

Thesis Report

Tangible Code

David A. Mellis
Interaction Design Insitute Ivrea

<http://dam.mellis.org/thesis/>
dam@mellis.org

May 22, 2006

Neil Churcher
Thesis Advisor
Director and Chair of Examiners

Yaniv Steiner
Thesis Advisor

Phil Tabor
Report Advisor

Heather Martin
Director and Chair of Examiners

Abstract

Tangible Code

David A. Mellis

Current methods for debugging and reading source code impose too much cognitive burden on programmers. This report presents the design of a tool for exploring a program's behavior, making tangible actions and constructions that previously existed only in the mind of the programmer. 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.

a Ivrea la Bella

Acknowledgements

To Neil Churcher, for helping me plan, analyze and communicate my work in his own precise, inscrutable way.

To Yaniv Steiner, for challenging me to defend my ideas.

To Heather Martin, my non-thesis advisor.

To Massimo Banzi, for distracting me with a good project.

To Gillian Crampton-Smith and Phil Tabor, for giving these two years a solid foundation.

To Elena Baratono, for trying to teach me Italian.

To Silvia Rollino, the soul of IDII.

To Aram, Haiyan, James, and Vinay for teaching me as much as my classes.

To Paul, Doug, and Lee for their patience and insight.

To all the current and former students, faculty, and staff who made Interaction-Ivrea what it was.

Contents

1	Introduction	4
2	Background Research	6
2.1	Language-Aware Editing	6
2.2	Code Evolution	8
2.3	Testing	9
2.4	Logging	10
2.5	Traditional Debuggers	10
2.6	Tracing Debuggers	11
2.7	Algorithm/Program Visualization	12
2.8	Visual Programming	13
2.9	Live Programming	14
3	Explanatory Prototype	16
3.1	Introduction to Scheme	16
3.2	How it Works	17
3.3	Implementation	17
3.4	Discussion	18
3.5	Questions	18
4	Analysis	19
4.1	Frustrations with Current Tools	19
4.2	Breakdown of the Debugging Process	20
4.3	Other Motivations for Understanding Code	21
4.4	Interface Metaphors	21
4.5	Debugging Scenario	23
5	Interface Design	25
5.1	Function Log and Timeline	26
5.2	Live Function	27
5.3	Flagging	29
5.4	Search and Traces	30
5.5	File Listing and Dependencies	31
6	Feasibility	34
7	Evaluation	35
8	Conclusion	37

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 thought-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, the 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.

Because this work concerns reading and understanding - not writing - code, it develops tools for traditional programming languages like C/C++ and Java. The prevalence of these languages and the large amounts of existing programs written in them suggests that such tools will be necessary for the conceivable future.

This report first 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

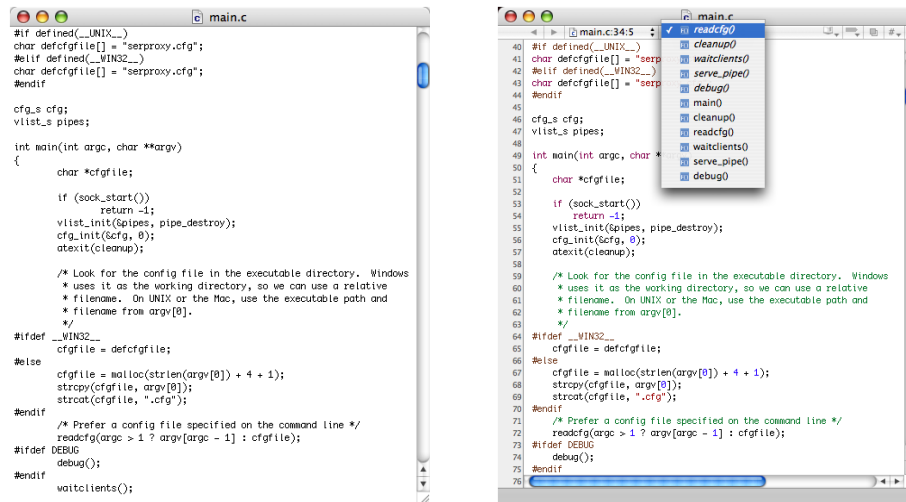
A discussion of tools for creating, analyzing, and understanding code and other methods of instructing the computer.

2.1 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 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. Some systems, like Javadoc, convert specially formatted comments into hypertext documentation.
- 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).



(a) A generic text editor with no understanding of the code being edited.

(b) Xcode parses, indexes, and formats the code being edited.

Figure 2.1: The evolution of code editors.

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

“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” 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). It allows programmers to make consistent changes across related sections of code without the cognitive overhead of restructuring or abstracting them. 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 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.

2.2 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.

In “Version Sensitive Editing: Change History as a Programming Tool,” 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.

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

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 occurrence of the bug, and coordinating edits with a programmer whose changes have not yet been approved. The software was, however, limited to basic typographic and interface capabilities.

“CVSSearch: Searching through Source Code using CVS Comments” describes a tool for searching a project’s source code and associated comments written when changes were committed to the source code versioning repository CVS. 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.

The interface is rudimentary, but the idea good. CVS comments tend to summarize and explain code changes and are a valuable reference for anyone looking to understand a program. This is a step towards making them more available when needed.

2.3 Testing

A 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. They can be used to ensure that changes or additions to a component don’t change existing behavior. They are easy to run quickly and an excellent source of examples with which to attempt to understand code.

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 developers 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.

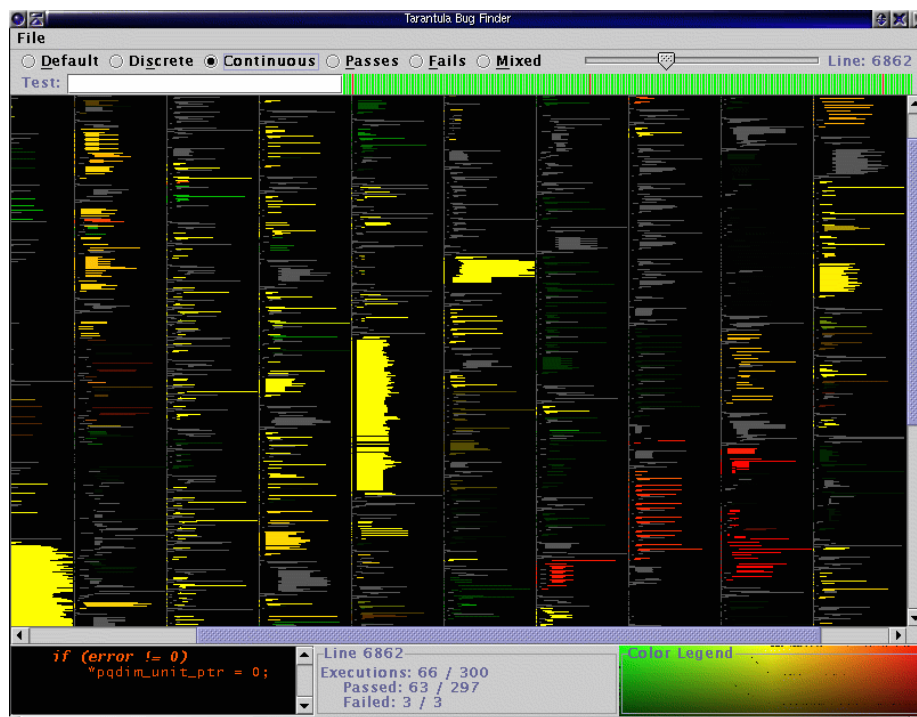


Figure 2.3: Visualization of test results from Tarantula. Lines of source code that pass tests are shown in green, failing lines in red.

