The Perils of Prototyping

by Alan Cooper

Which is harder to change: a program with 1000 lines of code or a 1000 square foot slab of concrete? The concrete is ten inches thick and has steel reinforcing rods criss-crossing within it. Every cubic foot of it weighs almost 100 pounds. The software has almost no physical existence at all. It weighs nothing. It consumes no space. A few microamps and those bits flip from zero to one without a second glance. The answer to my question seems a simple one, doesn't it?

Which is the best medium for designing software: Visual Basic or a sharp pencil and a couple of sheets of paper? Visual Basic is a powerful, flexible integrated development environment. It is on its way to becoming the most widely used language ever. It has won every industry award there is. Paper is not interactive. Paper offers no palette of pre-made controls. It just lays there and you have to do all of the work. The answer to my question seems a simple one, doesn't it?

Considering the first question, if you said the code was easier to change you are dead wrong. In order to change a slab of concrete all you need are Jim and Larry, a jackhammer and a six-pack of brewskis for Miller Time. Jim and Larry will never ask "why?" You don't have to justify the changes to them to get them to work. They'll just start bustin' and stop when they're done. Whereas to change the code, you have to deal with a programmer. You have to convince that programmer to change it. He will be reluctant to break working code. She will need to have things explained. He will disagree with your reasoning. He will be argumentative. She will give you several difficult-to-dispute reasons why you are wrong. Even after you extract agreement he might change his mind without consulting you and do something different than you expect.

Believe me, it's easier to break concrete than to change code. It's like that 70's riddle: How many Californians does it take to change a lightbulb? Just one, but the bulb has to want to change! We are dealing with more than just the physical world here. We have to deal with the programmer's mind-a much more resistant material than mere steel-reinforced concrete.

Regarding the question of prototyping versus paper as a design tool, if you chose VB you are wrong again. Everybody knows that Visual Basic is a great tool for prototyping, so many VB programmers design user interfaces by prototyping them. Daily, I talk to people excited about the great prototypes they are building. While they exclaim, I shudder, because these prototypes confuse and confound the design process. They are misinterpreting programming for design, and make no mistake about it: prototyping is programming and it is harder than concrete to change.

Most software designers and developers sincerely enjoy prototyping. That joy they experience is the joy of construction and not the joy of design. I have personally coded hundreds of thousands of lines of code (many of them for prototypes) and I love it too. It is a very creative and intellectually satisfying activity. It is not software design! Software design happens when you turn off

the computer. Software design is a dynamic process demanding instant flexibility, adaptability and openness to change. Software design is a complex, user-centered process involving continuous trial and error. Programming is a complex, computer-centered process involving continuous trial and error of an unrelated sort. Although code is constantly changing, it is very resistant to change from outside forces, and in the context of programming, the user's needs are an outside force.

Alright, I'll admit that this is stated in high-contrast black and white, and that the reality is more fuzzy-edged than I let on. But why begin your journey by nailing your foot to the floor? Sure it will still be possible to get there, but only slowly, painfully, and you'll arrive with a limp.

Software has a life of its own. It is full of little quirks and idiosyncracies that influence how it is built. This is true whether the tools are high-level (VB or Hypercard) or low-level (asm, C). Some things are easy to build and other things are hard to build. That this strongly influences the design is almost universally unrecognized. If you take the great risk of building something divergent from the tool's strength, you will quickly find yourself in a minefield of tactical choices: "Hmmm, let's see, if I code it this way the user presentation is better but then I have to build my own , or I could code it the other way and have this operational conflict with this store-bought gizmo that I need to make this other part of the program easy to do, or I could roll my own and then it would be correct but I won't be able to test it for six months, or I could..." None of this has anything to do with software design!

Software prototyping tools have a very limited repertoire. Whatever you build in Hypercard will always look like it was built in Hypercard-it will reflect the inherent structure of the cardstack idiom and it will contain only those gizmos written into the program eight years ago. Likewise, whatever you build in Visual Basic will almost certainly emerge in the form of a menu/dialog/toolbar/commercially-available-gizmo program. The tools severely limit the designer's palette and hamper her creativity.

