A Simple Model of Agile Software Processes
– or –
Extreme Programming Annealed

Glenn Vanderburg
2240 Dampton Dr.
Plano, TX 75025
glv@vanderburg.org

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *software process models (e.g., CMM, ISO, PSP), life cycle, programming teams.*

General Terms
Management.

Keywords
Extreme Programming, Agile software development, Process customization.

INTRODUCTION
"If you built a piece of software that was as tightly coupled as Extreme Programming, you'd be fired."

It was late 1999, and I was sitting at lunch with Pragmatic Dave Thomas and the rest of the North Texas XP interest group. It's not unusual for Dave to make provocative statements like that, but this time I was dumbstruck. From the beginning, I had liked Extreme Programming's redundancy and interconnectedness—it seemed like a strength of the process to me. Plus, there were some easy answers. "But people aren't software components" was one I can remember almost voicing.

And yet there was a deeper point Dave was making, one that I didn't have an answer for. He was right. The same characteristic that I appreciated in XP, I would decry in a software design. Glib answers notwithstanding, I could see how that tight coupling could have a strong negative impact for a software process, too. For any process, Extreme or not, to be really useful and successful in a variety of situations for different teams, we have to understand how to tailor it. Every project team inevitably augments, trims, or otherwise tailors the process they set out to use. The problem is that most of the time we do it blindly. Oh, sure … we may have an idea what problem we're trying to solve by adding some new practice, or some reason that we don't need a particular artifact. But process elements don't exist in isolation from one another. Typically, each provides input, support, or validation for one or more other process elements, and may in turn depend on other elements for similar reasons.

Is this internal coupling as bad for software *processes* as it is for software? Dave, Chris Morris, and I spent some time discussing that question, but soon we all became distracted by other things. After a while, though, I found myself thinking about the question again. I think it's an important question not just for Extreme Programming, but for all software processes. Until we understand how process elements depend upon and reinforce one another, process design and tailoring will continue to be the hit-or-miss black art that it is today.

Extreme Programming is an excellent subject for studying internal process dependencies. One reason is that it acknowledges those dependencies and tries to enumerate them (Kent Beck's *Extreme Programming Explained* devotes a chapter to explaining many of them[1]). Additionally, XP is unusual in covering not just the management of the project, but day-to-day coding practices as well. It provides an unusually broad (if not necessarily complete) picture of the software development process.

TIGHTLY COUPLED
The published literature about Extreme Programming is incomplete in several ways. If you follow discussions of how successful teams actually apply XP, you'll see that there are many implicit practices, including the physical layout of the team workspace and fixed-length iterations. Likewise, since relationships between practices are more difficult to see than the practices themselves, it's probable that there are unidentified relationships between the practices—perhaps even strong, primary dependencies.

However, just diagramming the twelve explicit XP practices and the relationships documented in *Extreme Programming Explained* shows the high degree of interconnectedness, as seen in Figure 1.

http://doi.acm.org/10.1145/1094811.1094854

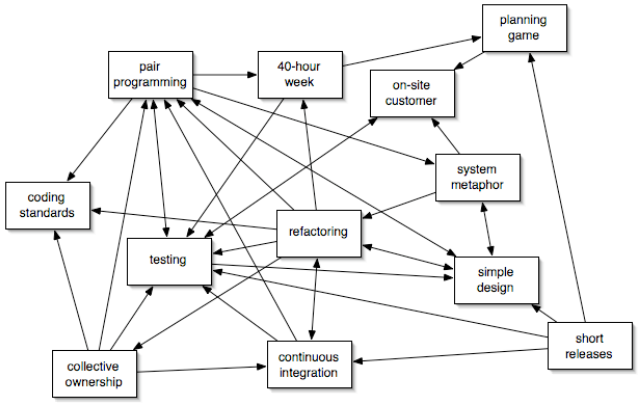

Figure 1. The original 12 practices and their dependencies.