Tarantula visualizes test results using the code tested (Figure 2.3). The more tests a particular line failed, the more red it is; the more passed, the greener; lines that pass and fail are yellow; untested lines, gray.

2.4 Logging

In situations in which debugging tools are unavailable or inadequate, programmers often add print or logging statements to their code. By reviewing the output, they get a sense of the behavior of the program. This technique requires little setup and can be used in a wide variety of situations. It is often, however, arbitrary and ad-hoc, with the debugging statements added each time they’re needed, and having meaning only to the programmer who inserted them. It doesn’t scale well to problems involving the interactions of many different components, nor to work in teams.

2.5 Traditional Debuggers

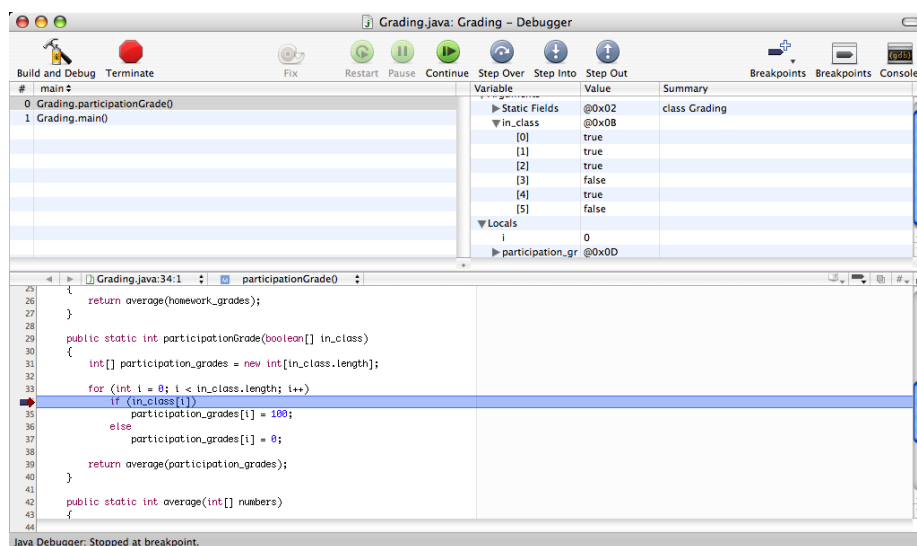


Figure 2.4: The default debugging perspective in the Apple’s IDE Xcode. Execution is paused at a particular line of code, indicated by the red arrow to its left. Above are the call stack (sequence of function calls leading to the current one) and watch window (showing the current values of variables).

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 as particular lines.

Different interfaces use this same paradigm. Originally, such debuggers were driven by textual command prompts. Now, most IDEs include a debugger with a graphical interface, but with similar functionality.

2.6 Tracing Debuggers

These debuggers instrument a program’s code and keep track of various events during its execution, such as function calls and variable assignments. 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 occurrence in the execution of a program. This allows the programmer to run a program, interact with it, load the program trace in the debugger, and explore backwards and forwards in this record of the program’s execution.

The Omniscient Debugging project has released a tracing debugger for Java, and they also exist for functional languages such as Haskell and OCaml.

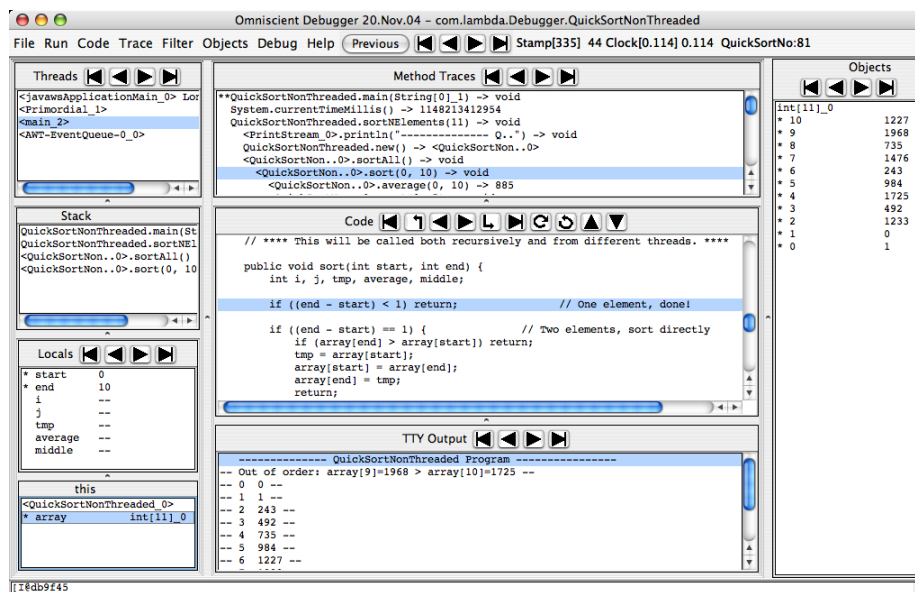


Figure 2.5: Omniscient Debugging. The left and right arrows step forwards and backwards through the execution of the program.

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 and 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.

2.7 Algorithm/Program Visualization

In 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.

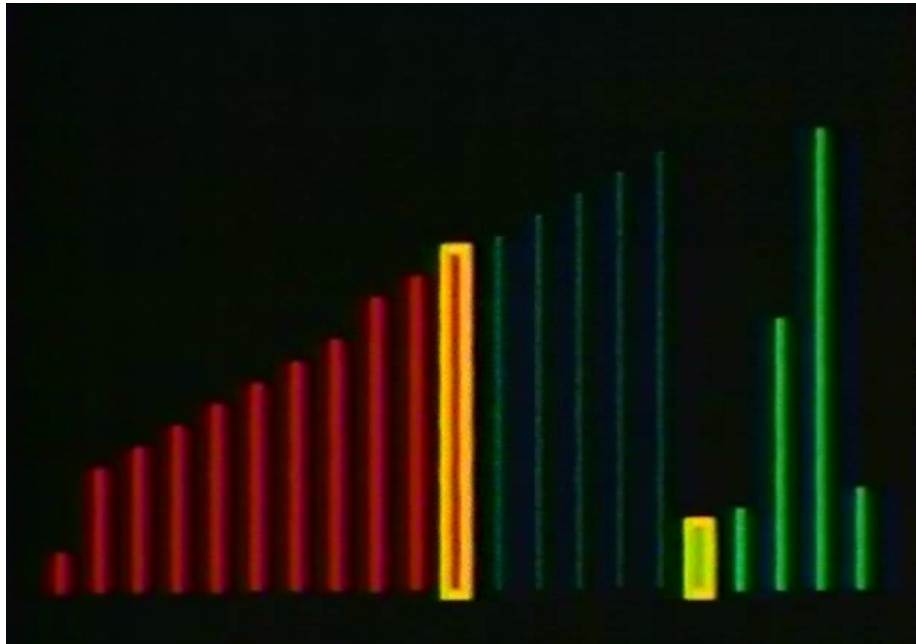


Figure 2.6: Algorithm Visualization from Sorting out Sorting.

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 understood 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 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.

One example of a program visualization is Code Profiles by W. Bradford Paley (Figure 2.7). Created as a part of the *CODEDOC* online exhibit of software art, Code Profiles attempts to provide the general public a sense of the visualized program’s activity rather than serve as a practical tool for programmers.



Figure 2.7: Code Profiles. The curves connect lines of the program’s source code in their order of execution and slowly fade out over time.

