CUTTER CONSORTIUM

EXTREME PROGRAMMING

AGILE PROJECT MANAGEMENT ADVISORY SERVICE WHITE PAPER

by Jim Highsmith

CUTTER CONSORTIUM, 37 BROADWAY, SUITE 1, ARLINGTON, MA 02474-5552, USA;
Tel: +1 781 648 8700; Fax: +1 781 648 8707;
consortium@cutter.com; www.cutter.com

Extreme Programming

[This article orginally appeared in the February 2000 edition of Cutter Consortium's *e-Business Application Delivery* newsletter, now titled the Agile Project Management Advisory Service.]

by Jim Highsmith

As we have explored in several issues of ***eAD***, the two most pressing issues in information technology today are:

- How do we deliver functionality to business clients quickly?
- How do we keep up with near-continuous change?

Change is changing. Not only does the pace of change continue to accelerate, but, as the September issue of ***eAD*** pointed out, organizations are having to deal with different types of change — disruptive change and punctuated equilibrium. Disruptive technologies, like personal computers in the early 1980s, impact an industry (in the case of PCs, several related industries), while a punctuated equilibrium — a massive intervention into an ecosystem or an economy — impacts a very large number of species, or companies. The Internet, which has become the backbone for e-commerce and e-business, has disrupted a wide range of industries — more a punctuated equilibrium than a disruption.

When whole business models are changing, when time-to-market becomes the mantra of companies, when flexibility and interconnectedness are demanded from even the most staid organization, it is then that we must examine every aspect of how business is managed, customers are delighted, and products are developed.

The Extreme Programming movement has been a subset of the object-oriented (OO) programming community for several years, but has recently attracted more attention, especially with the recent release of Kent Beck's new book *Extreme Programming Explained: Embrace Change*. Don't be put off by the somewhat "in-your-face" moniker of Extreme Programming (XP to practitioners). Although Beck doesn't claim that practices such as pair programming and incremental planning originated with XP, there are some very interesting, and I think important, concepts articulated by XP. There's a lot of talk today about change, but XP has some pretty good ideas about how to actually do it. Hence the subtitle, Embrace Change.

There is a tendency, particularly by rigorous methodologists, to dismiss anything less ponderous than the Capability Maturity Model (CMM) or maybe the International Organization for Standardization's standards, as hacking. The connotation: hacking promotes doing rather than thinking and therefore results in low quality. This is an easy way to dismiss practices that conflict with one's own assumptions about the world.

Looked at another way, XP may be a *potential* piece of a puzzle I've been writing about over the past 18 months. Turbulent times give rise to new problems that, in turn, give rise to new practices — new practices that often fly in the face of conventional wisdom but survive because they are better adapted to the new reality. There are at least four practices I would assign to this category:

- XP — the focus of this issue
- Lean development — discussed in the November 1998 issue of ***eAD***
- Crystal Light methods — mentioned in the November 1999 issue of ***eAD*** and further discussed in this issue
- Adaptive software development — described in the August 1998 issue of ***eAD*** (then called ***Application Development Strategies — ADS***)

Although there are differences in

each of these practices, there are also similarities: they each describe variations from the conventional wisdom about how to approach software development. Whereas lean and adaptive development practices target strategic and project management, XP brings its differing world view to the realm of the developer and tester.

Much of XP is derived from good practices that have been around for a long time. “None of the ideas in XP are new. Most are as old as programming,” Beck offers to readers in the preface to his book. I might differ with Beck in one respect: although the practices XP uses aren’t new, the conceptual foundation and how they are melded together greatly enhance these “older” practices. I think there are four critical ideas to take away from XP (in addition to a number of other good ideas):

- The cost of change
- Refactoring
- Collaboration
- Simplicity

But first, I discuss some XP basics: the dozen practices that define XP.

XP — The Basics

I must admit that one thing I like about XP’s principal figures is their lack of pretension. XP proponents are careful to articulate where they think XP is appropriate and where it is not. While practitioners like Beck and Ron Jeffries may envision that XP has wider applicability, they are generally circumspect about their claims. For example, both are clear about XP’s applicability to small (less than 10 people), co-located teams (with which they have direct experience); they don’t try to convince people that the practices will work for teams of 200.

The Project

The most prominent XP project reported on to date is the Chrysler Comprehensive Compensation system (the C3 project) that was initiated in the mid-1990s and converted to an XP project in 1997. Jeffries, one of the “Three Extremoes” (with Beck and Ward Cunningham), and I spent several hours talking about the C3 project and other XP issues at the recent Miller Freeman *Software Developer* conference in Washington, DC, USA.

Originally, the C3 project was conceived as an OO programming project, specifically using Smalltalk. Beck, a well-known Smalltalk expert, was called in to consult on Smalltalk performance optimization, and the project was transformed into a pilot of OO (XP) practices after the original project was deemed unreclaimable. Beck brought in Jeffries to assist on a more full-time basis, and Jeffries worked with the C3 team until spring 1999. The initial requirements were to handle the monthly payroll of some 10,000 salaried employees. The system consists of approximately 2,000 classes and 30,000 methods and was ready within a reasonable tolerance period of the planned schedule.

As we talked, I asked Jeffries how success on the C3 project translated into XP use on other Chrysler IT projects. His grin told me all I needed to know. I’ve been involved in enough rapid application development (RAD) projects for large IT organizations over the years to understand why success does not consistently translate into acceptance. There are always at least a hundred very good reasons why success at RAD, or XP, or lean development, or other out-of-the-box approaches doesn’t translate into wider use — but more on this issue later.

Practices

One thing to keep in mind is that XP practices are intended for use with small, co-located teams. They therefore tend toward minimalism, at least as far as artifacts other than code and test cases are concerned. The presentation of XP’s practices have both positive and negative aspects. At one level, they sound like rules — do this, don’t do that. Beck explains that the practices are more like guidelines than rules, guidelines that are pliable depending on the situation. However, some, like the “40-hour week,” can come off as a little preachy. Jeffries makes the point that the practices also interact, counterbalance, and reinforce each other, such that picking and choosing which to use and which to discard can be tricky.

The planning game. XP’s planning approach mirrors that of most iterative RAD approaches to projects. Short, three-week cycles, frequent updates, splitting business and technical priorities, and assigning “stories” (a story defines a particular feature requirement and is displayed in a simple card format) all define XP’s approach to planning.

