Chapter 4

Analysis

Frustrations with Current Tools

Background chapter should explain limitations of current tools; this section describes the programmer frustrations that arise from those limitations.

Current tools for understanding and debugging code offer many frustrations to programmers. Most fall under the overall complaint, “I can’t keep it all in my head.” Programmers need to remember previous states of time and connections between different parts of the program.

a result of user research

- I can’t tell what connects to what.
- I’m getting lost in the details.
- I can’t tell what’s happening.
- I don’t know where in the code to look.
- I don’t know what will happen if I change this.

Breakdown of the Debugging Process

In order to better understand the process and requirements of debugging (one of the major reasons for seeking to understand a program), I created this breakdown based on my own experience and interviews with two professional programmers.

Task 1. Determine relevant general section of code.

technique a: guess source of problem. place breakpoints before and after.

- action: look through different source files
- action: place breakpoints
- action: start debugger

technique b: flag lines of code triggered by a particular action/command/input.

- action: run program
- action: start recording actions/inputs
- action: interact with program to reach area of interest
- action: begin flagging
- action: perform action which triggers incorrect behavior
- action: end flagging
- action: stop recording
- action: exit program
- action: start debugger

technique c: logging (print statements)
technique d: compare with a working version of the code

Task 2. Narrow in on specific, proximate cause (i.e. specific line of code).

- action: step through code one line at a time
- action: monitor watch window (variables window) while stepping, checking for incorrect values
- action: flag proximate cause

Task 3. Determine if proximate cause is root cause (i.e. is that line of code right or wrong?).

- action: reasoning
- action: reading documentation for objects/functions used in the line of code: do they do what the author of the code thought they do?
- action: perform computations.
- action: fudge values

Task 4. If this is not the root cause, repeat tasks 2 and 3.

- action: step into function which returned incorrect value/performed incorrect computation
- action: review previous changes to variable which has incorrect value
- action: examine external dependencies (e.g. values in database, behavior of other programs, contents of a file, etc.)

Task 5. Fix root cause (i.e. edit the code).

- action: text editing
- action: look up classes/functions in documentation

Task 6. Check that it corrected the incorrect behavior.

- action: replay input (GUI, files, network, DB, etc.)

Task 7. Make sure nothing else broke.

- action: run unit tests
- action: review unit test results for failures

Task 8: If something else broke, figure out why (tricky).

Other Motivations for Understanding Code

Debugging isn't the only reason for seeking to understand a program. Here are some of the others.

Need to understand a program/library generally

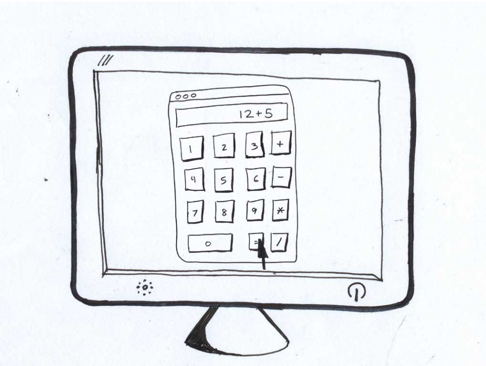
Reason 1: want to use the program/library
Reason 2: want to modify/improve program/library
Reason 3: want to learn from program/library

Need to understand dependencies

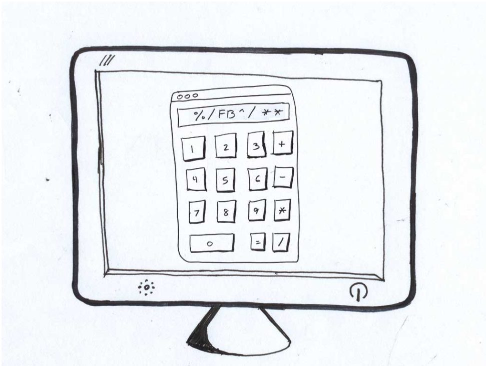
SITUATION IV: need to monitor a long-running program (e.g. memory usage, performance, errors, etc.)