Rather than add additional complications to the problem right from the start, I decided to focus on the relationships Beck described. The one change I made from the beginning was to split the "testing" practice into "unit testing" and "acceptance testing." They are different activities, and the XP literature emphasizes the differences in their purpose, timing, and practice, so it seemed appropriate to treat them as distinct practices. Therefore, instead of the original twelve practices of Extreme Programming, this analysis deals with the thirteen shown in Figure 2.

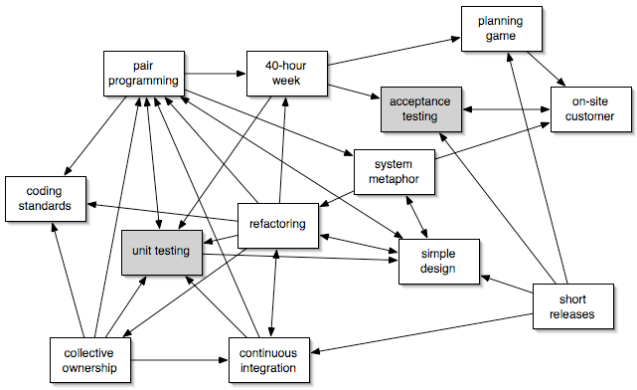

Figure 2. The thirteen practices.

Once the complex web of dependencies is shown so clearly, it's easy to understand Dave Thomas' point and the challenge implicit in it. Are those interdependencies worth their cost? Given that there are good reasons to customize a chosen software process, can you even start to do it sensibly in an XP context? If you have to omit or modify one of the XP practices, how can you understand what you're really losing? Can you do a reasonable job of choosing another practice (or set of practices) to fill *all* the roles, primary and secondary, that the original fills? If you notice a problem on your project that XP isn't adequately addressing, how can you fit a new practice into this web?

That became my goal—understanding these dependencies well enough to permit informed adjustment. The point is not to "decouple" Extreme Programming. I believe those interdependencies are beneficial. Software processes involve humans, with all of our failings, weaknesses, and inconsistencies. We have good days and bad days, we follow processes inconsistently and imperfectly, and we overlook things: requirements, scenarios, errors, better designs, and things our tools can do for us. Some process redundancy is invaluable in the face of such flawed workers.

Many processes try to deal with these problems by strengthening the practices, adding enforcement steps or inspectors, or by adding practices solely for the purpose of redundancy. But such measures are costly in terms of time and effort, and they probably also harm team morale and cohesion. One strength of the XP approach is that the practices play multiple roles. In most cases when an XP practice serves to compensate for the flaws of another practice, the redundant compensation is merely a secondary role of the practice. This helps keep the number of practices to a minimum, and has the added benefit of using core team members in enforcement roles without making them seem like "enforcers."

Without *some* coupling, even in software designs, nothing will ever get done. The trick is to build relationships between components when they are appropriate and helpful, and avoid them otherwise. The coupling within XP is only harmful if it makes the process difficult to change. If we can understand the relationships well enough, perhaps they would not be barriers to making appropriate changes to the process.

TEASING OUT THE TANGLES

Is there some underlying structure to these dependencies that will make them more comprehensible? Are there (in the language of graph theory) strongly connected subcomponents that have weaker connections between them? Alternatively, are the relationships ordered in some way? When Dave, Chris, and I began discussing this problem, I instinctively felt that there was some structure that was not yet understood. I began drawing dependency graphs, moving nodes around, looking for some hint of that structure.

Dave pointed out that what I was trying to do was similar to the metallurgical process of *annealing*, where a metal is heated and then slowly cooled to strengthen it and reduce brittleness. The process allows the molecules of the metal, as it cools, to assume a tighter, more nearly regular structure. Some automated graph-drawing algorithms employ a process of *simulated annealing*, jostling the nodes of the graph randomly and adjusting position to reach an equilibrium state that minimizes the total length of the arcs in the graph[6].

Assuming there is a structure to those dependencies, how could I discover it? Graph drawing algorithms didn't help, and neither did more *ad hoc* graph layout methods. Figure 2 is, in fact, the cleanest two-dimensional arrangement I was able to achieve with all thirteen practices and their dependencies.