2.8 Visual Programming

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

In *Aesthetics of Computation: Unveiling the Visual Machine*, Schiffman describes several of his visual programming interfaces, including Plate, in which traditional textual constructs are placed on plates (two-dimensional movable blocks), 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. This thesis attempts to apply these principles to text-based languages.

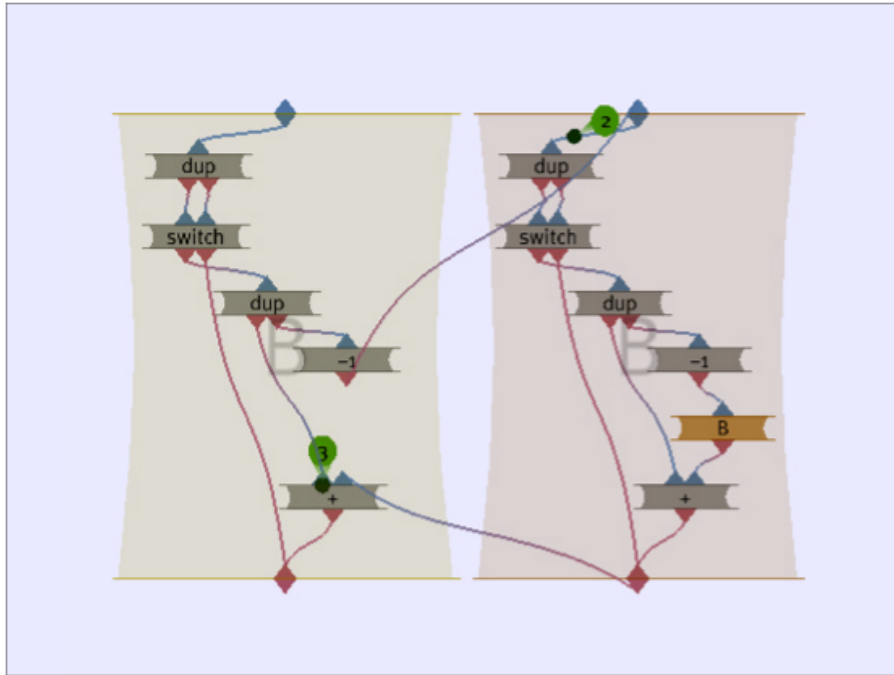


Figure 2.8: Pablo in the midst of evaluating a function. On the right, a function call has been expanded through the creation of another box.

2.9 Live Programming

Some programming environments provide a different sort of continuity by making no distinctions between the process of editing, compiling, and running code. That is, the current state of the program is visible at all times, and updates automatically as the code changes. An familiar example is a spreadsheet, in which cells containing formulas constantly recompute and display the correct value as other cells are changed (Figure 2.9).

	A	B	C	D	E
1	Part No.	Part Name	Quantity	Price (each)	Total
2	32	Screwdriver	3	\$9.95	\$29.85
3	9032	2" Nail	100	\$0.05	\$5.00
4	11384	3" Screw	50	\$0.10	\$5.00
5	13956	Sandpaper	10	\$1.00	=C5*D5
6	16898	Light Bulb	5	\$1.25	\$6.25
7				Total	\$56.10
8					
9					

Figure 2.9: Spreadsheet formulas update continuously as other cells change value.

Subtext extends these ideas to more general programming constructs. 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 be 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.

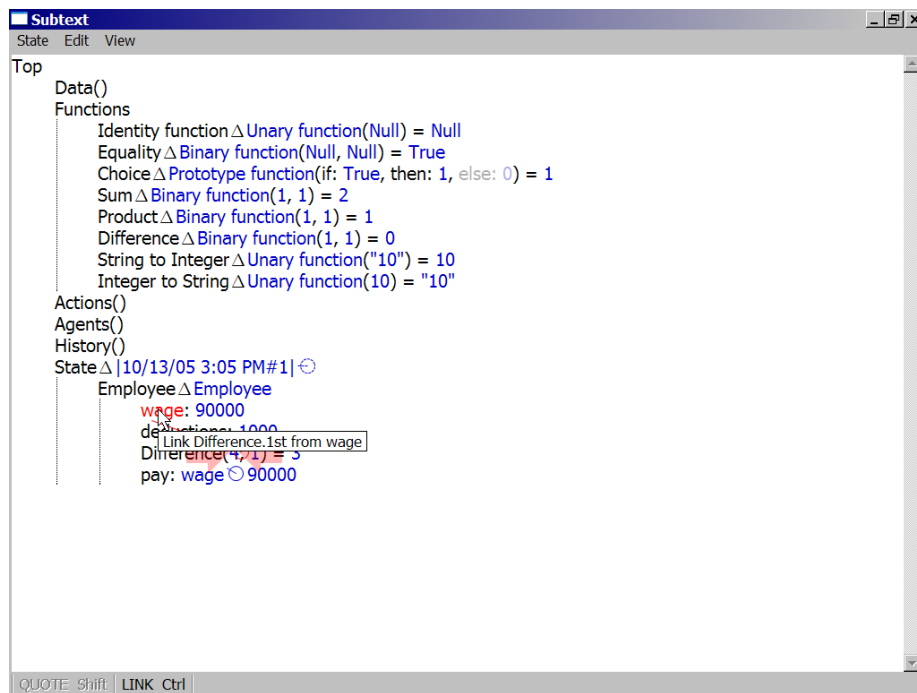


Figure 2.10: Subtext in the process of making a link.

Chapter 3

Explanatory Prototype

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.

3.1 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 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 evaluation 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.