Debugging Scenario

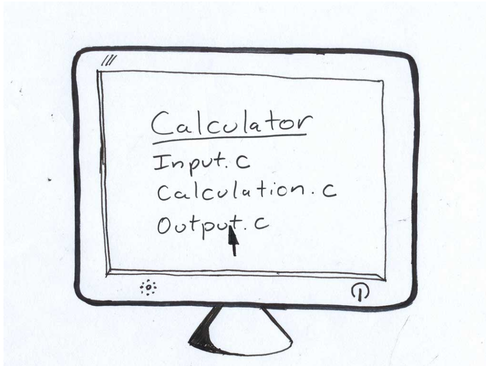
See Figure 4.1.



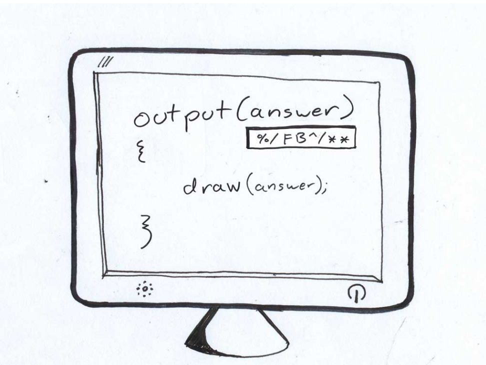
(a) Adding 12 and 5 with the calculator



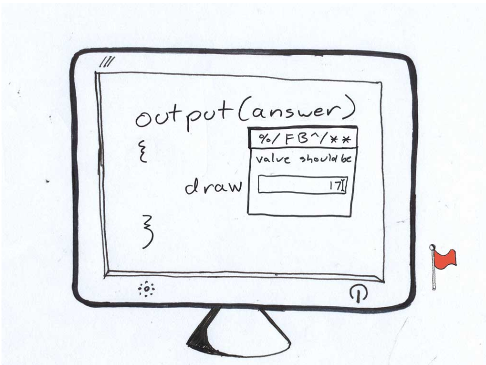
(b) The program gives the wrong answer; it must have a bug.



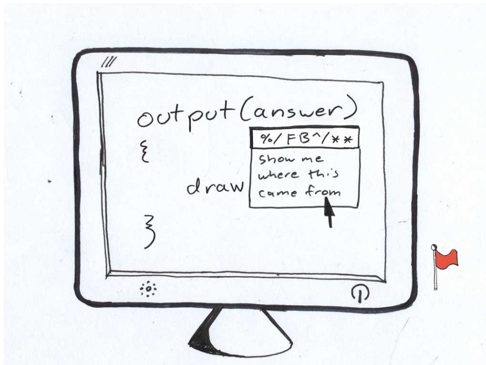
(c) A list of the files of source code used by the program; something that most current debuggers don't show.



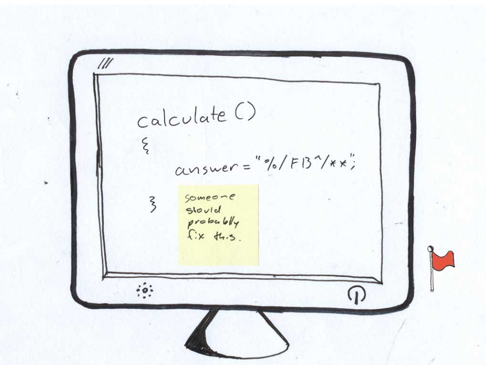
(d) The contents of `output.c`. The box shows the value of the variable answer passed into this function; it is wrong.