Neither was it particularly helpful to try sorting the graph topologically. It's full of cycles. After that, I tried to visualize clusters of dependencies by arranging the practices in a circle and changing the order to bring related closely related practices close together (this amounted to a kind of circular topological sort). In the process I did notice that there were ways to arrange the practices so that only a few of the dependencies—9 out of a total of 38—skipped more than three intervening practices (as shown in

Figure 3). In that arrangement, most of the dependencies were between nearby practices. That could have been coincidental, but I began investigating that particular ordering. What did practices that were close to each other on the circle have in common? What distinguished practices on opposite sides of the circle?

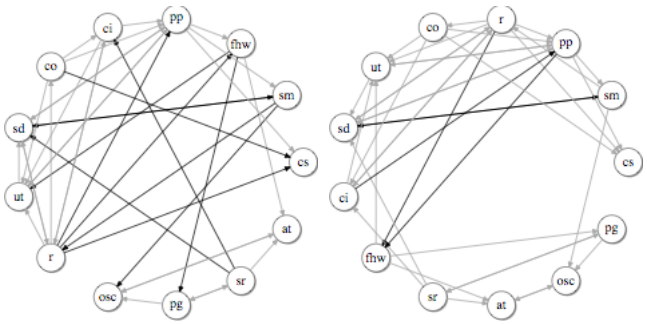

Figure 3. Before and after a "circular topological sort."

What I discovered was that there were often—not always—particularly strong relationships between practices that operate on similar scales. The low-level programming practices depend on each other more than they depend on the product-scale practices like the planning game and short releases. That doesn't seem particularly startling, but for want of any other ideas, I began exploring the relationships between practices and scales.

There are nine practices that seem to operate at particular scales, as illustrated in Figure 4. Each of these practices seems to provide feedback about particular kinds of decisions, from very small to the large, sweeping decisions. I am sure there are many arguments to be had over the particular ordering, but the basic trend is clearly from small scales to large.

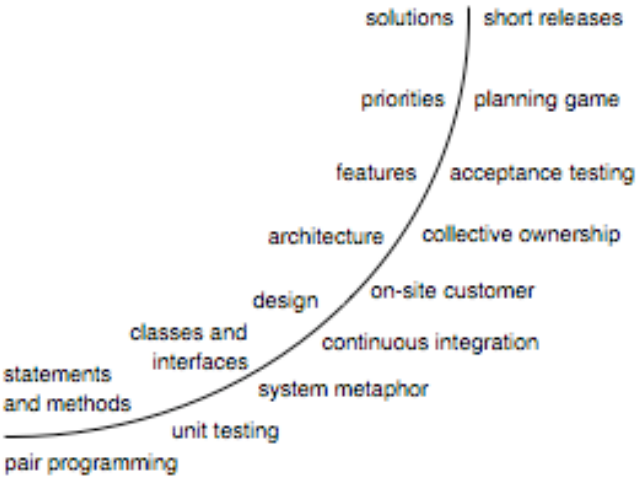

Figure 4. Scale-defined practices

Of course, that leaves four other practices out, which is a problem when you're trying to understand all of the practices and how they relate. But for now, we may be onto something, and perhaps we can make sense of the others later.

Armed with this observation about scales, I began to see that not all of the dependencies within XP are of the same *kind*. For example, consider the bidirectional dependency between pair programming and unit testing. How does pair programming help unit testing? It *strengthens* unit testing by suggesting good tests, and by encouraging the unit-testing discipline. It also helps to ensure that the unit-testing process is dealing with well-designed code, making the testing process itself more efficient and productive.

Now turn it around. How does unit testing support pair programming? It *guides* the programmers by helping them structure their work, setting short-term goals on which to focus. It guides their design work as well; unit testing has well known benefits as a design technique. It also *defends* against shortcomings of pair programming (even two minds don't write perfect code) by catching errors.