Small releases. "Every release should be as small as possible, containing the most valuable business requirements," states Beck. This mirrors two of Tom Gilb's principles of evolutionary delivery from his book *Principles of Software Engineering Management*: "All large projects are capable of being divided into many useful partial result steps," and "Evolutionary steps should be delivered on the principle of the juiciest one next."

Small releases provide the sense of accomplishment that is often missing in long projects as well as more frequent (and more relevant) feedback. However, a development team needs to also consider the difference between "release" and "releasable." The cost of each release — installation, training, conversions — needs to be factored into whether or not the product produced at the end of a cycle is actually released to the end user or is simply declared to be in a releasable state.

Metaphor. XP's use of the terms "metaphor" and "story" take a little wearing in to become comfortable. However, both terms help make the technology more understandable in human terms, especially to clients. At one level, metaphor and architecture are synonyms — they are both intended to provide a broad view of the project's goal. But architectures often get bogged down in symbols and connections. XP uses "metaphor" in an attempt to define an overall coherent theme to which both developers and business clients can relate. The metaphor describes the broad sweep of the project, while stories are used to describe individual features.

Simple design. Simple design has two parts. One, design for the functionality that has been defined, not for potential future functionality. Two, create the best design that can deliver that functionality. In other words, don't guess about the future: create the best (simple) design you can today. "If you believe that the future is uncertain, and you believe that you can cheaply change your mind, then putting in functionality on speculation is crazy," writes Beck. "Put in what you need when you need it."

In the early 1980s, I published an article in *Datamation* magazine titled "Synchronizing Data with Reality." The gist of the article was that data quality is a function of use, not capture and storage. Furthermore, I said that data that was not systematically used would rapidly go bad. Data quality is a function of systematic usage, not anticipatory design. Trying to anticipate what data we will need in the future only leads us to design for data that we will probably never use; even the data we did guess correctly on won't be correct anyway. XP's simple design approach mirrors the same concepts. As described later in this article, this doesn't mean that no anticipatory design ever happens; it does mean that the economics of anticipatory design changes dramatically.

Refactoring. If I had to pick one thing that sets XP apart from other approaches, it would be refactoring — the ongoing redesign of software to improve its responsiveness to change. RAD approaches have often been associated with little or no design; XP should be thought of as continuous design. In times of rapid, constant change, much more attention needs to be focused on refactoring. See the sections "Refactoring" and "Data Refactoring," below.

Testing. XP is full of interesting twists that encourage one to think — for example, how about "Test and then code"? I've worked with software companies and a few IT organizations in which programmer performance was measured on lines of code delivered and testing was measured on defects found — neither side was motivated to reduce the number of defects prior to testing. XP uses two types of testing: unit and functional. However, the practice for unit testing involves developing the test for the feature prior to writing the code and further states that the tests should be automated. Once the code is written, it is immediately subjected to the test suite — instant feedback.

The most active discussion group on XP remains the Wiki exchange (XP is a piece of the overall discussion about patterns). One of the discussions centers around a lifecycle of Listen (requirements) → Test → Code → Design. Listen closely to customers while gathering their requirements. Develop test cases. Code the objects (using pair programming). Design (or refactor) as more objects are added to the system. This seemingly convoluted lifecycle begins to make sense only in an environment in which change dominates.

Pair programming. One of the few software engineering practices that enjoys near-universal acceptance (at least in theory) and has been well measured is software inspections (also referred to as reviews or walkthroughs). At their best, inspections are collaborative interactions that speed learning as much as they uncover defects. One of the lesser-known statistics about inspections is that although they are very cost effective in uncovering defects, they are even more effective at preventing defects in the first place through the team's ongoing learning and incorporation of better programming practices.

One software company client I worked with cited an internal study that showed that the amount of time to isolate defects was 15 hours per defect with testing, 2-3 hours per defect using inspections, and 15 minutes per defect by finding the defect before it got to the inspection. The latter figure arises from the ongoing team learning engendered by regular inspections. Pair programming takes this to the next step — rather than the incremental learning using inspections, why not continuous learning using pair programming?

"Pair programming is a dialog between two people trying to simultaneously program and understand how to program better," writes Beck. Having two people sitting in front of the same terminal (one entering code or test cases, one reviewing and thinking) creates a continuous, dynamic interchange. Research conducted by Laurie Williams for her doctoral dissertation at the University of Utah confirm that pair programming's benefits aren't just wishful thinking (see Resources and References).

Collective ownership. XP defines collective ownership as the practice that anyone on the project team can change any of the code at any time. For many programmers, and certainly for many managers, the prospect of communal code raises concerns, ranging from "I don't want those bozos changing my code" to "Who do I blame when problems arise?" Collective ownership provides another level to the collaboration begun by pair programming.

Pair programming encourages two people to work closely together: each drives the other a little harder to excel. Collective ownership encourages the entire team to work more closely together: each individual and each pair strives a little harder to produce high-quality designs, code, and test cases. Granted, all this forced "togetherness" may not work for every project team.

Continuous integration. Daily builds have become the norm in many software companies — mimicking the published material on the "Microsoft" process (see, for example, Michael A. Cusumano and Richard Selby's *Microsoft Secrets)*. Whereas many companies set daily builds as a minimum, XP practitioners set the daily integration as the maximum — opting for frequent builds every couple of hours. XP's feedback cycles are quick: develop the test case, code, integrate (build), and test.

The perils of integration defects have been understood for many years, but we haven't always had the tools and practices to put that knowledge to good use. XP not only reminds us of the potential for serious integration errors, but provides a revised perspective on practices and tools.

40-hour week. Some of XP's 12 practices are principles, while others, such as the 40-hour practice, sound more like rules. I agree with XP's sentiments here; I just don't think work hours define the issue. I would prefer a statement like, "Don't burn out the troops," rather than a 40-hour rule. There are situations in which working 40 hours is pure drudgery and others in which the team has to be pried away from a 60-hour work week.

Jeffries provided additional thoughts on overtime. "What we say is that overtime is defined as time in the office when you don't want to be there. And that you should work no more than one week of overtime. If you go beyond that, there's something wrong — and you're tiring out and probably doing worse than if you were on a normal schedule. I agree with you on the sentiment about the 60-hour work week. When we were young and eager, they were probably okay. It's the dragging weeks to watch for."

I don't think the number of hours makes much difference. What defines the difference is volunteered commitment. Do people want to come to work? Do they anticipate each day with great relish? People have to come to work, but they perform great feats by being committed to the project, and commitment only arises from a sense of purpose.