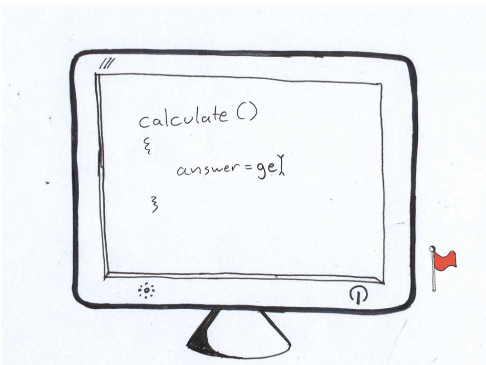
(e) The programmer enters the correct value for this variable; a red flag appears to indicate that there is currently an incorrect value somewhere in the code.



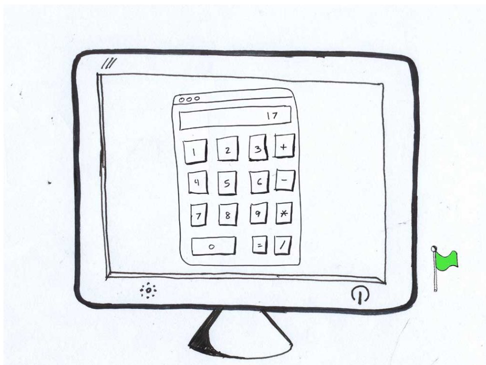
(f) To help track down the root of the problem, the tool can jump to the source of the value of this variable.



(g) Here's the problem: someone neglected to write the code to calculate the answer.



(h) The programmer types in the correct code.



(i) And now the calculator gives the right answer. Notice that the flag has turned green, to indicate that the value of the answer variable is correct.

Figure 4.1: Debugging scenario showing mindset and needs of programmer.

Chapter 5

Interface Design

Understanding Code presents a unified history of a program's behavior. It is designed for mainstream programming languages like C, C++, or Java, which share a similar syntax. It differs from previous tools in its integration of variables and values; its unified display of the progression of time; and its visualization of the connection between different parts of the program.

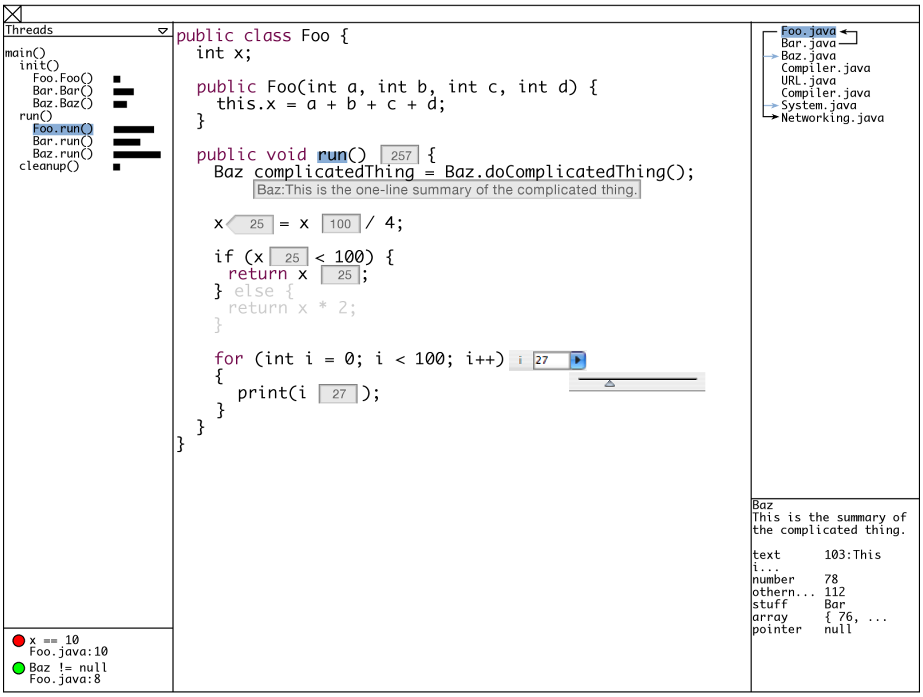
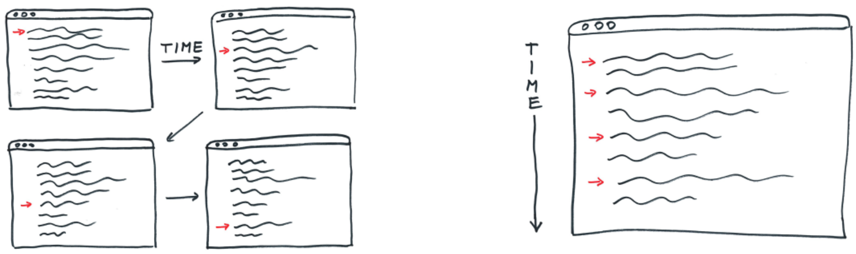


