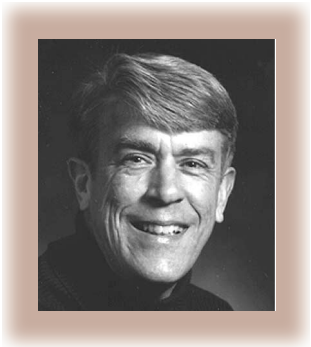
Editor: Robert L. Glass ■ Computing Trends ■ rglass@indiana.edu

Extreme Programming: The Good, the Bad, and the Bottom Line

Robert L. Glass



Ah, Extreme Programming!

It began as a kind of software counterculture collection of ideas. The subtitle of Kent Beck's innovative book *Extreme Programming Explained* is *Embrace Change*, and Erich Gamma's foreword to the book says "there is no doubt XP challenges some traditional big M [methodological] approaches."

But now, at least in some circles, XP is becoming the "in" way to develop software. It has evolved into a near religion, complete with advocates and even zealots, and with its own kind of (at least little) M approaches.

It makes you wonder what the Loyal Opposition side of this issue should be. Is it pro-XP, because Beck essentially saw it as a radical new approach? Or is it anti-XP, because its advocates are pushing it into (their view of) the software mainstream?

I'll do here what I think is the right thing to do with any methodology—disaggregate it and examine its constituent elements in isolation. (Undoubtedly, XP should be considered a methodology—Beck, in the second sentence of his book's preface, calls it a "lightweight methodology.") That approach also eases its way around the dilemma of what the proper Loyal Opposition position should be. There might or might not be an appropriate position on XP as a whole, but there certainly are appropriate positions to take on its elements.

On the basis on my reading of Beck's book, XP consists of

- pair programming (ongoing shared code reviews),
- unit testing (all the time),
- refactoring (ongoing design),
- relying on metaphors (with ongoing architectural refinement),
- continuous integration (perhaps even several times a day),
- short bursts of planning (measured in minutes and hours), and
- an emphasis on simplicity.

(How did these elements get chosen? Beck says he took all the best practices he knew and "turned the knob up to 10 to see what happened.")

I see good news and bad news in that collection. First, let's look at …

The bad news

Pair programming: It takes a certain kind of programmer to want to work in harness with another. Many of us prefer to work for bursts of creative time in isolation, getting back together with our team when we have pieces of information to share and decisions to make. I cannot imagine holding ongoing conversations with a pairmate when I am operating in creative mode. I cannot imagine that many programmers I know would want to operate that way, either.

Continued on p. 111

0740-7459/01/$10.00 © 2001 IEEE

Continued from p. 112

Refactoring: This practice is good, at least as Martin Fowler presents it in his groundbreaking book *Refactoring: Improving the Design of Existing Code*. But when refactoring comes to mean ongoing design at development time, as I interpret Beck to define it, something has gone wrong with the idea. Suddenly it has become an excuse for putting off design until the last minute, until coding time. That might be okay for tiny projects (Beck says XP is for small to medium-sized projects), but it's a disaster in the making for larger ones.

Short bursts of planning: This goes hand-in-hand with Beck's use of refactoring. Clearly, he views planning as something you can do on the fly. That, too, is a recipe for disaster on projects of any significance. Beck's software experience and mine must be very different. I, too, am troubled by the "analysis paralysis" that sometimes develops on software projects. But a certain amount of that analysis—planning—is necessary on any project I have worked on since computer memory grew beyond 4K words (allowing the tackling of truly significant real-world problems)!

On the other hand, there's …

The good news

Unit testing: Few, other than formal-method and fault tolerance zealots, will argue against unit testing. It might well be the most agreed-upon software best practice of all time. Unit testing "all the time" might be a bit extreme, but this might well be one of those "if a little is good, then a lot is better" ideas.

Relying on metaphors: Once again, it's hard to argue with this concept. We humans tend to think best when we can relate a problem at hand to one we've solved in the past. Beck ties the idea into "ongoing architectural refinement." Architecture and refinement are good. The only troubling aspect of this element is the notion of making that an ongoing activity. If that means encouraging architectural improvement as a project evolves, that's a good thing. To the extent it means deferring architecture in the same ways that we are to defer design and planning, it's bad.

Continuous integration: I see this element as being based on Microsoft's (and others') "daily build." Although traditional software engineering has not completely embraced this idea, continuous integration is certainly an excellent way to ward off "big bang" catastrophes.

An emphasis on simplicity: Everyone supports this. Who was it who said a problem's solution should be as simple as possible, but never simpler? Was it Occam's Razor that implied the simplest solution was the best? Too much software means fatware; too much software means clumsy design and coding; too much software means a tangled web.

More good news

With all those goods and bads on the table, I have some other things I want to say about Extreme Programming. It is extremely human-focused, in ways the rest of the software field would do well to emulate.

It is programmer focused:

- The 40-hour workweek is an expected practice, and overtime is discouraged as being counterproductive.
- The programmer may choose one of the prime factors of project planning—cost, time, quality, and scope—after the customer has chosen the other three. (This is by far my favorite part of Extreme Programming!)

It is customer focused:

- The customer is expected to be on the software team.
- Listening to the customer is one of the "four basic activities" (the others are coding, testing, and designing).
- The customer gets to pick three of the four prime factors.

So what's my bottom line? Extreme Programming is a fascinating collection of elements, some good and some bad. Many elements are inappropriate for all but the tiniest projects, so there is little excuse for seeing this as a mainstream methodology (zealots on almost any subject are usually wrong, in my view). But some elements could scale up in truly important ways.

I'm curious about your bottom line on the matter.

Robert L. Glass is the editor of Elsevier's *Journal of Systems and Software* and the publisher and editor of the *Software Practitioner* newsletter. Contact him at rglass@indiana.edu; he'd be pleased to hear from you.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Subscription rates: IEEE Computer Society members get the lowest rates and choice of media option—$42/34/55 US print/electronic/combination; go to http://computer.org/subscribe to order and for more information on other subscription prices. Back issues: $10 for members, $20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.