On-site customer. This practice corresponds to one of the oldest cries in software development — user involvement. XP, as with every other rapid development approach, calls for ongoing, on-site user involvement with the project team.

Coding standards. XP practices are supportive of each other. For example, if you do pair programming and let anyone modify the communal code, then coding standards would seem to be a necessity.

Values and Principles

On Saturday, 1 January 2000, the *Wall Street Journal* (you know, the "Monday through Friday" newspaper) published a special 58-page millennial edition. The introduction to the Industry & Economics section, titled "So Long Supply and Demand: There's a new economy out there — and it looks nothing like the old one," was written by Tom Petzinger. "The bottom line: creativity is overtaking capital as the principal elixir of growth," Petzinger states.

Petzinger isn't talking about a handful of creative geniuses, but the creativity of groups — from teams to departments to companies. Once we leave the realm of the single creative genius, creativity becomes a function of the environment and how people interact and collaborate to produce results. If your company's fundamental principles point to software development as a statistically repeatable, rigorous, engineering process, then XP is probably not for you. Although XP contains certain rigorous practices, its intent is to foster creativity and communication.

Environments are driven by values and principles. XP (or the other practices mentioned in this issue) may or may not work in your organization, but, ultimately, success won't depend on using 40-hour work weeks or pair programming — it will depend on whether or not the values and principles of XP align with those of your organization.

Beck identifies four values, five fundamental principles, and ten secondary principles — but I'll mention five that should provide enough background.

Communication. So, what's new here? It depends on your perspective. XP focuses on building a person-to-person, mutual understanding of the problem environment through minimal formal documentation and maximum face-to-face interaction. "Problems with projects can invariably be traced back to somebody not talking to somebody else about something important," Beck says. XP's practices are designed to encourage interaction — developer to developer, developer to customer.

Simplicity. XP asks of each team member, "What is the simplest thing that could possibly work?" Make it simple today, and create an environment in which the cost of change tomorrow is low.

Feedback. "Optimism is an occupational hazard of programming," says Beck. "Feedback is the treatment." Whether it's hourly builds or frequent functionality testing with customers, XP embraces change by constant feedback. Although every approach to software development advocates feedback — even the much-maligned waterfall model — the difference is that XP practitioners understand that *feedback* is more important than *feedforward*. Whether it's fixing an object that failed a test case or refactoring a design that is resisting a change, high-change environments require a much different understanding of feedback.

Courage. Whether it's a CMM practice or an XP practice that defines your discipline, discipline requires courage. Many define courage as doing what's right, even when pressured to do something else. Developers often cite the pressure to ship a buggy product and the courage to resist. However, the deeper issues can involve legitimate differences of opinion over what is right. Often, people don't lack courage — they lack conviction, which puts us right back to other values. If a team's values aren't aligned, the team won't be convinced that some practice is "right," and, without conviction, courage doesn't seem so important. It's hard to work up the energy to fight for something you don't believe in.

"Courage isn't just about having the discipline," says Jeffries. "It is also a resultant value. If you do the practices that are based on communication, simplicity, and feedback, you are given courage, the confidence to go ahead in a lightweight manner," as opposed to being weighted down by the more cumbersome, design-heavy practices.

Quality work. Okay, all of you out there, please raise your hand if you advocate poor-quality work. Whether you are a proponent of the Rational Unified Process, CMM, or XP, the real issues are "How do you define quality?" and "What actions do you think deliver high quality?" Defining quality as "no defects" provides one perspective on the question; Jerry Weinberg's definition, "Quality is value to some person," provides another. I get weary of methodologists who use the "hacker" label to ward off the intrusion of approaches like XP and lean development. It seems unproductive to return the favor. Let's concede that all these approaches are based on the fundamental principle that individuals want to do a good, high-quality job; what "quality" means and how to achieve it — now there's the gist of the real debate!

Managing XP

One area in which XP (at least as articulated in Beck's book) falls short is management, understandable for a practice oriented toward both small project teams and programming. As Beck puts it, "Perhaps the most important job for the coach is the acquisition of toys and food." (Coaching is one of Beck's components of management strategy.)

With many programmers, their recommended management strategy seems to be: get out of the way. The underlying assumption? Getting out of the way will create a collaborative environment. Steeped in the tradition of task-based project management, this assumption seems valid. However, in my experience, creating and maintaining highly functional collaborative environments challenges management far beyond making up task lists and checking off their completion.

The Cost of Change

Early on in Beck's book, he challenges one of the oldest assumptions in software engineering. From the mid-1970s, structured methods and then more comprehensive methodologies were sold based on the "facts" shown in Figure 1. I should know; I developed, taught, sold, and installed several of these methodologies during the 1980s.

Beck asks us to consider that perhaps the economics of Figure 1, probably valid in the 1970s and 1980s, now look like Figure 2 — that is, the cost of maintenance, or ongoing change, flattens out rather than escalates. Actually, whether Figure 2 shows today's cost profile or not is irrelevant — we have to make it true! If Figure 1 remains true, then we are doomed because of today's pace of change.

The vertical axis in Figure 1 usually depicts the cost of finding defects late in the development cycle. However, this assumes that all changes are the results of a mistake — i.e., a defect. Viewed from this perspective, traditional methods have concentrated on "defect prevention" in early lifecycle stages. But in today's environment, we can't prevent what we don't know about — changes arise from iteratively gaining knowledge about the application, not from a defective process. So, although our practices need to be geared toward preventing some defects, they must also be geared toward reducing the cost of continuous change. Actually, as Alistair Cockburn points out, the high cost of removing defects shown by Figure 1 provides an economic justification for practices like pair programming.

In this issue, I want to restrict the discussion to change at the project or application level — decisions about operating systems, development language, database, middleware, etc., are constraints outside the control of the development team. (For ideas on "architectural" flexibility, see the June and July 1999 issues of ***ADS***.) Let's simplify even further and assume, for now, that the business and operational requirements are known.

Our design goal is to balance the rapid delivery of functionality while we also create a design that can be easily modified. Even within the goal of rapid delivery, there remains another balance: proceed too hurriedly and bugs creep in; try to anticipate every eventuality and time flies. However, let's again simplify our problem and assume we have reached a reasonable balance of design versus code and test time.

With all these simplifications, we are left with one question: how much anticipatory design work do we do? Current design produces the functionality we have already specified. Anticipatory design builds in extra facilities with the anticipation that future requirements will be faster to implement. Anticipatory design trades current time for future time, under the assumption that a little time now will save more time later. But under what conditions is that assumption true? Might it not be faster to redesign later, when we know exactly what the changes are, rather than guessing now?