Figure 5.1: Preliminary wireframe of the *Understanding Code* software tool.

Time and Sequence

Traditional debuggers freeze a program at a particular moment. To show the state of a program at a later point in time, the debugger runs more of the program. There is no way to look back at previous states of the program, so the programmer remember any relevant information and be careful not to miss any important calculations by letting too much of the program run at once. See Figure 5.2(a).

The *Understanding Code* environment, in contrast, presents the *entire history* of the program in an unified interface (Figure 5.2(b)). It does this by keeping track of every operation performed by the software when it runs, including input and output. This means that a programmer can compare multiple states of the program, and watch the evolution of particular variables. This passive interface puts the programmer in control.



(a) In a traditional debugger, looking ahead in a program's execution completely replaces the previous state.

(b) In the *Understanding Code* environment, multiple states of the program can be seen at once.

Figure 5.2: Two approaches to handling time. **Should this go in analysis?**

Sequences of time are ordered and grouped in a few different ways. Within a particular function, execution typically proceeds down through the code, one line at a time.

Conditionals (like `if`-statements) cause certain lines of code to be executed only under given conditions. In *Understanding Code*, unexecuted lines are grayed out. Removing them completely would disorient programmers familiar with the code and expecting to see those lines. **(This should be user tested.)**

Loops can cause some lines of code to be repeated multiple times, each time with potentially different values and behaviors (Figure 5.4).

Need to design and discuss a way of showing multiple iterations of the loop in succession.

The most important method of organizing programs is that of splitting the code into functions: named, relatively self-contained pieces that take inputs, perform a particular task, modify the program's state and return outputs. The record of which functions were called, in what order, with what inputs, and by

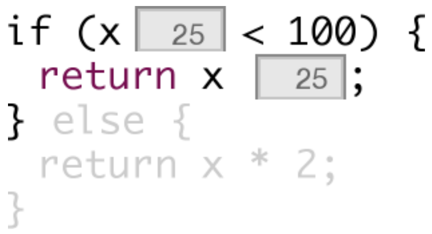


Figure 5.3: An `if`-statement. The condition `x < 100` is true, so the first block is executed and the second skipped.

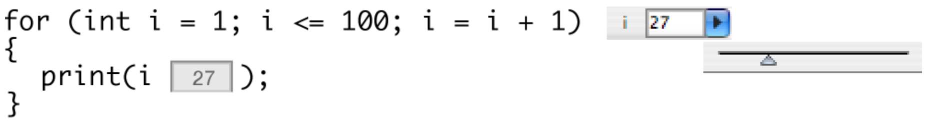


Figure 5.4: A loop whose body runs 100 times, each time giving `i` a new value. Editing the text box or dragging the slider selects which of these 100 iterations to show.

which other functions forms the *function log*, the core of the program history displayed by *Understanding Code*. It provides a timeline not of minutes and seconds, but of structural units. As each function may call many other functions, this record forms a hierarchical tree (Figure 5.5).

A typical file of source code provides little indication of the relative importance of its lines. One file may contain dozens of function calls, most of which perform an unimportant task and a few of which contain most of the work of the entire program. *Understanding Code* attempts to visualize this relative importance of various function calls. It does this by displaying a bar next to each line in the function call record whose length is proportional to the number of function calls nested beneath that line (Figure 5.6).