3.2 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)    (factorial 3)
(if (= 3 1)      (if (= 3 1)      (if (= 3 1)
  1              1              1
  (* 3 2))      (* 3 2))          (* 3 2))
                                     |
                                     (factorial 2)
                                     (if (= 2 1)
                                       1
                                       (* 2 1))
                                       .
                                       .
                                       .

```

Figure 3.2: Expanding the function calls.

3.3 Implementation

Initially, the prototype consisted of a simple, hand-coded web page. 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.

3.4 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.

3.5 Questions

While this prototype delimited 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

This chapter breaks down the challenges in understanding code and presents some principles for overcoming them.

4.1 Frustrations with Current Tools

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.

Debugging using a traditional debugger can be very awkward and time consuming. 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, relieving programmers of the cognitive burden of combining many small facets of the program’s state.

The following list of frustrations was gathered from extended interviews with three professional programmers in their mid-twenties, with an average of three years of post-college experience, along with casual conversations with other programmers.

- 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.

4.2 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 of the aforementioned 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: logging (print statements)

Technique c: 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 the line of code correct?).

- 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).

4.3 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

Need to monitor a long-running program (e.g. memory usage, performance, errors, etc.)

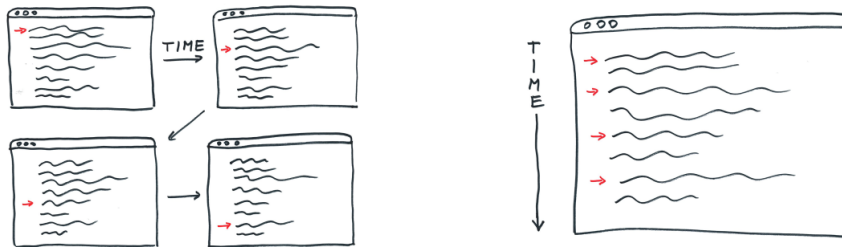
4.4 Interface Metaphors

These three metaphors introduce some of the primary differences between a traditional debugger and the Tangible Code interface described in the next chapter. They were created to help explain the concepts of this thesis to a non-technical audience.

Disjoint vs. Unified Time

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 must 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 4.1(a).

The Tangible Code environment, in contrast, presents the *entire history* of the program in an unified interface (Figure 4.1(b)).



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

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

Figure 4.1: Two approaches to handling time.

Abstract Instructions vs. Concrete Operation

Because they offer only frozen snapshots of a program's execution, traditional debuggers don't show the code operating on data. Instead, specific values tend to be separated from variables, leaving the code as an abstract set of instructions about whose operation the programmer must reason instead of observe.



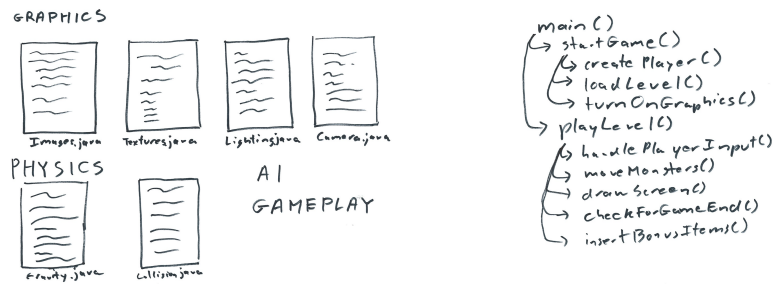
(a) Most debuggers present instructions separate from the data they manipulate.

(b) Tangible Code attempts to integrate them.

Figure 4.2: Variables and their values.

Static vs. Dynamic Structure

As discussed in the background research chapter, current tools comprehend something of the static structure of code. They do not, however, typically analyze or display anything of the dynamic structure of the code. This puts on the programmer the additional burden of inferring and remembering this structure. There's a computer present, but the programmer has to mentally run the program!



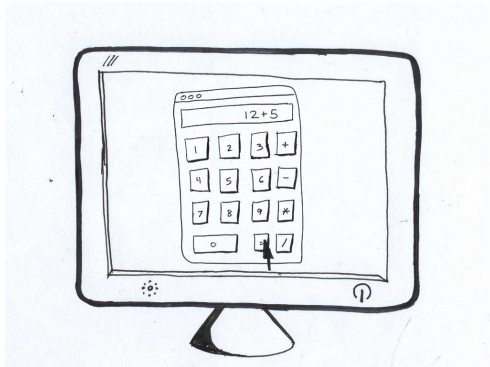
(a) Current tools understand the static structure of code.

(b) Tangible Code shows the dynamic structure as well.

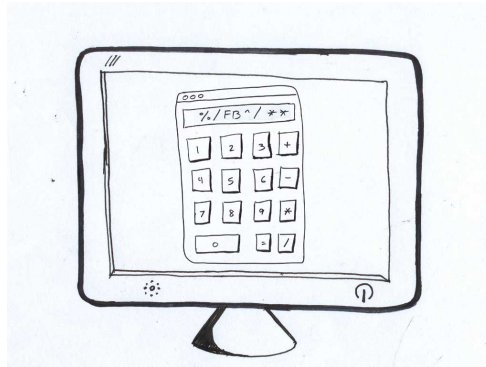
Figure 4.3: Static and dynamic structure.

4.5 Debugging Scenario

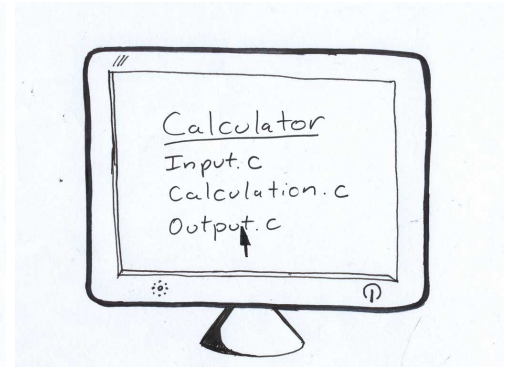