Figure 1 — Historical lifecycle change costs

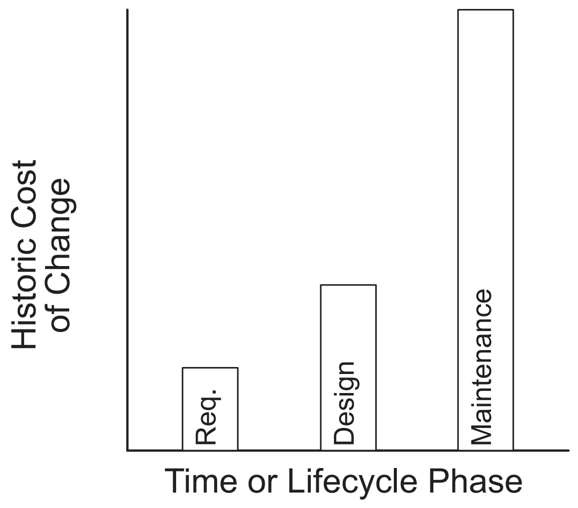
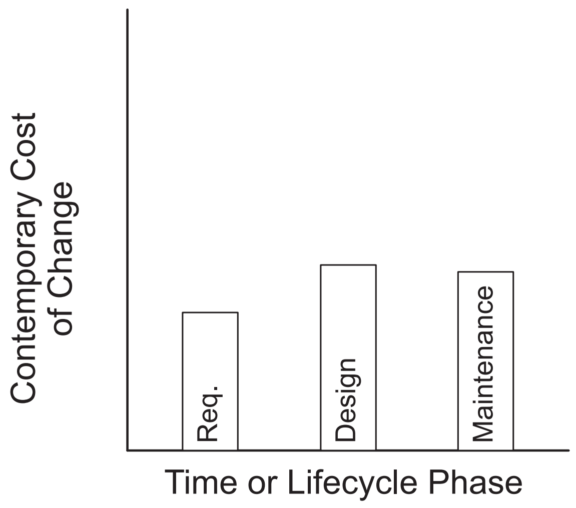


Figure 2 — Contemporary lifecycle change costs.



(Source: Adapted from Beck)

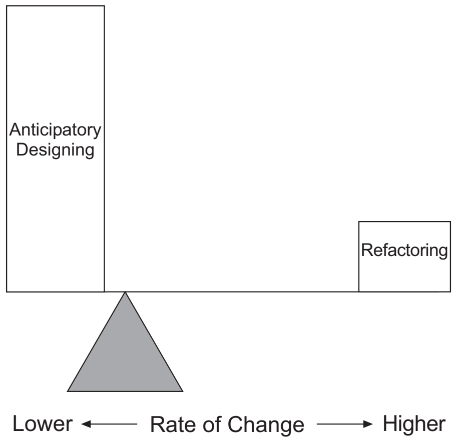


Figure 3 — Balancing design and refactoring, pre-internet.

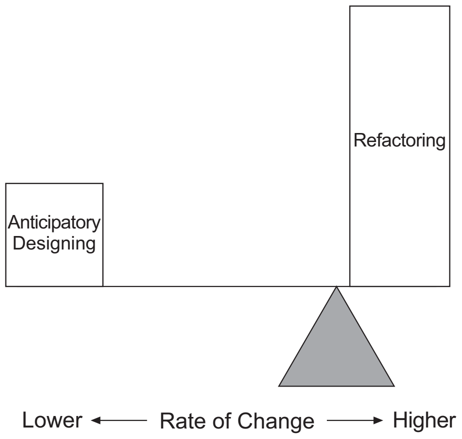


Figure 4 — Balancing design and refactoring today.

This is where refactoring enters the equation. Refactoring, according to author Martin Fowler, is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." XP proponents practice continuous, incremental refactoring as a way to incorporate change. If changes are continuous, then we'll never get an up-front design completed. Furthermore, as changes become more unpredictable — a great likelihood today — then much anticipatory design likely will be wasted.

I think the diagram in Figure 3 depicts the situation prior to the rapid-paced change of the Internet era. Since the rate of change (illustrated by the positioning of the balance point in the figure) was lower, more anticipatory designing versus refactoring may have been reasonable. As Figure 4 shows, however, as the rate of change increases, the viability of anticipatory design loses out to refactoring — a situation I think defines many systems today.

In the long run, the only way to test whether a design is flexible involves making changes and measuring how easy they are to implement. One of the biggest problems with the traditional up-front-design-then-maintain strategy has been that software systems exhibit tremendous entropy; they degrade over time as maintainers rush fixes, patches, and enhancements into production. The problem is worse today because of the accelerated pace of change, but current refactoring approaches aren't the first to address the problem. Back in the "dark ages" (circa 1986), Dave Higgins wrote *Data Structured Software Maintenance*, a book that addressed the high cost of maintenance, due in large part to the cumulative effects of changes to systems over time. Although Higgins advocated a particular program-design approach (the Warnier/Orr Approach), one of his primary themes was to stop the degradation of systems over time by systematically redesigning programs during maintenance activities.

Higgins's approach to program maintenance was first to develop a pattern (although the term pattern was not used then) for how the program "should be" designed, then to create a map from the "good" pattern to the "spaghetti" code. Programmers would then use the map to help understand the program and, further, to revise the program over time to look more like the pattern. Using Higgins's approach, program maintenance counteracted the natural tendency of applications to degrade over time. "The objective was not to rewrite the entire application," said Higgins in a recent conversation, "but to rewrite those portions for which enhancements had been requested."

Although this older-style "refactoring" was not widely practiced, the ideas are the same as they are today — the need today is just greater. Two things enable, or drive, increased levels of refactoring: one is better languages and tools, and the other is rapid change.

Another approach to high change arose in the early days of RAD: the idea of throwaway code. The idea was that things were changing so rapidly that we could just code applications very quickly, then throw them away and start over when the time for change arose. This turned out to be a poor long-term strategy.

Refactoring

Refactoring is closely related to factoring, or what is now referred to as using design patterns. Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, provides the foundational work on design patterns. Design Patterns serves modern-day OO programmers much as Larry Constantine and Ed Yourdon's Structural Design served a previous generation; it provides guidelines for program structures that are more effective than other program structures.

