Report on Software Issues

To understand good programming, we must create programs that do the world some good ...

by Greg Bryant

Presented March, 1997 in Aspen, Colorado to the 'Gatemaker Salon': Christopher Alexander, Bill Joy, Peter Gabriel, John Seamster, Richard Gabriel, Mike Clary, Emily Suter

***

The most pressing question in engineering is: "how do you keep your eye on the *whole* while creating its *parts*?" The answer is simple: make sure that each new part adds to the whole. Natural systems achieve this through a fundamental process, which creates robust and coherent structures of enormous complexity. This process is now visible, and potentially available for the construction of software.

***

Our goal was to create a computer-based tool, in under three months, which helps someone design something good for the environment.

During development, it was important to know if the tool was growing in the right direction, and this was done through careful evaluation as it evolved. The emerging tool had to feel right, because the user has to feel good about what they produce *with* it. But we found that this flew directly against standard practice in software development.

Normally, software applications are evaluated after rather large steps, each of which involve quite a lot of work. Typically, the question then posed is: "does this meet the specification or milestone?" If the specification needs to change, a change request, spoken or written, is made. This happens a great deal, and the changing nature of specifications has become an understood fact in the industry.

When software is evaluated, a straightforward question is: "has this chunk of effort truly empowered the user in the most effective way?" But if the answer is "not really", then a great deal of work may need to be undone. So the tendency is to accept it as long as it works *somewhat*.

So, *large* efforts between evaluations naturally lead to programs we *can't* like, because someone saying "I don't feel good about this step" will tend to be ignored when the cost of undoing something is considered. Consequently, there are many bad products.

This process puts the *programmer* in a bad state as well. If he follows the current spec to the next milestone, he will naturally tend to code until that feature is testable. He may do a few experiments, look a few things up, but then he designs the whole feature as best he can and starts coding. Then he debugs it.

It's a messy approach, really. It's not particularly pleasant or satisfying until the whole feature is working. The result tends to be code that the programmer doesn't really care much for, as well as behavior that doesn't really shine in any way. But it meets the milestone.

If, instead, one made *much smaller* efforts in programming, the results could be judged, and kept or discarded, without undue loss of time. This would make for better products. Also, since true satisfaction comes from a good result, as well as a good structure supporting it, the approach of engineering by *small, complete efforts* offers the programmer many more opportunities to feel good.

Preliminarily, we have found that steps to a program that is wonderful, on the inside and out, must be delightful, *surprisingly small* efforts (of, say, just 1-5 lines of code, or adjustments of code, or macros), *which must then be compiled and run*. At that point there is an evaluation -- does it *work*? Does it improve the *whole*? Does it improve *both* the structure of the program *and* its value to people? If not, wipe it out: it's a small effort (although potentially of large effect), and simple to undo. If it *does* work, then go on ... and again find the *next most important, effective, well-leveraged small effort needed for the whole.* This is then compiled, evaluated, kept or tossed ... etc.

This is a major aspect of the fundamental process of design, applied to programs. Work done this way is enormously rewarding, and results in something incalculably better than building to large preconceptions or plans. It lets engineers concentrate all their ability into small, brilliant moments.

A smallest-best step process helps to link many common sense ideas. For example, Donald Knuth, in *Literate Programming*, has suggested that the mental state of commenting a program must be considered separate from that of creating one. But *when*, then, does one comment? The answer here is very simple: in the evaluation stage after each small effort.

After making a smallest-important step, compiling, and judging it to be successful, one is very much in the mood to pat oneself on the back and describe the latest engineering conquest. It's a perfect slip into that mode Knuth is describing. It's an easy time to make certain that one's eyes are pulled back to see the

whole, including one's own performance. Commenting at that point means that one doesn't have to write reams of comments at the end, or write them while *programming*: the first is a pain and the second is disruptive. Most importantly, commenting then becomes a supportive, natural prelude to taking the *next* important step, which requires a whole view.