This scenario illustrates the process of debugging and points out some of the ways in which Tangible Code improves upon existing debuggers (Figure 4.4).



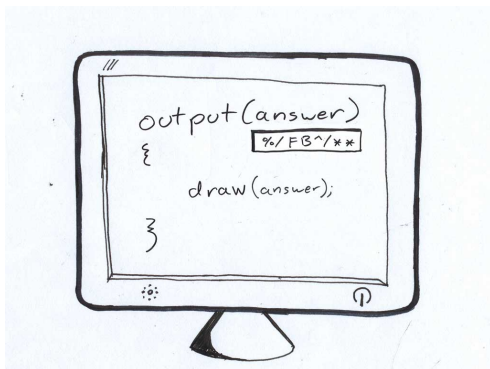
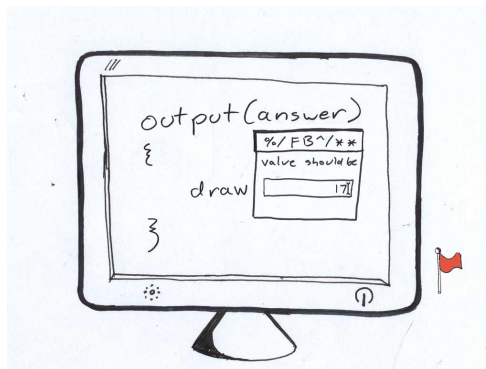
(a) Adding 12 and 5 with the calculator



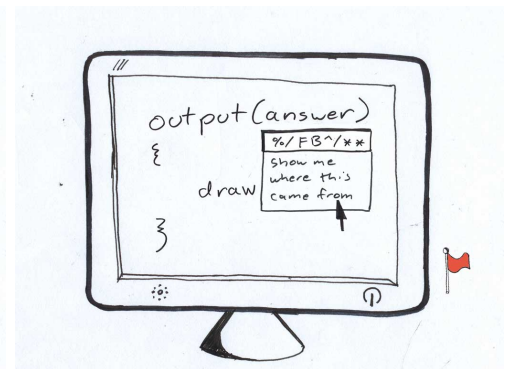
(b) The program gives the wrong answer; it must have a bug.



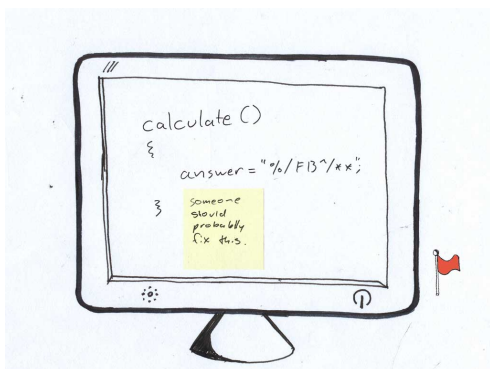
(c) A list of the files of source code used by the program; something that most current debuggers don't show.

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

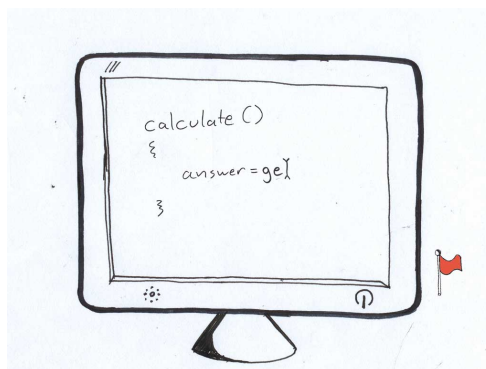
(e) The programmer enters the correct value for this variable; a red flag appears to indicate that there is currently an incorrect value somewhere in the code.



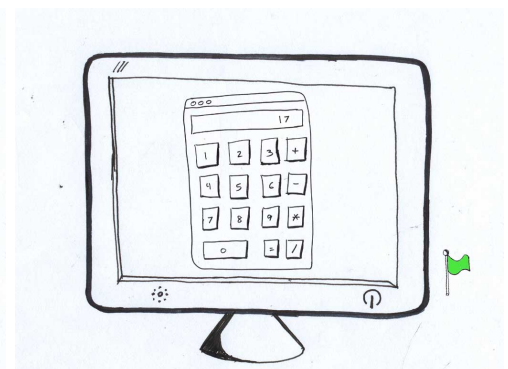
(f) To help track down the root of the problem, the tool can jump to the source of the value of this variable.



(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 4.4: Debugging scenario showing mindset and needs of programmer.

Chapter 5

Interface Design

Tangible 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. It makes tangible what were previously abstractions, giving time, values, and connections a location in the interface and makes them amenable to manipulation.

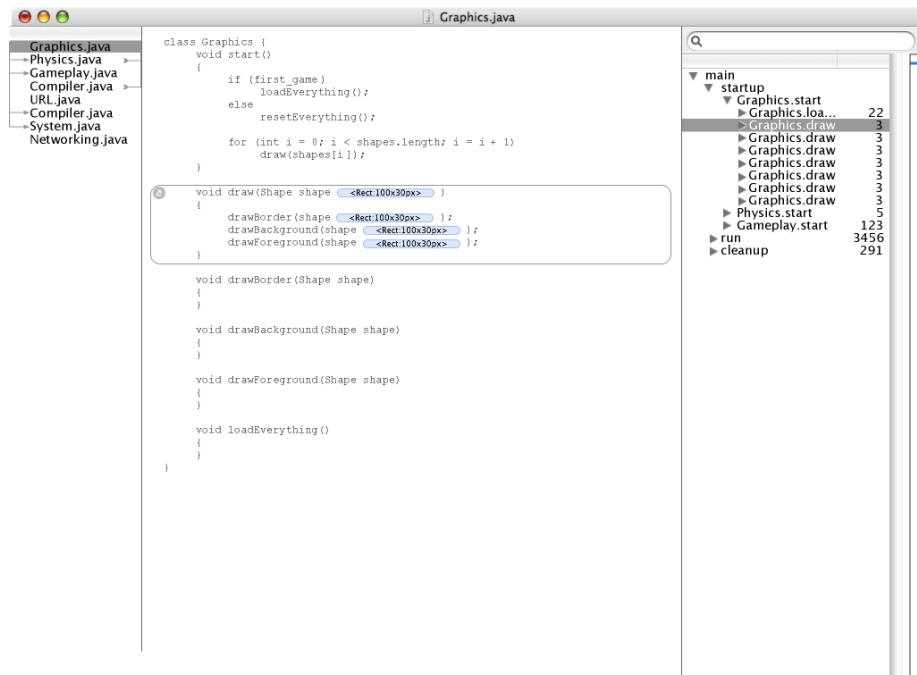


Figure 5.1: Overview of the Tangible Code software tool.

An implementation of Tangible Code would work by keeping track of every operation performed by the software when it runs, including input and output.

Then, after the execution is complete, Tangible Code analyzes this trace and presents it to the programmer. This interface puts programmers in control by allowing them to move through time as they choose, instead of necessarily following the order of the program itself.

5.1 Function Log and Timeline

Sequences of time are ordered and grouped in a few different ways. 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 which other functions forms the *function log*, the core of the program history displayed by Tangible Code. It is a hierarchical timeline of structural units. As each function may call many other functions, this record forms a hierarchical tree (Figure 5.2).

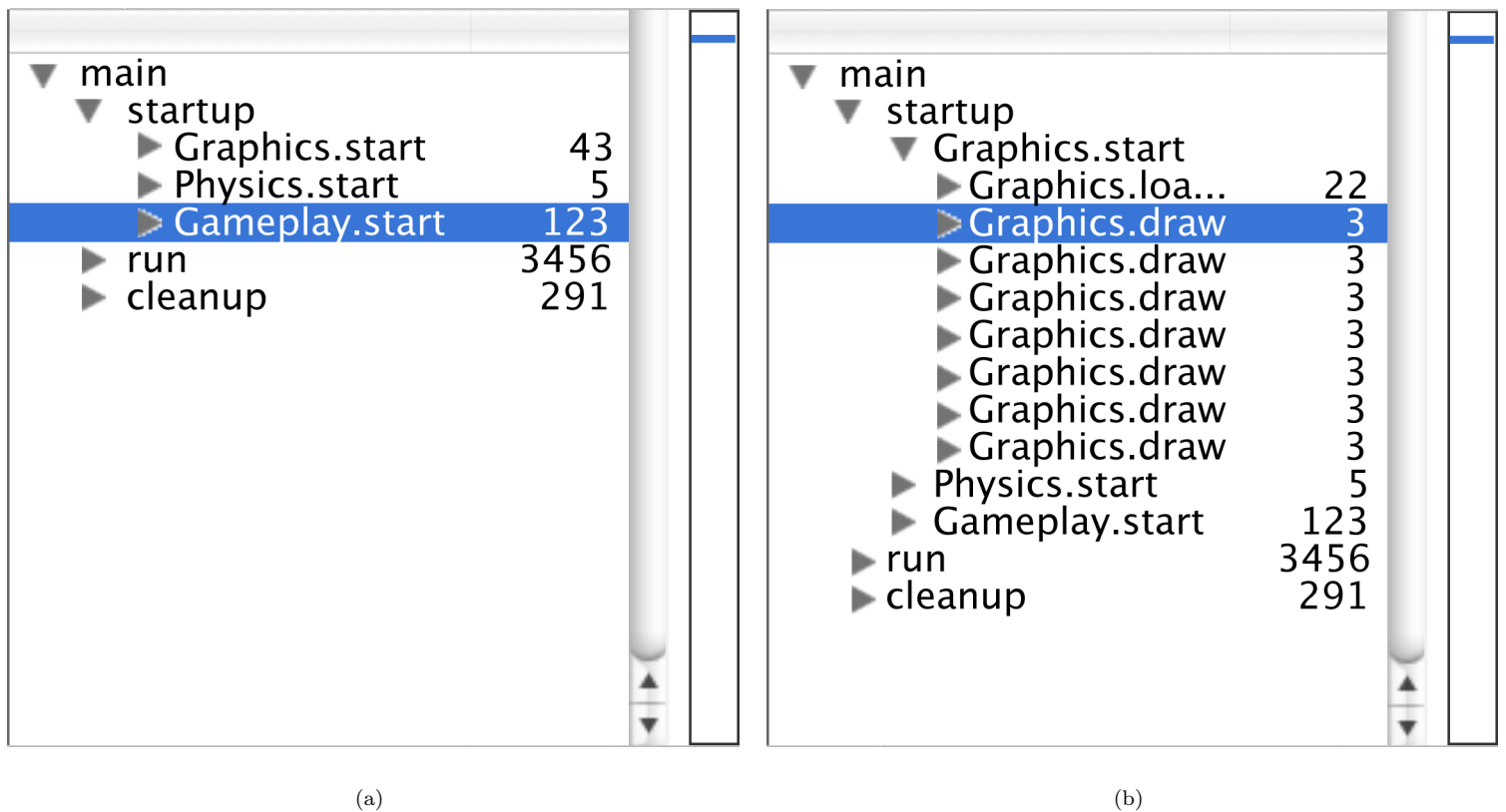


Figure 5.2: Log composed of the hierarchical sequence of function calls performed by the program. The function call named in a line occurred later in time than the line preceding it. Each function, however, may contain calls to other functions. The number on the right indicates the number of function calls nested beneath the corresponding line. The vertical white bar on the far right shows “wall-clock time”; the blue line corresponds to the selected function in the log.

The interface also handles *wall-clock time*: the actual seconds, minutes, and hours as measured by a clock independent of the program under study. A particular piece of code might take more or less wall-clock time to execute (depending, say, on the speed of the computer or the other programs running at the same time) despite carrying out the exact same computations. The vertical timeline at the right of the interface display the relative times of various events in the program.

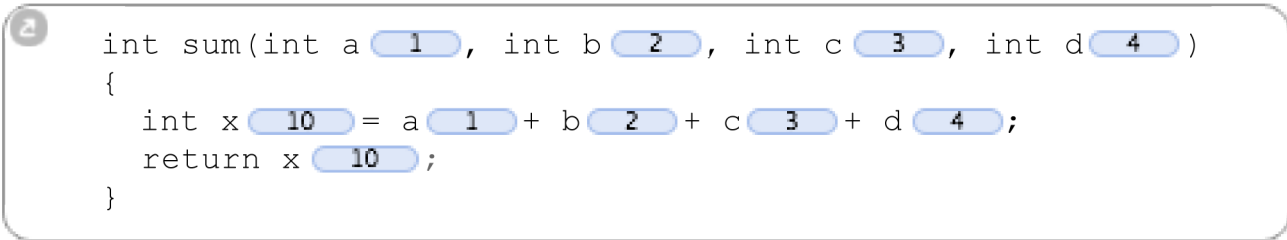
A typical file of source code provides little indication of the relative importance of its lines. One file may contain dozens of function calls, most of which perform an unimportant task and a few of which contain most of the work of the entire program. Tangible Code attempts to show this relative importance of various function calls by displaying next to each line in the function call record the number of function calls nested beneath that line. This allows the programmer to delve directly into the meat of the program without needing to scan through each part.

The function log and timeline turn time into a tangible object. Each fundamental event (a function call) receives a location on the screen, from which it can be visually compared to others and manipulated.

5.2 Live Function

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

Figure 5.3: The source code of a function: abstract instructions that can be applied to many different inputs.