If Figure 4 shows the correct balance of designing versus refactoring for environments experiencing high rates of change, then the quality of initial design remains extremely important. Design patterns provide the means for improving the quality of initial designs by offering models that have proven effective in the past.

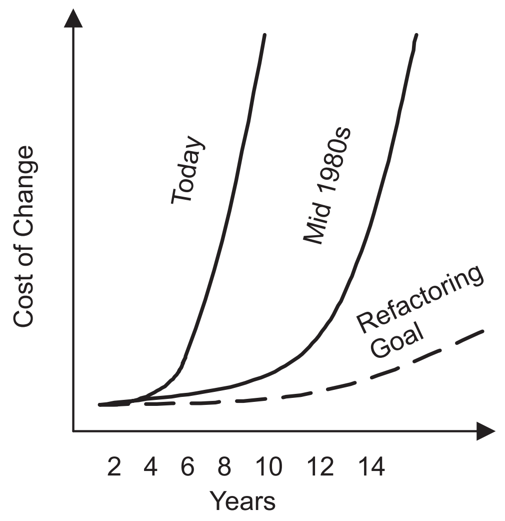
So, you might ask, why a separate refactoring book? Can't we just use the design patterns in redesign? Yes and no. As all developers (and their managers) understand, messing with existing code can be a ticklish proposition. The cliché "if it ain't broke, don't fix it" lives on in annals of development folklore. However, as Fowler comments, "The program may not be broken, but it does hurt." Fear of breaking some part of the code base that's "working" actually hastens the degradation of that code base. However, Fowler is well aware of the concern: "Before I do the refactoring, I need to figure out how to do it safely.… I've written down the safe steps in the catalog." Fowler's book, *Refactoring: Improving the Design of Existing Code*, catalogs not only the before (poor code) and after (better code based on patterns), but also the steps required to migrate from one to the other. These migration steps reduce the chances of introducing defects during the refactoring effort.

Beck describes his "two-hat" approach to refactoring — namely that adding new functionality and refactoring are two different activities. Refactoring, per se, doesn't change the observable behavior of the software; it enhances the internal structure. When new functionality needs to be added, the first step is often to refactor in order to simplify the addition of new functionality. This new functionality that is proposed, in fact, should provide the impetus to refactor.

Refactoring might be thought of as incremental, as opposed to monumental, redesign. "Without refactoring, the design of the program will decay," Fowler writes. "Loss of structure has a cumulative effect." Historically, our approach to maintenance has been "quick and dirty," so even in those cases where good initial design work was done, it degraded over time.

Figure 5 shows the impact of neglected refactoring — at some point, the cost of enhancements becomes prohibitive because the software is so shaky. At this point, monumental redesign (or replacement) becomes the only option, and these are usually high-risk, or at least high-cost, projects. Figure 5 also shows that while in the 1980s software decay might have taken a decade, the rate of change today hastens the decay. For example, many client-server applications hurriedly built in the early 1990s are now more costly to maintain than mainframe legacy applications built in the 1980s.

Figure 5 — Software entropy over time.



Data Refactoring: Comments by Ken Orr

Editor's Note: As I mentioned above, one thing I like about XP and refactoring proponents is that they are clear about the boundary conditions for which they consider their ideas applicable. For example, Fowler has an entire chapter titled "Problems with Refactoring." Database refactoring tops Fowler's list. Fowler's target, as stated in the subtitle to his book, is to improve code. So, for data, I turn to someone who has been thinking about data refactoring for a long time (although not using that specific term). The following section on data refactoring was written by Ken Orr.

When Jim asked me to put together something on refactoring, I had to ask him what that really meant. It seemed to me to come down to a couple of very simple ideas:

1. Do what you know how to do.
2. Do it quickly.
3. When changes occur, go back and redesign them in.
4. Go to 1.

Over the years, Jim and I have worked together on a variety of systems methodologies, all of which were consistent with the refactoring philosophy. Back in the 1970s, we created a methodology built on data structures. The idea was that if you knew what people wanted, you could work backward and design a database that would give you just the data that you needed, and from there you could determine just what inputs you needed to update the database so that you could produce the output required.

Creating systems by working backward from outputs to database to inputs proved to be a very effective and efficient means of developing systems. This methodology was developed at about the same time that relational databases were coming into vogue, and we could show that our approach would always create a well-behaved, normalized database. More than that, however, was the idea that approaching systems this way created minimal systems. In fact, one of our customers actually used this methodology to rebuild a system that was already in place. The customer started with the outputs and worked backward to design a minimal database with minimal input requirements.

The new system had only about one-third the data elements of the system it was replacing. This was a major breakthrough. These developers came to understand that creating minimal systems had enormous advantages: they were much smaller and therefore much faster to implement, and they were also easier to understand and change, since everything had a purpose.

Still, building minimal systems goes against the grain of many analysts and programmers, who pride themselves on thinking ahead and anticipating future needs, no matter how remote. I think this attitude stems from the difficulty that programmers have had with maintenance. Maintaining large systems has been so difficult and fraught with problems that many analysts and programmers would rather spend enormous effort at the front end of the systems development cycle, so they don't have to maintain the system ever again. But as history shows, this approach of guessing about the future never works out. No matter how clever we are in thinking ahead, some new, unanticipated requirement comes up to bite us. (How many people included Internet-based e-business as one of their top requirements in systems they were building 10 years ago?)

Ultimately, one of the reasons that maintenance is so difficult revolves around the problem of changing the database design. In most developers' eyes, once you design a database and start to program against it, it is almost impossible to change that database design. In a way, the database design is something like the foundation of the system: once you have poured concrete for the foundation, there is almost no way you can go back and change it. As it turns out, major changes to databases in large systems happen very infrequently, only when they are unavoidable. People simply do not think about redesigning a database as a normal part of systems maintenance, and, as a consequence, major changes are often unbelievably difficult.

Enter Data Refactoring

Jim and I had never been persuaded by the argument that the database design could never be changed once installed. We had the idea that if you wanted to have a minimal system, then it was necessary to take changes or new requirements to the system and repeat the basic system cycle over again, reintegrating these new requirements with the original requirements to create a new system. You could say that what we were doing was data refactoring, although we never called it that.

The advantages of this approach turned out to be significant. For one thing, there was no major difference between development of a new system and the maintenance or major modification of an existing one. This meant that training and project management could be simplified considerably. It also meant that our systems tended not to degrade over time, since we "built in" changes rather than "adding them on" to the existing system.