Comments then help chart the growth of the product. This is very broadly useful.

In Gatemaker, the user can review the unfolding of gates by themselves and others. One can imagine an analogous tool for software. The step-by-step comments mentioned above, along with the emerging states of the program, can be stored together to show the growth of the software. By watching this kind of software grow, programmers can review the structure of the system very easily. They need only understand one step at a time! Adele Goldberg is making similar proposals to project teams taking on new members: in order to get them up to speed, have them review the *evolution* of the project in detail. But this approach can only be successful if the team takes steps that are small enough to understand, and small enough to be *seen unambiguously as good.*

Re-usability and technical transfer are greatly facilitated when one can watch the growth of an object. Imagine some lovely place in a city, which works wonderfully, and you want to understand it. Watching its evolution, where it first emerged, how it's used, how it was made and adjusted and for what reasons, will tell you how to build that same sort of thing where *you* live, in such a way that it will be unique to your context, but still benefit from the experience of others.

Any process animated in this way can be better understood. But a process can be *easily* understood only if the evolution took place through clear, excellent steps.

So, imagine a wide number of programs, built by small, brilliant steps. You know that a program does X, and you need to do something like X too. Normally, reading the code tells you very little of substance: you want to know how to *approach* this kind of problem, and ask "what's the *important* stuff here?" But if you can see the emerging picture, then understanding the design pressures and resolutions comes relatively painlessly.

Also, the *descriptions* that comment completed work are better guides for *new* work than prescriptive instructions. We've found that sequential descriptions, in an order reflecting the emergence of a coherent whole, are *more generative than instructions*. The

reasons for this have to do with our natural ability to make coherent things, but it's enough for now to say that descriptions are simply more evocative and less restrictive than prescriptions.

But even without comments and replaying of growth, the code resulting from this process will be more readable. This is because small, good steps more efficiently create better organization. If the programmer's goal is to make a satisfying improvement with little effort, he must build his program well. He'll try to do so with every step, and then be able to evaluate his success, gradually getting better at on-the-fly, stepwise organization.

The moment when one is trying to find that smallest solution to the maximum number of current problems, is the time one can use inspiration from *patterns*. Patterns (good solutions to problems) aren't anywhere near as effective in other circumstances. For example, they can seriously mislead a product if they are part of some giant architectural design. Patterns must be available *when the problem is at hand*, when the programmer is searching, trying to understand what the best next thing to do might be.

"Take the smallest-best step", incidentally, is also the best procedure for *discovering* good patterns. After making a "best move" one immediately comments on the *nature* of the move. This will be a simple enough observation that it can be re-used by the programmer, and others looking for real examples of excellent moves to use in their own situations. *Large* complex steps are simply *not* similarly re-usable, describable, or useful as patterns.

Good patterns, and good steps, are strongly and usefully *differentiating*: they make simple distinctions and connections in a system to make it adapt to pressures coherently. This is how living organisms both develop and evolve. For example, organisms are always coherent, stable and able to function, and yet they can adapt structurally. This is only possible through small, differentiating efforts. Making small changes in a program, then compiling, running and evaluating the results, is the same process.

In designing with this process, we are playing into our natural abilities to guide the emergence of good order. As a result, we are beginning to see sets of practical, good habits that should be incorporated into differentiating steps for program design.

***

This is just our preliminary understanding of this. What we would like to do is a full project using the

"smallest-best steps" method, and slowly build up tools and libraries that facilitate this approach. The project would be completely documented step-by-step, and the results would be published. The publication would also document the differentiating patterns discovered and the resulting interlocking descriptive sequences.

Certainly the subject of this experiment must be a genuine project. For reasons touched on above, *only* in a project *requiring* small evaluations of goodness can this process be understood. So we propose that this "smallest-best steps" group in CES be directly responsible to the production of the environmental design tool.

Reciprocally, the environmental tool will benefit *significantly* from a programming workbench that makes this kind of development easier.