```
int sum(int a 1, int b 2, int c 3, int d 4)
{
    int x 10 = a 1 + b 2 + c 3 + d 4;
    return x 10;
}
```

Figure 5.4: 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.

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 5.2(b), `Graphics.draw()` was called repeatedly.) To view a particular call to a function, the programmer selects the corresponding line from the function log. Because this line refers to a precise moment in time, the data being operated on can be shown. I refer to this function call and associated data as the *live function*.

Within the live function, Tangible Code displays a variable’s value alongside it, using GUI widgets to embed these values into textual source code (Figure 5.4). 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.

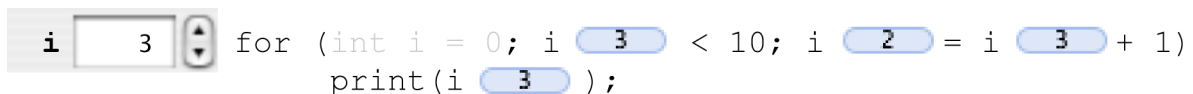
The live function makes the program’s operation tangible, allowing programmers to follow the sequence of computations by simply reading the code. To see the past, they need only look back to preceeding lines.

Within the live function, execution typically proceeds down through the code, one line at a time. Conditionals (like `if`-statements), however, cause certain lines of code to be executed only under given conditions. In Tangible Code, unexecuted lines are grayed out. Removing them completely would disorient programmers familiar with the code and expecting to see those lines.