Do the relationships at larger scales look similar? Another bidirectional dependency on a larger scale exists between on-site customer and acceptance testing. The relationship between the two is clearly different in details from the one we just explored between pair programming and unit testing, but it seems to me to be similar in terms of the respective roles of the two practices. Having an on-site customer strengthens acceptance testing by guiding the development of tests, and by helping maintain correspondence between stories and tests. In the opposite direction, acceptance testing *guides* feature development (again by providing goals) and *defends* against the weaknesses of on-site customer, providing a concrete, executable record of key decisions the customer made that might otherwise be undocumented. It also provides a testbed for the consistency of customer decisions.

At all of these scales, the characteristics of the dependencies seem similar. Smaller-scale practices *strengthen* larger-scale practices by providing high-quality input. In other words, smaller-scale practices take care of most of the small details so that the larger-scale practices can effectively deal with appropriately scaled issues. In the reverse direction, larger-scale practices *guide* smaller-scale activities, and also *defend* against the mistakes that might slip through.

Does this help make sense of the four remaining practices? Refactoring, forty-hour weeks, simple design, and coding standards seem to all have a strengthening role. One way of looking at the strengthening dependencies is to see them as *noise filters*. The "noise" I'm speaking of is (to use Fred Brooks' terminology[3]) the *accidental* complexity: the extra complexity in our systems over and above the *essential* complexity that is inherent in the problem being solved. In a software system, that noise can take many forms: unused methods, duplicate code, misplaced responsibility, inappropriate coupling, overly complex algorithms, and so on. Such noise obscures the essential aspects of the system, making it more difficult to understand, test, and change.

The four practices that operate independent of scale seem to be aimed at reducing noise, improving the overall quality of the system in ways that allow the other practices to be more effective. Refactoring is an active practice that seeks to filter messy code

from the system whenever it is found. Simple design and coding standards are yardsticks against which the system's quality can be measured, and help guide the other practices to produce a high quality system. (It's arguable whether those are actually "practices" at all; rather, they're criteria we use to guide the other practices.) Finally, forty-hour week helps eliminate mistakes by reducing physical and mental fatigue in the team members. The four noise-filtering practices, along with their interdependencies, are shown in Figure 5.

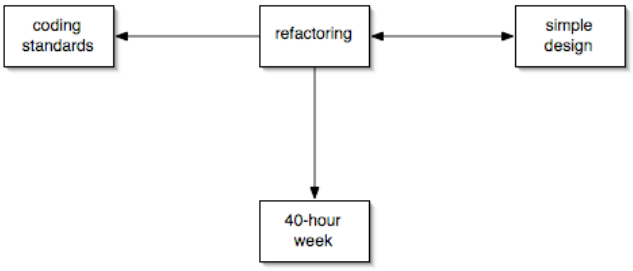

Figure 5. Noise filters.

Those four noise-filtering practices help many of the other practices to operate more effectively by maximizing clarity and reducing complexity in the code. They help minimize the accidental complexity in the system in favor of the essential complexity. Many of the other practices perform similar functions (albeit more limited in scale) in their upward strengthening relationships with other practices.

So far, so good. We have a set of four *noise filters*, and nine other practices that operate in a rough hierarchy of scales, *strengthening* the practices at larger scales while *defending* against mistakes that slip through the lower-scale activities.

A FEEDBACK ENGINE

There's more going on, though. The nine practices are characterized not only by the scale of entity they work with; additionally, they function primarily within a certain span of time. Not surprisingly, the practices that operate on small-scale things also operate very quickly. The correspondence between practices and time scales is shown in Figure 6. Again, while we may quibble about the details of this ordering, the trend from smaller to larger increments is clear, as is the general correspondence with the earlier ranking from Figure 4.

We see, therefore, that the practices that deal with small-scale entities also operate very rapidly, whereas practices that deal with larger-scale issues take longer to iterate. As for the other practices—the noise filters—just as they are independent of scale, they also seem to function across small or large timeframes, as appropriate.

