Back to the future: Is worse (still) better?

Moderator: Martine Devos, EDS
martine.devos@eds.com

Positions: Richard P. Gabriel, Sun Microsystems and Stanford University
rpg@ rpg@Steam.Stanford.EDU

Panelmembers: James O. Coplien, T.J.D'Hondt, Jutta Eckstein, Brian Foote, Richard Gabriel, Kevlin Henney, Alan O'Callaghan

Abstract: Functional programming, AI, patterns, OO, structured programming - they were promising, and yet they seem to have failed to deliver. Did we lose interest too soon? Is the best too good for our industry? Is there "a" best for our industry or is our endless search for the silver bullet driving us? Do we want the "best" to (again) be a popular goal? How?

Back to the Future: Worse (Still) is Better! *

Spring-Watching Pavilion
A gentle spring evening arrives
airily, unclouded by worldly dust.
Three times the bell tolls echoes like a wave.
We see heaven upside down in sad puddles.
Love's vast sea cannot be emptied.
And springs of grace flow easily everywhere.
Where is nirvana?
Nirvana is here, nine times out of ten.
-from the Vietnamese of Ho Xuan Huong

Technology, art, popular media including network TV, and just about every aspect of our lives and probably life itself follows a disappointing pattern: Worse is better, the good drives out the excellent, and the most popular is least good.

You can look at it like this: To appeal to the majority of people, an artifact must appeal to something that those people have in common; as we increase the set of people we consider, the less those people have in common, and that which they do have in common becomes more base. For example, network TV is flooded with adolescent humor, sex, and violence because these are the most basic drives people have, while an interest in Vietnamese poetry - which is a very refined poetry - is quite rare. We call this the intersection effect.

The phenomenon of worse is better occurs even within a relatively esoteric sub-field like object- oriented programming and programming languages. C++ is still the language of choice though its kissing cousin, Java, is gaining popularity. CLOS (an advanced OO language with meta-objects and tremendous power), Smalltalk (perhaps the purest form of OO), Eiffel (well-thought-out and elegant), and Self (simplicity embodied) all sit on the sidelines while all the starters are C, C++, and Java.

We see the same thing in other languages. Common Lisp, which includes, CLOS, runs roughly the speed of Java or better, its runtime is smaller than Java by a lot, it has a programmatically portable executable format for code, and yet it is not only not popular, but it is not even taken seriously as a programming language. Lisp isn't even on the junior varsity. Never mind that Yahoo stores are written in Common Lisp and that NASA's Deep Space 1 space probe was written in Common Lisp.

This second example is illustrative of a devastating point: It is not simply that worse programming languages prevail through a reduction of quality via the

intersection effect, but the perpetuation of worse programming languages, once they become popular, is argued for and acted on stridently. We have seen the same thing happen to Prolog, Smalltalk, Self, ML, and Haskell.

More disappointing is witnessing this same effect at work in the patterns community. Christopher Alexander's ideas of patterns has as its smallest part the pattern form - the concept of patterns really has to do with pattern languages and QWAN (the Quality Without a Name). It is not about construction tricks. It is about building artifacts not only suitable for human habitation, but artifacts that increase their human inhabitants' feeling of life and wholeness. Alexander is all about beauty and quality, not about how to stick things together cleverly.

How could worse-is-better come to be? Are there good reasons - like it is better to release something initially that is not so good but on the right track and then let a community of inhabitants repair it using piecemeal growth? Or maybe it's that lower cost and otherwise less effective technologies eventually push out overpriced and over-engineered competitors? Or is it that quality is like Vietnamese poetry and thus rarely appreciated?

Back to the Future: Worse (Still) is Better!

Many people find the worse-is-better philosophy rather funny, more of a parody than anything else. It is a fascinating philosophy because it sounds silly while at the same time it works. The problem with The Right Thing - the anti-worse-is-better philosophy that most people tout as best - is that it can work only if luck is on your side, and then rarely even when it is.

Alexander explained it best when he talked about how living, whole, QWANful designs are ones that are developed over time by their inhabitants while paying attention to the smallest details. He talks about details down to 1/8" as being the scale you need to pay attention to in something the size of a house. In houses, to work at that level you need appropriate tools - tools that work at that scale comfortably. Modular parts don't work because they have details set in ways you cannot modify.

Let's look at languages: Many high-level programming languages hide detail by providing big abstractions. This means that you are working with modular parts and not with small pieces constructed by hand. For the right level of close detail you need a low-level language like C, assembler, or C++. With these languages you can control data layout and design very precise and exact algorithms. With Lisp and Smalltalk, for example, you are working with their big abstractions in a ham-fisted way - just fine for prototyping and understanding larger issues, but not good for minute algorithm and data structure design. For these we need craftsmen, not storyboard designers.

The only problem that the so-called high-level languages solve is being able to put large things together quickly - this produces mock ups. The real problem is thinking that you can build whole, alive software fast. Alexander never says great buildings are built quickly; in fact if anything he says the opposite. You need to build, inhabit, feel, then build some more - over time.

Habitable software means nice interfaces and all that, but it also means a reasonable size for the software and good performance. Size and performance mean a lot to usability, and for those qualities you need low-level languages. It does not hurt that programming in such languages is difficult, because there is merit in going slowly.

Come on. Alexander says, pay attention to the details and do things slowly according to feelings, not according to abstract aesthetics.

Martine Devos, *EDS EMEA*
martine.devos@eds.com

James O. Coplien, *Lucent Technologies*
cope@bell-labs.com

Theo D'Hondt, *Free University of Brussels*
tjdhondt@vub.ac.be

Jutta Eckstein, *Objects in Action*
jeckstein@acm.org

Brian Foote, *The Refactory, Inc.*
foote@laputan.org,

Richard Gabriel, *Sun Microsystems and Stanford University*
rpg@Steam.Stanford.EDU

Kevlin Henney, *Curbralan Limited*
kevlin@curbralan.com

Alan O'Callaghan, *De Montfort University*
Alan_OCallaghan@compuserve.com