A function may be called many times in the course of a program – each time the same source code operates on (possibly) different inputs and may behave in different ways. (For example, in Figure 5.5(c), `Foo.thing1()` was called twice.) Most software only displays the code: the abstract instructions isolated from the data it processes. *Understanding Code*, however, shows the code together with its values; that is, the programmer can watch the program in action. To view a portion of the execution, the programmer selects a line from the function log. Because this line refers to an individual call to the function listed, the data operated on at that particular moment can be shown. I refer to this function call and associated data as the *live function*. The next section describes its display.

Navigating the call stack.

```
main()
 init()
 run()
 cleanup()
```

(a)

```
main()
 init()
 run()
   Foo.run()
   Bar.run()
   Baz.run()
 cleanup()
```

(b)

```
main()
 init()
 run()
   Foo.run()
     Foo.thing1()
     Foo.thing2()
     Foo.thing1()
   Bar.run()
     Bar.step1()
     Bar.step2()
   Baz.run()
     Baz.doItAll()
 cleanup()
```

(c)

Figure 5.5: Function log composed of the hierarchical sequence of function call performed by the program. The functional call named in a line occurred later in time than the line preceeding it. Each function, however, may contain calls to other functions.

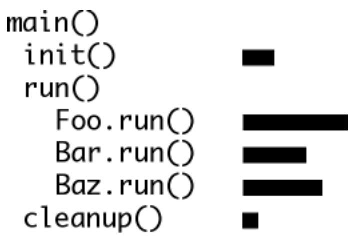


Figure 5.6: Bars next to each line show the amount of complexity contained inside the corresponding function; that is, the length is proportional to the number of function calls that would appear contained beneath that line if it were fully expanded.

Variables and Values

Within the live function, *Understanding Code* displays a variable's value alongside it, using custom interface widgets to embed these values into textual source code (Figure 5.8). This integration allows the programmer to see the different values a variable takes on at different points in the code. The display of the values (which would benefit from further graphical refinement) attempts to be legible but visually separate, so that the programmer can read just the code if desired.

This needs user testing. What about showing values as sidenotes next to the relevant code?

```
int sum(int a, int b, int c, int d) {
  int x = a + b + c + d;
  return x;
}
```

Figure 5.7: The source code of a function: abstract instructions that can be applied to many different inputs.

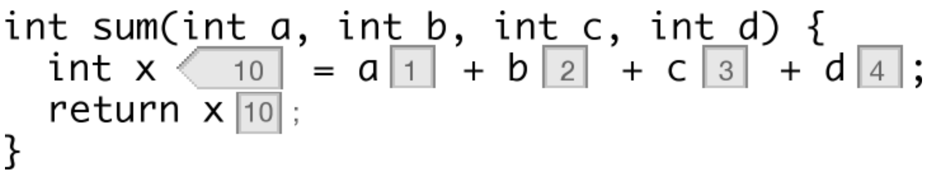


Figure 5.8: Variable values integrated into the code. Here, we see the numbers 1, 2, 3, and 4 that were passed into the `sum` function, and the number 10 which is returned as its result.

Some variables contain complex data that cannot be easily displayed within the code itself. In this case, a summary appears instead, which can be clicked to pop up more details. These details can also be docked to a corner or side of the *Understanding Code* interface. This allows the programmer to monitor changes to its value across different parts of the program.

Searching backwards for changes to a variable.

A crucial debugging is determining how a particular state came about, or how a variable acquired a particular value. The current *Understanding Code* interface offers the ability to search backwards to locate previous changes to a variable. Additionally, the programmer can filter these results by value – to see, for example, only places where a numerical value was negative. See Figure 5.10.

This would be a good place for an animation.

Viewing the history of a variable.

Flagging incorrect values (Figure 5.11).