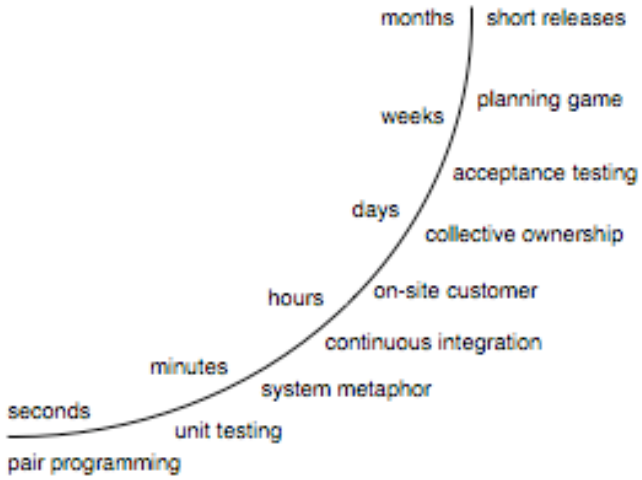

Figure 6. Practices and time scales.

There's a structure here that's comprehensible, unlike the original web of undifferentiated dependencies. I assert that the nesting of XP's feedback loops is the fundamental structural characteristic of Extreme Programming. All of the explicit dependencies between individual practices that have been identified by Beck and others—while important—are natural consequences of this overall structure, and not independent features that need to be managed.

However, we're still not to the point of being able to use this structure to help us tailor the process sensibly. For that, we must turn to traditional models of software processes and understand where this model fits into them.

Cost of Feedback

Barry Boehm's *cost of change curve* is one of the linchpins of software engineering theory and practice[2]. Boehm's observations of projects led him to conclude that, as projects advance through their lifecycles, the cost of making necessary changes to the software increases, and does so exponentially.

This observation led to a generation of processes that were designed to make all changes—all decisions—as early in the process as possible, when changes are cheaper. There are many things wrong with such strategies, but one in particular concerns me here. Many in the agile community have observed that Boehm's study dealt primarily with projects using a waterfall-style process, where decisions (in the form of requirements gathering, analysis, and design) were made very early in the project. Those decisions were often carefully scrutinized to identify mistakes, but the only true test of software is to run it. (Recall Knuth's famous warning to Peter van Emde Boas: "Beware of bugs in the above code; I have only proved it correct, not tried it.") In classic waterfall projects, such empirical verification typically didn't happen until near the end of the project, when everything was integrated and tested. Agile, iterative processes seem to enjoy a shallower cost-of-change curve, suggesting that perhaps Boehm's study was actually showing how the cost of change increased as a function of the

length of the feedback loop, rather than merely the point in the project lifecycle.

As I mentioned, that analysis of the cost of change curve is not new to the agile process community. But I believe understanding XP's structure sheds new light on how the process manages that curve. With its time- and scale-sensitive practices and dependencies, XP is an *efficient* feedback engine. Its nested feedback loops, each one optimized for the size of decision involved, don't just hasten feedback; they do so in very cost-effective ways. Very small decisions, such as those made when writing statements and methods in a program, are very cheap to validate, so XP projects get that feedback continuously, minute-by-minute, through interactions within programming pairs and through unit testing. Larger decisions, such as the selection of features to help solve a business problem and the best way to spend project budget, are quite costly to validate. Therefore XP projects validate those decisions somewhat more slowly, through day-to-day interaction with customers, giving the customer control over each iteration's feature choice, and by providing a release (for production use, if desired) every few weeks at most. At every scale, Extreme Programming's practices provide feedback in a way that balances timeliness and economy.

(That XP relies heavily on feedback is also old news; many people have made that observation, including Beck and Alistair Cockburn[4]. What isn't widely known is how the notion of nested feedback loops serves as a model for understanding the dependencies within XP.)

Defense in Depth

Another traditional view of the purpose and function of a software process—closely related to managing the cost of change—is that it is *defensive*, guarding against the introduction of defects into the product.