```
if (x 25 < 100) {
    return x 25 ;
} else {
    return x * 2;
}
```

Figure 5.5: An `if`-statement. The condition `x < 100` is true, so the first block is executed and the second skipped.

Loops can cause some lines of code to be repeated multiple times, each time with potentially different values and behaviors. Tangible Code provides an interface component for cycling through each iteration of a loop (Figure 5.6).



```
i 3 for (int i = 0; i 3 < 10; i 2 = i 3 + 1)
    print(i 3 );
```

Figure 5.6: A loop whose body runs 10 times, each time giving `i` a new value. Editing the text box or clicking the arrows changes the iteration shown.

The live function occurs somewhere within a chain of function – those that called it and those which it calls. Tangible Code makes it easy to navigate between functions. Because the live function refers to a particular moment, so too does each call within it to another function. Clicking one of these jumps to that function’s code, and *makes it the live function*. That is, the function call itself becomes a tangible connection – there is no need for the programmer to manipulate auxiliary controls or to find a corresponding line in another list, lessening cognitive burden and allowing the programmer to concentrate on their task, not the navigation.

Clicking the arrow at the upper left of the live function returns to the function that called it (i.e. its parent). After jumping to the parent function’s code, an outline zooms out from the statement that calls the child function. This guides the programmer’s eye from the previous live function out to the current one, showing where the environment jumped from (Figure 5.7).

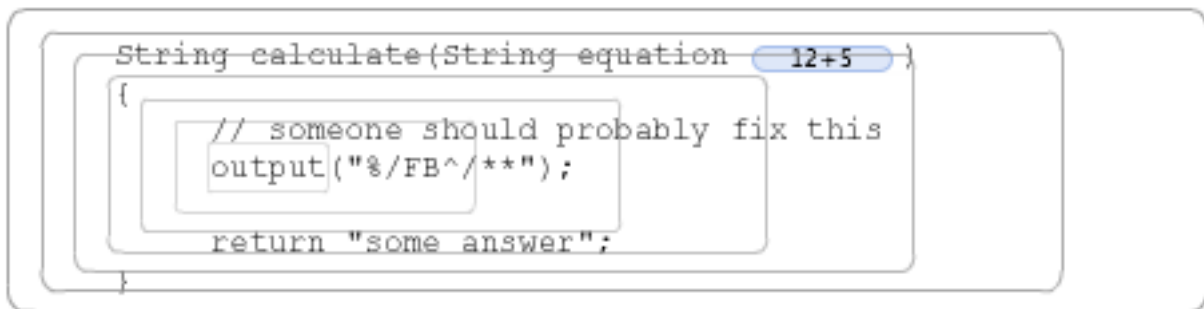


Figure 5.7: An animated outline guides the programmer's eye out from the previous live function – the call to `output` – to the new one, `calculate`. This is an overlay of multiple positions in the outline's movement – onscreen, the outline would move continuously between them.

5.3 Flagging

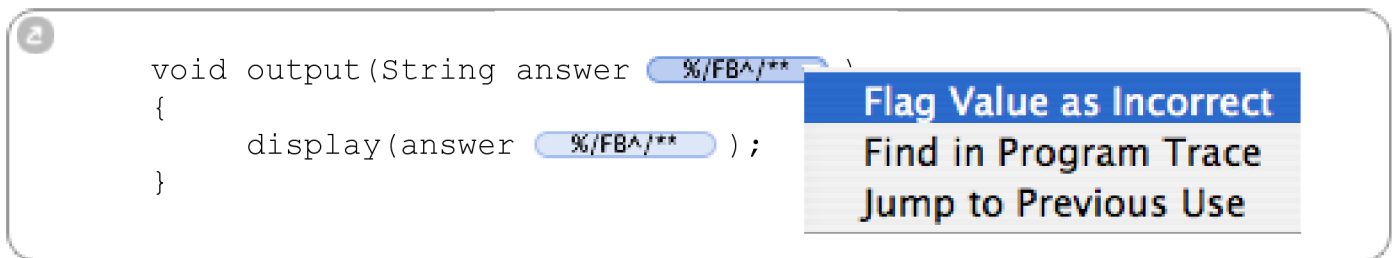
▼	main	
▼	startup	
▼	Graphics.start	
▶	Graphics.loa...	22
◆	Graphics.draw	3
▶	Graphics.draw	3
▶	Graphics.draw	3
▶	Graphics.draw	3
▶	Graphics.draw	3
▶	Graphics.draw	3
▶	Graphics.draw	3
▶	Physics.start	5
▶	Gameplay.start	123
◇	run	3456
▶	cleanup	291

Figure 5.8: A solid red diamond indicates a flagged value inside the corresponding function; a hollow diamond means that there is a flagged value in a function nested beneath the corresponding function.

As in the debugging scenario above, a programmer may sometimes know that the value of a variable at a particular point in the program's execution is not what it should be. The ability to *flag* this location establishes it as a reference – both for precisely communicating a problem with others and for

checking whether changes to the code fix the problem.

In Tangible Code, the value itself becomes a control for performing actions on it, including flagging. Clicking it drops down a menu – the flagging command turns the value red. If, as in the scenario, the programmer knows the correct value of the variable, that can be entered next to the current, incorrect value. Once the code has been fixed and the flagged value is correct, it turns green. It can then be kept as a check during subsequent code changes, or unflagged.



(a) Clicking a variable value drops down a menu.