The internal construction of software is very complex. In order to express yourself as a software designer in code you must grapple with extremely complex tools and work in a medium of unequalled complexity. Heart surgery may be more tense and more risky but it's less complex than your average shrink-wrapped Windows application. By their very nature, the tools will unavoidably distract the designer from the real issue at hand: being an advocate for the user.

Programmers are professional complexity-wranglers. They love details and complications and it takes a mind like that to create software. The users on the other hand, hate details. Users detest complexity and will always prefer simplicity when it is offered. Programmers are happier dealing with details than with the big picture stuff. Details are easier to wrangle with; they tend to have more deterministic answers; they tend to have more polarized answers with less nasty gray area; they tend to have more written about them; they lend themselves to empirical testing; they tend to be technique-oriented, therefore very interesting and absorbing to programmers. Compared to getting the big picture stuff right getting the details right is easy. The latter is often done at the expense of the former. Id est: the flight was flawless but the plane landed at the wrong airport; the program works well and even tests well, but user's won't buy it because it doesn't do the right thing.

Just how do I design if not with prototyping? An excellent question. The short answer is "on paper." Visual presentation author Edward Tufte extolls the generous bandwidth and flexibility of paper as a communications medium. He's right. The tools currently available to sketch screens on screens are terrible, and you hobble yourself enormously if you can't visualize screens and their behavior without actually seeing the pixels. To be a successful software designer, you must develop your visualization skills. If you can't, you won't succeed just as an astrophysicist who couldn't visualize with just a few sketchy models would be an abject failure as a scientist.

Of course, this begs the question: how does one learn to visualize? And the answer is, in large part, by writing software and seeing what it looks like. You have to have plenty of up-close and personal experience creating software in order to visualize. However, as noted above, creating software is not a particularly efficient method of visualizing software. It's like saying to be a good General you must have combat experience, but that doesn't necessarily make a battle a good place to General from. This point is mistaken in many academic institutions where most of the software writing experience of the students and researchers is only in prototyping. They confuse learning to visualize with the act of visualizing and draw the erroneous conclusion that prototyping is designing.

So what good is prototyping? There are two good reasons for prototyping, one of which I'll tackle here: Prototyping is a manufacturing proof.

When an object to be manufactured leaves the drawing board it is then prototyped-not to see if it will work or to see if people can use it but to see if it can be manufactured! The prototype demonstrates the feasibility of construction techniques: Can this piece be formed by a five-ton press or do we need a ten-ton press? Can a drill press actually reach into this casting where we need a hole? What sequence must be used to assemble this subassembly? Must this piece be cast or forged? Stamped or machined? Delrin or aluminum?

And this is exactly why a software prototype should be built: To answer questions like: Is sequential access fast enough for this collection of data? Can we squeeze enough bytes down the LAN fast enough? Exactly what nasty implementation problems are hidden in building a class library? What side effects will we encounter if we break the program into loadable phases? Heap or linked list? C++ or VB? Did you notice that all of these issues are architectural issues related to performance-they don't address user interface issues at all? Prototypes are manufacturing tools, used by the builders. They are not design tools used by designers.

With today's awesome computer size and power the performance imperative just isn't that dominant any more. Our programs will have decent performance regardless of how efficient we are and how diligently we shave microseconds. This means that many applications can be reasonably built without going through the prototype stage at all. If there are no significant technical mysteries in the application and no looming performance bottlenecks, why bother?

Prototype code has an annoying susceptibility to live longer than was originally planned. The same tendency that makes code harder to change than concrete applies when it comes time to do as Fred Brooks exhorts us and "plan to throw one away." How often have you seen experimental code, originally destined for

a brief test, incorporated in the shipping product? This malignancy calls for either unnatural will-power or complete abstinence. Knowing programmers as I do, take my advice: abstinence is better.

On your next design project, try to analyze what you would normally expect from a prototype and find other ways to get it. Use the whiteboard or try good 'ol paper. You will find it refreshingly easy to toss out a bad design when it is just a half dozen sheets of foolscap and not three days worth of hard-won VB logic. You will find that after a week of wrestling with user scenarios on paper the stack of rejected ideas and half-baked concepts is a tall one indeed. And you will find that the one design that you keep from that week is better than the one you would have gotten had you tried to manifest it in code, even in prototyped code.

Last updated January 6, 1997
Talk back to us!