Over a period of years, we built a methodology (Data Structured Systems Development or Warnier/Orr) and trained thousands of systems analysts and programmers. The process that we developed was largely manual, although we thought that if we built a detailed-enough methodology, it should be possible to automate large pieces of that methodology in CASE tools.

Automating Data Refactoring

To make the story short, a group of systems developers in South America finally accomplished the automation of our data refactoring approach in the late 1980s. A company led by Breogán Gonda and Nicolás Jodal created a tool called

GeneXus[1] that accomplished what we had conceived in the 1970s. They created an approach in which you could enter data structures for input screens; with those data structures, GeneXus automatically designed a normalized database and generated the code to navigate, update, and report against that database.

But that was the easy part. They designed their tool in such a way that when requirements changed or users came up with something new or different, they could restate their requirements, rerun (recompile), and GeneXus would redesign the database, convert the previous database automatically to the new design, and then regenerate just those programs that were affected by the changes in the database design. They created a closed-loop refactoring cycle based on data requirements.

GeneXus showed us what was really possible using a refactoring framework. For the first time in my experience, developers were freed from having to worry about future requirements. It allowed them to define just what they knew and then rapidly build a system that did just what they had defined. Then, when (not if) the requirements changed, they could simply reenter those changes, recompile the system, and they had a new, completely integrated, minimal system that incorporated the new requirements.

What Does All This Mean?

Refactoring is becoming something of a buzzword. And like all buzzwords, there is some good news and some bad news. The good news is that, when implemented correctly, refactoring makes it possible for us to build very robust systems very rapidly. The bad news is that we have to rethink how we go about developing systems. Many of our most cherished project management and development strategies need to be rethought. We have to become very conscious of interactive, incremental design. We have to be much more willing to prototype our way to success and to use tools that will do complex parts of the systems development process (database design and code generation) for us.

In the 1980s, CASE was a technology that was somehow going to revolutionize programming. In the 1990s, objects and OO development were going to do the same. Neither of these technologies lived up to their early expectations. But today, tools like GeneXus really do many of the things that the system gurus of the 1980s anticipated. It is possible, currently, to take a set of requirements, automatically design a database from those requirements, generate an operational database from among the number of commercially available relational databases (Oracle, DB2, Informix, MS SQL Server, and Access), and generate code (prototype and production) that will navigate, update, and report against those databases in a variety of different languages (COBOL, RPG, C, C++, and Java). Moreover, it will do this at very high speed.

This new approach to systems development allows us to spend much more time with users, exploring their requirements and giving them user interface choices that were never possible when we were building things at arm's length. But not everybody appreciates this new world. For one thing, it takes a great deal of the mystery out of the process. For another, it puts much more stress on rapid development.

When people tell you that building simple, minimal systems is out of date in this Internet age, tell them that the Internet is all about speed and service. Tell them that refactoring is not just the best way to build the kind of systems that we need for the 21st century, it is the *only* way.

CRYSTAL LIGHT METHODS: COMMENTS BY ALISTAIR COCKBURN

Editor's note: In the early 1990s, Alistair Cockburn was hired by the IBM Consulting Group to construct and document a methodology for OO development. IBM had no preferences as to what the answer might look like, just that it work. Cockburn's approach to the assignment was to interview as many project team members as possible, writing down whatever the teams said was important to their success (or failure). The results were surprising. The remainder of this section was written by Cockburn and is based on his "in-process" book on minimal methodologies.

In the IBM study, team after successful team "apologized" for not following a formal process, for not using a high-tech CASE tool, for "merely" sitting close to each other and discussing as they went. Meanwhile, a number of failing teams puzzled over why they failed despite using a formal process — maybe they hadn't followed it well enough? I finally started encountering teams who asserted that they succeeded exactly because they did not get caught up in fancy processes and deliverables, but instead sat close together so they could talk easily and delivered tested software frequently.

[1]Gonda and Jodal created a company called ARTech to market the GeneXus product. It currently has more than 3,000 customers worldwide and is marketed in the US by GeneXus, Inc.

These results have been consistent, from 1991 to 1999, from Hong Kong to the Americas, Norway, and South Africa, in COBOL, Smalltalk, Java, Visual Basic, Sapiens, and Synon. The shortest statement of the results are:

> To the extent you can replace written documentation with face-to-face interactions, you can reduce reliance on written work products and improve the likelihood of delivering the system.
>
> The more frequently you can deliver running, tested slices of the system, the more you can reduce reliance on written "promissory" notes and improve the likelihood of delivering the system.

People are communicating beings. Even introverted programmers do better with informal, face-to-face communication than with paper documents. From a cost and time perspective, writing takes longer and is less communicative than discussing at the whiteboard.

Written, reviewed requirements and design documents are "promises" for what will be built, serving as timed progress markers. There are times when creating them is good. However, a more accurate timed progress marker is running tested code. It is more accurate because it is not a timed promise, it is a timed accomplishment.

Recently, a bank's IT group decided to take the above results at face value. They began a small project by *simply putting three people into the same room* and more or less leaving them alone. Surprisingly (to them), the team delivered the system in a fine, timely manner. The bank management team was a bit bemused. Surely it can't be this simple?

It isn't quite so simple. Another result of all those project interviews was that: different projects have different needs. Terribly obvious, except (somehow) to methodologists. Sure, if your project only needs 3 to 6 people, just put them into a room together. But if you have 45 or 100 people, that won't work. If you have to pass Food & Drug Administration process scrutiny, you can't get away with this. If you are going to shoot me to Mars in a rocket, I'll ask you not to try it. We must remember factors such as team size and demands on the project, such as:

- As the number of people involved grows, so does the need to coordinate communications.
- As the potential for damage increases, the need for public scrutiny increases, and the tolerance for personal stylistic variations decreases.
- Some projects depend on time-to-market and can tolerate defects (Web browsers being an example); other projects aim for traceability or legal liability protection.