```
void output(String answer %/FB^/** )
```

(b) Flagged values are shown with a red background.

```
void output(String answer %/FB^/** 17 )
```

(c) If known, the correct value can be entered.

```
void output(String answer 17 )
```

(d) When the flagged value is correct, the background turns green.

Figure 5.9: Flagging a variable.

All flagged variables are also shown in the function log and the timeline. This enables the programmer to see the locations and amount of incorrect values, and to tell when they have been corrected.

5.4 Search and Traces

Tangible Code provides a simple search feature with a wide range of uses. Possible searches include files, classes, functions, and variables. Search results are shown in the function log and timeline (Figure 5.10). Their quantity and distribution provides the programmer an indication of the way in which that item is

used in the program: initialization functions will occur only near the beginning, whereas a low-level class might be almost uniformly distributed across the entire timeline. Additionally, by entering expression (e.g. `x < 0` or `person.firstname == "David" && person.age == 26`), the programmer can narrow in on particular areas that may contain problems or potential problems. Giving the search field the same syntax as the programming language allows programmers to use their knowledge of it to construct complex search queries.

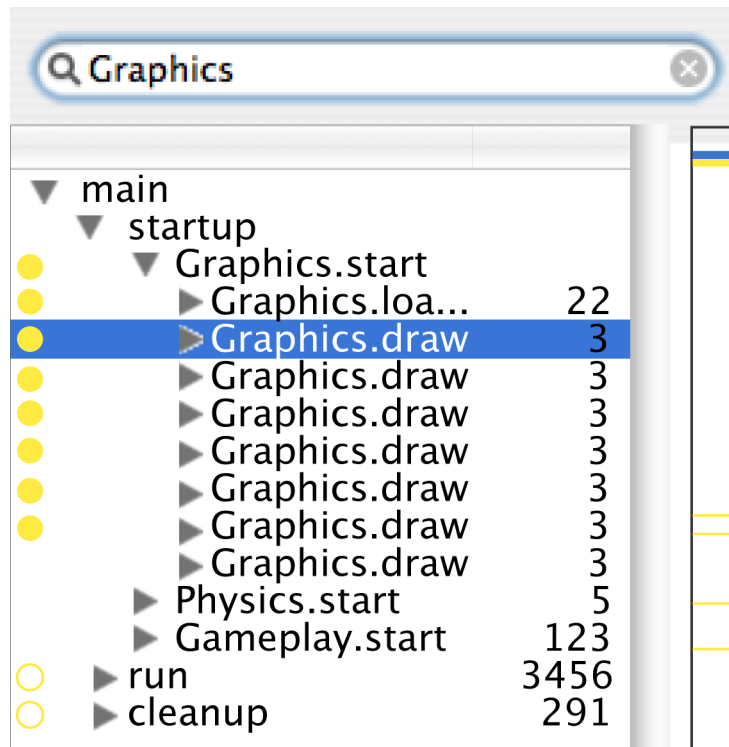


Figure 5.10: Search results shown in yellow. Solid circles represent a result in the corresponding function itself, hollow ones a result nested within the corresponding function.

Also, the function log can show the changes to a variable over the course of the program (Figure 5.11). This enables a programmer to see the precise lifetime of a variable and to locate the beginning or end of a particular problematic value (e.g. `null`). It also enables comparison between multiple variables. Here, the history of the values of a variable become a tangible, on-screen entity.

5.5 File Listing 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

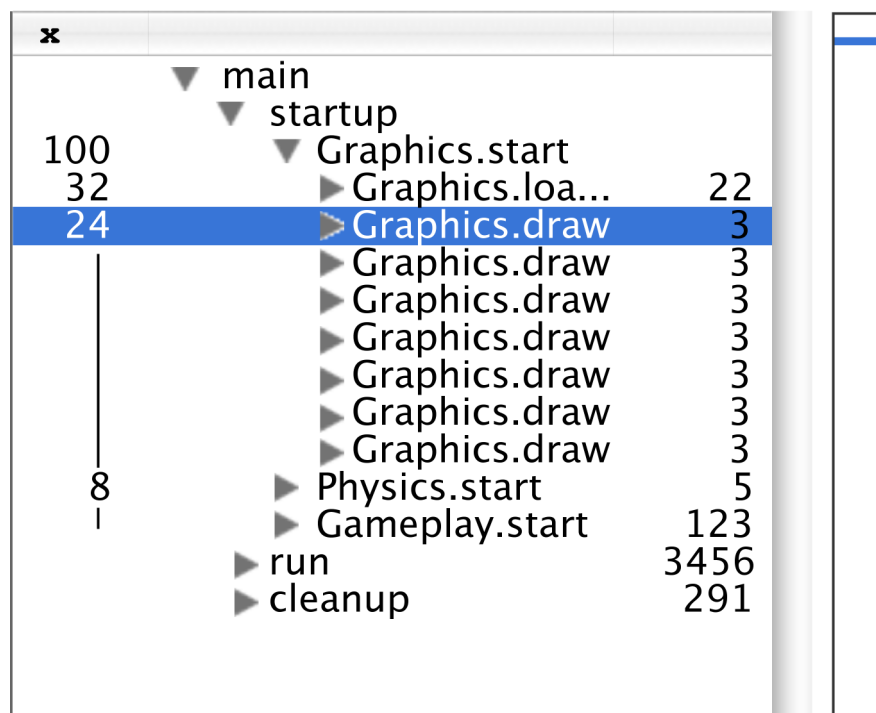


Figure 5.11: A variable trace. The vertical lines indicate the variable's value remained constant. Blank space means the variable didn't exist at that point.

over. Or a programmer about to change a piece of code might want to review all the places where that code is used. Tangible 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 to the files it uses, and from files that use it. If one of the functions in the file is currently live, the file is highlighted in blue, and the files used by that particular function are pointed to with blue arrows. (Figure 5.12.)

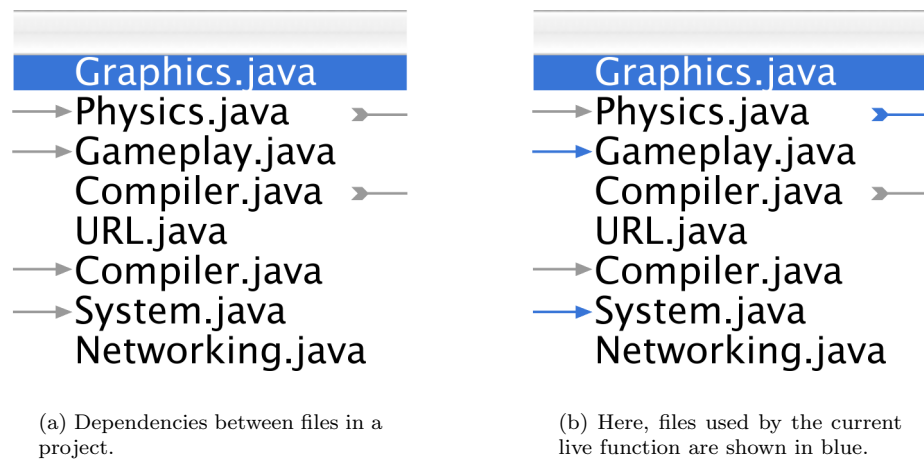


Figure 5.12: Showing dependencies between parts of a program.

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 preceding 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.

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.

Tangible Code makes economic sense as well. Programmers time is expensive and not easily substituted. As Brooks famously wrote in *The Mythical Man-Month*, adding more programmers to a project tends to make it later. It only makes sense to improve the productivity of programmers through better tools.

Chapter 7

Evaluation

The design of Tangible Code underwent a continual process of evaluation and refinement, from initial sketches (on paper and screen), wireframes, detailed mockups, and animations. This ensured that the process was cumulative and that little work was wasted.

Feedback on an earlier version of the wireframes was received from seven programmers (mid-20's to mid-30's, with at least a few years of professional programming experience). They expressed approval of the general direction and made specific suggestions and feature requests. Some have been incorporated into the above interface design.

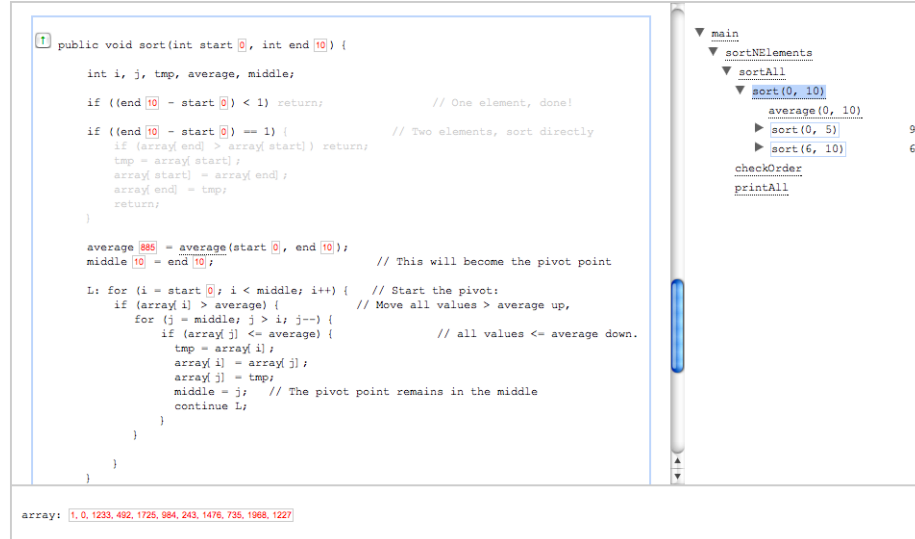


Figure 7.1: An HTML prototype of Tangible Code operating on the buggy quicksort example from the Omniscient Debugger.

An HTML prototype (Figure 7.1) elicited more specific responses in extended interviews with two of these programmers. The prototype provided the core functionality of the Tangible Code interface, allowing the respondents to

simulate the experience of using it to debug a particular program. The prototype's approximation of the desired appearance and behavior, however, led to some confusion about the meaning of some parts of the interface. The function log, for example, was not immediately perceived as a record of an already completed program execution. This difference from the presentation in traditional debuggers of an individual snapshots of time represents a fundamental shift and requires an adjustment in the perspective of the programmer. From this preliminary user feedback, however, it seems that once the shift is made it is a natural and readily accepted approach.

Many possibilities exist for expansion of the Tangible Code interface. The timeline, for instance, could be extended – one per thread – to display time-slicing and locking. Integration with the GUI of the program being inspected would allow a programmer to see the code triggered by a particular action in the interface. Storing various traces with the corresponding revisions of code – and displaying differences between them – would allow programmers to easily understand the effect of code changes on the behavior of the program.

Through the principles expressed in the metaphors in the analysis chapter, however, Tangible Code helps to relieve the cognitive burden on programmers as they debug or otherwise read code. The use of tangible time, tangible values, and tangible connections give form to what were previously abstractions in the mind. Code and its behavior become manipulable entities – you can practically touch them.

Chapter 8

Conclusion

This thesis has required repeated explanation to a non-technical audience. The visual metaphors and scenario above are just two examples of the methods used. This constant effort had great benefit – it forced me to breakdown the principles, pieces, and accomplishments of my work in order to explain them. It also helped me maintain my focus, for despite many requests to create a tool for non-programmers (and frequent misconceptions that such was actually my goal) – this specialized effort seems to have yielded some general insights about the ways in which abstract concepts can be translated into tangible components of an interface.

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*. Springer-Verlag: Berlin. 1998. 146–157.
- [3] Baecker, Ron, Chris DiGiano, and Aaron Marcus, ‘Software Visualization for Debugging’. *Communications of the ACM* Vol. 40, No. 4, Apr. 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. Springer-Verlag: Berlin. 176193.
- [5] Edwards, Jonathan, ‘Example Centric Programming’. *OOPSLA04*, 24-28 Oct. 2004.
- [6] ———, *The Future of Programming*. 6 Dec. 2004. Alarming Development. <http://alarmingdevelopment.org/index.php?p=6>
- [7] ———, ‘Subtext: Uncovering the Simplicity of Programming’. *OOPSLA05*, 16-20 Oct. 2005
- [8] Goldsmith, Simon, Robert O’Callahan, and Alex Aiken. ‘Relational Queries Over Program Traces’. *OOPSLA05*, 16–20 Oct. 2005.
- [9] Jones, James A., Mary Jean Harrold, John Stasko, ‘Visualization of Test Information to Assist Fault Localization’. *Proceedings of the International Conference on Software Engineering*, May 2002.
- [10] Paul, Christiane, curator. *CODeDOC*. Whitney Artport. Sept. 2002. <http://artport.whitney.org/commissions/codedoc/>
- [11] Petzold, Charles. *Does Visual Studio Rot the Mind?*. 20 Oct. 2005. <http://charlespetzold.com/etc/DoesVisualStudioRotTheMind.html>
- [12] Saff, David, and Michael D. Ernst. ‘Reducing Wasted Development Time via Continuous Testing’. *Fourteenth International Symposium on Software Reliability Engineering*, 17-20 Nov. 2003. 281–292.

- [13] Schiffman, Jared. *Aesthetics of Computation – Unveiling the Visual Machine*, Thesis: Master of Science in Media Arts and Sciences, MIT, 2001.
- [14] Spencer, Rick. *Typical Usability Problems with Debugging in C#*. 20 Mar. 2004. ricksp's weblog. 10 Apr. 2006. <http://blogs.msdn.com/ricksp/archive/2004/03/30/104168.aspx>
- [15] ——. *What Is Typical C# Debugger Usage?*. 29 Mar. 2004. ricksp's weblog. 10 Apr. 2006. <http://blogs.msdn.com/ricksp/archive/2004/03/29/101410.aspx>
- [16] Toomim, Michael, Andrew Begel and Susan L. Graham. 'Managing Duplicated Code with Linked Editing'. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Sept. 2004.
- [17] Tung, Sho-Huan Simon, 'Visualizing Evaluation in Scheme'. *LISP and Symbolic Computation*, No. 5. 1997. 1–23.