Our model of XP's inner structure also makes sense when measured against this view. In fact, it resembles the timeworn security strategy of *defense in depth*. Extreme Programming can be seen as a gauntlet of checks through which every line of code must pass before it is ultimately accepted for inclusion in the final product. At each stage, it is likely that most defects will be eliminated … but for those that slip through, the next stage is waiting. Furthermore, the iterative nature of XP means that in most cases code will be revisited, run through the gauntlet again, during later iterations.

One important difference between Extreme Programming's defensive strategy and those of more traditional processes is the notion of what constitutes a defect. Bugs, of course, are viewed by both process models as defects to be avoided. Traditional processes see a second big category of defects: missed or incorrect requirements. XP, on the other hand, like other agile processes, sees changing requirements as inevitable, and even as a business advantage. Instead of working so hard to guard against missed requirements, XP actively guards against another kind of defect altogether: unnecessary complexity that will inhibit change. The practices I've called *noise filters* are specifically aimed at keeping those defects at bay.

OTHER PROCESSES

I've focused on Extreme Programming primarily because XP is unusual in two respects: mandating practices at all scales of the development process, and acknowledging (and even enumerating) its internal dependencies. It was those characteristics, in fact, that prompted Dave Thomas to make the observation that started me on this path.

I don't believe, however, that this structural pattern is peculiar to Extreme Programming. Many agile processes identify "feedback" as a guiding principle, and explicitly provide opportunities to gather feedback.

Take Scrum, for example. The overall structure of Scrum is a series of iterations, of course—such iterations are a central feedback mechanism used in all agile processes. Scrum's iterations take the form of 30-day "sprints." After each sprint comes a "sprint review," designed to understand how successful the sprint was and make adjustments for the next sprint. Within the sprint, the core feedback mechanism is the daily scrum meeting. It's designed to be inexpensive (requiring just a few minutes from the team each day) but effective. The team itself can respond to issues identified in the scrum meeting; the fact that feedback is gathered in a *whole-team* setting such as the scrum meeting probably amplifies its effect.

As in XP, Scrum's feedback mechanisms are sized to match the scale of artifact being examined. Sprints (and the accompanying sprint planning and review exercises) are costly, large-scale cycles, and are focused in large part on overall quality, the suitability of the solution to the task, and overall team velocity (averaged over the course of a 30-day sprint). They are so costly that they are carefully controlled; Scrum strongly advises against reducing the length of sprints, and aborting a sprint in the middle is an extraordinary event, with carefully defined conditions. Scrum meetings, on the other hand, are inexpensive, frequent, and explicitly focused on the previous day and the next—and therefore on the day-sized tasks: individual features, test completion, development environment issues, and so on.

Unlike XP, Scrum doesn't explicitly have thorough coverage of feedback at many different scales. However, most Scrum teams adopt *ad hoc* practices to help them monitor the health of the project during sprints. One popular example is the use of an automated build-and-test server, which is a way of providing hourly or daily feedback about unit correctness and interfaces between units and components.

Scrum's creators explicitly acknowledge the central role of feedback (although the role of scale is implicit):

> Scrum employs the empirical process control model. Scrum regularly inspects activities to see what is occurring and empirically adapts activities to produce desired and predictable outcomes. [...] Empirical process control models are elegantly simple. They employ feedback mechanisms to monitor and adapt to the unexpected, providing regularity and predictability[7].

In addition, Scrum's creators focus on the problem of *noise* in a software project:

> In this context, the term "noise" refers not to a sonic phenomenon, but to the unpredictable, irregular, nonlinear parts of system development. [...] When noise-to-signal ratio is too high, the sound of what I want to hear is obscured by the sound of what I don't want to hear, or the noise[7].

The large amount of noise in modern system development projects is part of the motivation for Scrum's focus on an empirical process-control model, focused on feedback mechanisms.

THE AGILE METHODOLOGIST

Armed with these observations, I believe we can make sensible, informed decisions about how to tailor Extreme Programming. If, for some reason, you are unable to implement one of XP's practices on your project, what do you put in its place?