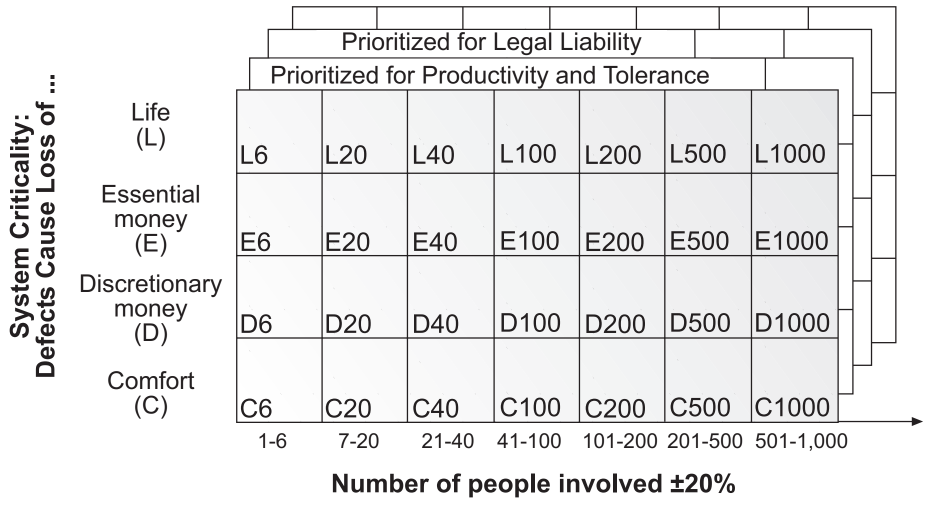
The result of collecting those factors is shown in Figure 6. The figure shows three factors that influence the selection of methodology: communications load (as given by staff size), system criticality, and project priorities.

Locate the segment of the X axis for the staff size (typically just the development team). For a distributed development project, move right one box to account for the loss of face-to-face communications.

On the Y axis, identify the damage effect of the system: loss of comfort, loss of "discretionary" monies, loss of "essential" monies (e.g., going bankrupt), or loss of life.

The different planes behind the top layer reflect the different possible project priorities, whether it is time to market at all costs (such as in the first layer), productivity and tolerance (the hidden second layer), or legal liability (the hidden third layer). The box in the grid indicates the class of projects (for example, C6) with similar communications load and safety needs and can be used to select a methodology.

Figure 6 — The family of Crystal methods.


Prioritized for Legal Liability
Prioritized for Productivity and Tolerance
System Criticality: Defects Cause Loss of ...
Life (L)
Essential money (E)
Discretionary money (D)
Comfort (C)
L6 L20 L40 L100 L200 L500 L1000
E6 E20 E40 E100 E200 E500 E1000
D6 D20 D40 D100 D200 D500 D1000
C6 C20 C40 C100 C200 C500 C1000
1-6 7-20 21-40 41-100 101-200 201-500 501-1,000
Number of people involved ±20%

The grid characterizes projects fairly objectively, useful for choosing a methodology. I have used it myself to change methodologies on a project as it shifted in size and complexity. There are, of course, many other factors, but these three determine methodology selection quite well.

Suppose it is time to choose a methodology for the project. To benefit from the project interviews mentioned earlier, create the lightest methodology you can even imagine working for the cell in the grid, one in which person-to-person communication is enhanced as much as possible, and running tested code is the basic timing marker. The result is a light, habitable (meaning rather pleasant, as opposed to oppressive), effective methodology. Assign this methodology to C6 on the grid.

Repeating this for all the boxes produces a family of lightweight methods, related by their reliance on people, communication, and frequent delivery of running code. I call this family the Crystal Light family of methodologies. The family is segmented into vertical stripes by color (not shown in figure): The methodology for 2-6 person projects is Crystal Clear, for 6-20 person projects is Crystal Yellow, for 20-40 person projects is Crystal Orange, then Red, Magenta, Blue, etc.

Shifts in the vertical axis can be thought of as "hardening" of the methodology. A life-critical 2-6-person project would use "hardened" Crystal Clear, and so on. What surprises me is that the project interviews are showing rather little difference in the hardness requirement, up to life-critical projects.

Crystal Clear is documented in a forthcoming book, currently in draft form on the Web. Crystal Orange is outlined in the methodology chapter of *Surviving Object-Oriented Projects* (see Editor's note below).

Having worked with the Crystal Light methods for several years now, I found a few more surprises.

The first surprise is just how little process and control a team actually needs to thrive (this is thrive, not merely survive). It seems that most people are interested in being good citizens and in producing a quality product, and they use their native cognitive and communications abilities to accomplish this. This matches Jim's conclusions about adaptive software development (see Resources and References). You need one notch less control than you expect, and less is better when it comes to delivering quickly.

More specifically, when Jim and I traded notes on project management, we found we had both observed a critical success element of project management: that team members *understand* and *communicate* their work dependencies. They can do this in lots of simple, low-tech, low-overhead ways. It is often not necessary to introduce tool-intensive work products to manage it.

Oh, but it is necessary to introduce two more things into the project: *trust* and *communication*.

A project that is short on trust is in trouble in more substantial ways than just the weight of the methodology. To the extent that you can enhance trust and communication, you can reap the benefits of Crystal Clear, XP, and the other lightweight methods.

The second surprise with defining the Crystal Light methods was XP. I had designed Crystal Clear to be the least bureaucratic methodology I could imagine. Then XP showed up in the same place on the grid and made Clear look heavy! What was going on?

It turns out that Beck had found another knob to twist on the methodology control panel: discipline. To the extent that a team can increase its internal discipline and consistency of action, it can lighten its methodology even more. The Crystal Light family is predicated on allowing developers the maximum individual preference. XP is predicated on having everyone follow tight, disciplined practices:

- Everyone follows a tight coding standard.
- The team forms a consensus on what is "better" code, so that changes converge and don't just bounce around.
- Unit tests exist for all functions, and they always pass at 100%.
- All production code is written by two people working together.
- Tested function is delivered frequently, in the two- to four-week range.

In other words, Crystal Clear illustrates and XP magnifies the core principle of light methods:

> Intermediate work products can be reduced and project delivery enhanced, to the extent that team communications are improved and frequency of delivery increased.

XP and Crystal Clear are related to each other in these ways:

- XP pursues greater productivity through increased discipline, but it is harder for a team to follow.

- Crystal Clear permits greater individuality within the team and more relaxed work habits in exchange for some loss in productivity.
- Crystal Clear may be easier for a team to adopt, but XP produces better results if the team can follow it.
- A team can start with Crystal Clear and move itself to XP. A team that falls off XP can back up to Crystal Clear.

Although there are differences in Crystal Clear and XP, the fundamental values are consistent — simplicity, communications, and minimal formality.

Editor's note: For more information on the Crystal Clear methodology, see Alistair Cockburn's Web site, listed in the References and Resources section. For more information on Crystal Orange, it is covered in the book Surviving Object-Oriented Projects, also listed in the References and Resources section.

