Statement of Intent

Interaction Design Institute Ivrea

David A. Mellis

September 15, 2005

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.

As a programmer and interaction designer, I want to continue to do just that - apply HCI techniques and principles to the design of programming tools. To create scenarios that illuminate the tasks, needs, and capabilities of programmers. To design tools that account for those factors. To test and refine those designs. To point out areas in which even faster machines or smarter algorithms are needed. To help expose the inner workings of our programs. To build better castles in the air.

David A. Mellis – IDII – September 15, 2005

Research Questions

1. What tools and techniques do working programmers use to understand the structure and functioning of a program?
2. How can the principles of interaction design be applied to the practice of programming?
3. How can the programmer's view of a program – its source code – provide an interface to the computer's view of a program – its execution?
4. How can programming environments support exploration and experimentation (in addition to abstract reasoning) in the creation and understanding of code?

Context

This work concerns itself with professional programs working on real software. Designs will be evaluated not on their own merits, but considering their usefulness on actual programs. That said, research does not require complete, robust, and scalable implementations but can be fruitfully applied to prototypes or even non-interactive designs. The principles thus learned must be applicable, however, to the construction of commercial-quality software.

Project Plan

15–30 September	Background research collection and documentation.
October	Exploration I: design and prototype.
November	Exploration II: design and prototype.
December	Evalution, reflection, and proposal for primary prototype.
January to April	Implementation of primary prototype.
May	Evaluation, reflection, and documentation.