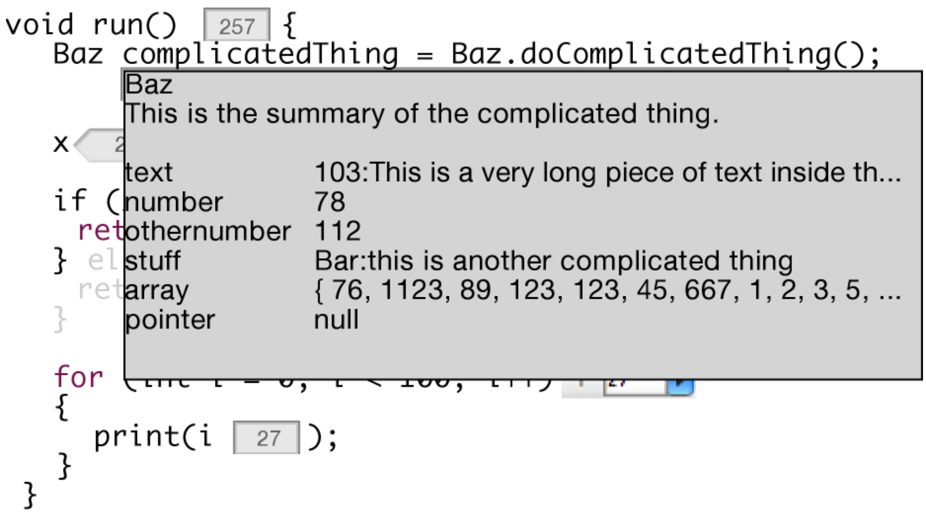


Figure 5.9: Popup showing details of a complex variable.

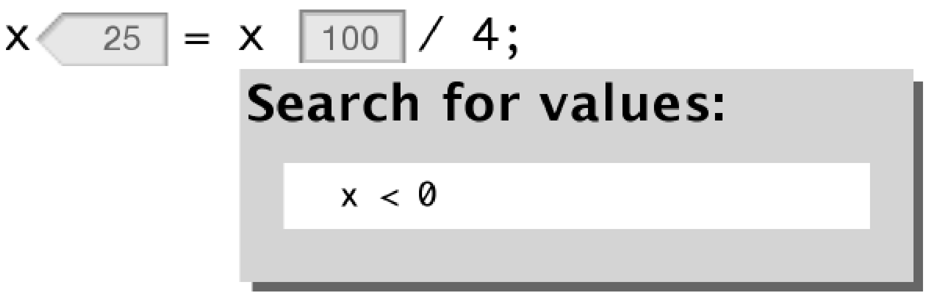


Figure 5.10: Searching backwards through the program's execution for times when a variable had a particular range of values.

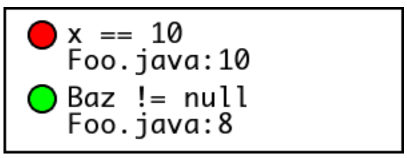


Figure 5.11: Flags provide an easy way to check whether certain parts of the code are functioning properly.

Connections and Dependencies

Another aspect of a program that's not apparent from the code is the way in which the various pieces depend on each other. A programmer hoping to reuse a particular component in another program, for example, would benefit from an easy way of determining which other components also needed to be brought over. Or a programmer about to change a piece of code might want to review all the places where that code is used. *Understanding Code* provides an easy way to see the dependencies between files and functions in a program. In the list of all the files in the project, the one which is currently open is highlighted. Arrows point from it to the files it uses, and to it from files that use it. If one of the functions in the file is currently live (i.e. a particular call to that function is being shown together with the data it operates on), the file is highlighted in blue, and the files used by that particular function are pointed to with blue arrows.

Consider adding ways to jump to or list all the specific dependencies (functions or lines) of a function, file, or line.

Static structure (files).

(a) Dependencies between files in a project.

(b) Here, files used by the current live function are shown in blue.

Figure 5.12: Showing dependencies between parts of a program.

Comparing Versions

For now, see Figure 5.13.

Need to determine the code view for these comparisons as well as how the versions are managed.

Threads.