CONCLUSIONS: GOING TO EXTREMES

Orr and Cockburn each describe their approaches and experience with lighter methodologies. But earlier, in describing Chrysler's C3 project, I alluded to the difficulty in extending the use of approaches like XP or even RAD. In every survey we have done of ***eAD*** subscribers, and every survey conducted of software organizations in general, respondents rate reducing delivery time as a critical initiative. But it is not just initial delivery that is critical. Although Amazon.com may have garnered an advantage by its early entry in the online bookstore market, it has maintained leadership by continuous adaptation to market conditions — which means continuous changes to software.

Deliver quickly. Change quickly. Change often. These three driving forces, in addition to better software tools, compel us to rethink traditional software engineering practices — not abandon the practices, but rethink them. XP, for example, doesn't ask us to abandon good software engineering practices. It does, however, ask us to consider closely the absolute minimum set of practices that enable a small, co-located team to function effectively in today's software delivery environment.

Cockburn made the observation that implementation of XP (at least as Beck and Jeffries define it) requires three key environmental features: inexpensive inter-face changes, close communications, and automated regression testing. Rather than asking "How do I reduce the cost of change?" XP, in effect, postulates a low-change cost environment and then says, "This is how we will work." For example, rather than experience the delays of a traditional relational database environment (and dealing with multiple outside groups), the C3 project used GemStone, an OO database.

Some might argue that this approach is cheating, but that is the point. For example, Southwest Airlines created a powerhouse by reducing costs — using a single type of aircraft (Boeing 737s). If turbulence and change are the norm, then perhaps the right question may be: how do we create an environment in which the cost (and time) of change is minimized? Southwest got to expand without an inventory of "legacy" airplanes, so its answer might be different than American Airline's answer, but the question remains an important one.

There are five key ideas to take away from this discussion of XP and light methods:

- For projects that must be managed in high-speed, high-change environments, we need to reexamine software development practices and the assumptions behind them.
- Practices such as refactoring, simplicity, and collaboration (pair programming, metaphor, collective ownership) prompt us to think in new ways.
- We need to rethink both how to reduce the cost of change in our existing environments and how to create new environments that minimize the cost of change.
- In times of high change, the ability to refactor code, data, and whole applications becomes a critical skill.
- Matching methods to the project, relying on people first and documentation later, and minimizing formality are methods geared to change and speed.

EDITOR'S MUSINGS

Extreme rules! In the middle of writing this issue, I received the 20 December issue of *BusinessWeek* magazine, which contains the cover story, "Xtreme Retailing," about "brick" stores fighting back against their "click" cousins. If we can have extreme retailing, why not Extreme Programming?

Refactoring, design patterns, comprehensive unit testing, pair programming — these are not the tools of hackers. These are the tools of developers who are exploring new ways to meet the difficult goals of rapid product delivery, low defect levels, and flexibility. Writing about quality, Beck says, “The only possible values are ‘excellent’ and ‘insanely excellent,’ depending on whether lives are at stake or not” and “runs the tests until they pass (100% correct).” You might accuse XP practitioners of being delusional, but not of being poor-quality-oriented hackers.

To traditional methodology proponents, reducing time-to-market is considered the enemy of quality. However, I’ve seen some very slow development efforts produce some very poor-quality software, just as I’ve seen speedy efforts produce poor-quality software. Although there is obviously some relationship between time and quality, I think it is a much more complicated relationship than we would like to think.

Traditional methodologies were developed to build software in environments characterized by low to moderate levels of change and reasonably predictable desired outcomes. However, the business world is no longer very predictable, and software requirements change at rates that swamp traditional methods. “The bureaucracy and inflexibility of organizations like the Software Engineering Institute and practices such as CMM are making them less and less relevant to today’s software development issues,” remarks Bob Charette, who originated the practices of lean development for software.

As Beck points out in the introduction to his book, the individual practices of XP are drawn from well-known, well-tested, traditional practices. The principles driving the use of these practices, along with the integrative nature of using a specific minimal set of practices, make XP a novel solution to modern software development problems.

But I must end with a cautionary note. None of these new practices has much history. Their successes are anecdotal, rather than studied and measured. Nevertheless, I firmly believe that our turbulent e-business economy requires us to revisit how we develop and manage software delivery. While new, these approaches offer alternatives well worth considering.

In the coming year, we will no doubt see more in print on XP. Beck, Jeffries, Fowler, and Cunningham are working in various combinations with others to publish additional books on XP, so additional information on practices, management philosophy, and project examples will be available.

Resources and References

Books and Articles

Beck, Kent. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

Cockburn, Alistair. *Surviving Object-Oriented Projects*. Addison-Wesley, 1998.

Cusumano, Michael A. and Richard Selby. *Microsoft Secrets.* Free Press, 1995.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Gilb, Tom. *Principles of Software Engineering Management.* Addison-Wesley, 1988.

Higgins, David. *Data Structured Software Maintenance*. Dorset House Publishing, 1986.

Yourdon, Edward and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice Hall, 1986.

General Resources

Adaptive software development. See the article in the August 1998 ***Application Development Strategy*** (now ***eAD***), “Managing Complexity.”

ARTech. Montevideo, Uruguay. Web site: www.artech.com.uy. Developers of GeneXus.

Crystal Clear method. Web site: http://members.aol.com/humansandt/crystal/clear.

Alistair Cockburn. Web site: http://members.aol.com/acockburn.

Bob Charette. Lean Development. ITABHI Corporation, 11609 Stonewall Jackson Drive, Spotsylvania, VA 22553, USA. E-mail: charette@erols.com.

Ward Cunningham’s Extreme Programming Roadmap. Web site: http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap.

eXtreme Programming and Flexible Processes in Software Engineering — XP2000 Conference. 21-23 June 2000, Cagliari, Sardinia, Italy. Web site: http://numa.sern.enel.ucalgary.ca/extreme.

Martin Fowler. Web site: http://ourworld.compuserve.com/homepages/martin_fowler/.

Ron Jeffries. E-mail: ronjeffries@acm.org. Web site: www.XProgramming.com.

Lean Development. See the November 1998 ***ADS*** article “Time is of the Essence.”

Object Mentor, Green Oaks, IL, USA. Web site: www.objectmentor.com/.

Ken Orr, Ken Orr Institute, Topeka, KS, USA. Web site: www.kenorrinst.com.

Laurie Williams. Web site: www.cs.utah.edu/~lwilliam.