Conclusion

Need to consistently scale the figures in this chapter.

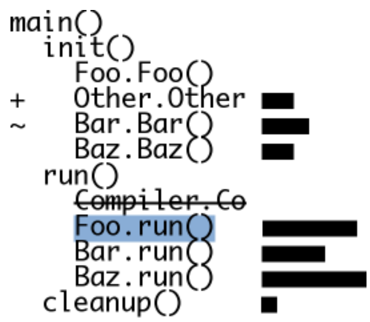


Figure 5.13: Comparing two runs of the program (different versions of the source code or on different input).

Chapter 6

Feasibility

The Omniscient Debugger mentioned in the background research provides evidence of the feasibility of collecting a comprehensive record of a program's execution and displaying it to the programmer. Though currently only a research prototype, it could be extended to handle larger and more complex programs. In particular, in order to reduce the amount of storage needed, portions of the execution history could be dynamically regenerated when viewed by the programmer. Or the programmer could select only certain portions of the program to be logged.

The interface presented in the preceeding chapter could be implemented with the same information collected by the Omniscient Debugger. More analysis would be required – in order to show, for example, the dependencies between the different files of a project. Some parts of the interface require custom interface components (e.g. the embedding of variables values within source code).

None of these tasks seem substantially harder than those involved in the development of the other parts of a programming environment such as Microsoft's Visual Studio or the open-source Eclipse.

Chapter 7

Evaluation

Chapter 8

Conclusion

Bibliography

[1] Abelson, Harold, and Gerald Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs*, MIT Press: Cambridge, MA, 1984.

[2] Atkins, David L., 'Version Sensitive Editing: Change History as a Programming Tool'. *ECOOP 98, SCM-8, LNCS 1439.* Berlin: Springer-Verlag. 1998. 146157,

[3] Baecker, Ron, Chris DiGiano, and Aaron Marcus, 'Software Visualization for Debugging'. *Communications of the ACM* Vol. 40, No. 4, April 1997. 44-54.

[4] Chitil, Olaf, Colin Runciman, and Malcolm Wallace, 'Freja, Hat and Hood A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs'. *IFL 2000, LNCS 2011.* Eds. M. Mohnen and P. Koopman, 2001. Berlin: Springer-Verlag. 176193.

[5] Edwards, Jonathon, 'Example Centric Programming'. *OOPSLA04*, 24-28 Oct. 2004.

[6] ——, 'Subtext: Uncovering the Simplicity of Programming'. *OOPSLA05*, 16-20 Oct., 2005

[7] Goldsmith, Simon, Rober t OCallahan, Alex Aiken. 'Relational Queries Over Program Traces'. *OOPSLA05*, 16-20 Oct., 2005.

[8] Saff, David, and Michael D. Ernst. 'Reducing wasted development time via continuous testing'.

[9] Schiffman, Jared. *Aesthetics of Computation – Unveiling the Visual Machine*, Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology, 2001.

[10] Tung, Sho-Huan Simon, 'Visualizing Evaluation in Scheme'. *LISP and Symbolic Computation*, no. 5. 1997. 1-23.

Appendix A

Elements of Programming

A short introduction to some programming concepts and vocabularly.

Is this needed at all? Should it be integrated into the main text?

Variables and Values

In a computer program, a *variable* is place in memory to store data, along with a name by which to refer to that place. The data stored in a variable is called its *value*. Variables may containd different kinds of values: for example, a number or a sequence of letters. Or a variable may hold a collection of different pieces of data; e.g. a name and address used to represent a person.

Loops

A *loop* repeats a certain piece of code (called the *body* of the loop) while or until a particular *condition* is satisfied. For example, a program might do something with each letter in a text file.

```
i = 1;
while (i <= 10) {
  print(i);
  i = i + 1;
}
```

Figure A.1: A `while` loop, which repeats while the variable `i` is less than or equal to 10. This will print the number 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Conditionals

Function calls

Call stack

Time and Sequence

Files

Dependencies

Bug