If it's one of the scale-dependent process, replace it with another practice that's designed to provide feedback on roughly the same scale of decisions, more or less as rapidly, for roughly the same cost. You should not have to worry about the details of how it supports adjacent practices; it will fill that role naturally, by virtue of providing feedback at the appropriate scale. It's unlikely that you'll find an exact match for one of the standard practices; you can expect the replacement to work a little more slowly, or be a little more costly, or to let a few more defects slip through, or be a less effective guide for the practices at smaller scales. You may need to strengthen your execution of the practices at adjacent scales to compensate a bit … but your first try is likely to work well enough to avoid major difficulties, providing time to learn the weaknesses of the new practice and react appropriately.

As an example, pair programming frequently presents problems for teams, whether because of geographic separation, inappropriate facilities, or skeptical management. How would you choose new practices to compensate for the loss of pair programming?

It's easy to see why the traditional answer, the group code inspection, is unsuitable. It is too costly and too slow, providing feedback days (or even weeks) later about much quicker, smaller-scale decisions. Instead, since a pair can't work together during a task, the team might try bracketing the task with shorter bursts of collaboration. A short design session before the task could provide feedback about the proposed strategy for implementing the task. Upon task completion, it would be good to have a quick, one-person review of the completed work, focusing on general style (for the production code) and thoroughness (for the unit tests). In this way, team members could receive rapid, economical feedback about the key decisions made during the implementation of a single small task: have coding standards been followed? Is the design simple and straightforward? Are the unit tests thorough?

What if you need to replace one of the noise-filtering practices? Those are a lot tougher to do without. A project can suspend them for a while, incurring "technical debt"[8], but the debt must soon be paid off or it will become a barrier to change. I'm not sure any agile project can live without these four practices for long. Unnecessary complexity is a defect to avoided, and not only does it inhibit change, it also breeds additional, more obvious defects. Agile teams need to think clearly, write consistent code, and keep the design simple. And in the absence of perfect requirements, perfect people, and perfect practices, extra complexity will creep in, requiring refactoring. This analysis has confirmed, for me, an intuition that those four practices are essential for any agile project.

The process may also need some tinkering when things aren't going as well as they should be. Perhaps team velocity is slowing, or older acceptance tests begin failing as new features are developed. How can a team augment the process to fill the gap?

The first tactic should probably be to examine the existing practices, to see if they're being applied properly. But if a new practice of some sort is required, it should be constructed to fit into the existing scale hierarchy. Defects—possibly in the form of extra complexity—are creeping in. At what scale? A new practice should provide hard-to-ignore feedback about appropriately scaled decisions, in a timeframe that's also appropriately scaled.

Alistair Cockburn recommends what he calls "just-in-time methodology construction"[4]. A team, he says, should actively watch their project for signs of trouble, changing their process as needed to address issues as they arise. The Unified Process similarly recommends that UP be "configured" for each project[5]. Risks should be identified and then the UP roles, activities, and artifacts that aren't needed to address those risks can be dropped from the process for the current project.

Cockburn and the designers of UP are right: we need to tailor our processes. Unfortunately, there is very little guidance about how to do that wisely—and in the absence of such guidance, most teams end up using processes that are inadequate or far too costly, or (worst of all) both of those things.

Extreme Programming has some tight coupling between its practices. But I've come to believe that my instinct was right: the redundant, "organic" interconnectedness of XP is the source of a lot of its robustness and speed. All those dependencies between practices have a structure that is actually fairly simple. (I believe that structure can help us identify previously unidentified relationships, although I have not pursued that analysis in the current essay.) That structure, once identified, provides crucial guidance for those who need to tailor and adjust the software process.

The feedback engine, with its nested feedback loops, is an excellent model for a process designed to manage the cost of change and respond efficiently to changing requirements. This is the essence of agility: letting go of the slow, deliberate decision-making process in favor of quick decisions, quickly and repeatedly tested. The feedback loops are optimized to validate decisions as soon as possible while still keeping cost to a minimum. Finally, that multi-scale hierarchy of feedback loops, once recognized, provides crucial guidance when we need to tailor and adjust the software process.

ACKNOWLEDGMENTS

I'm grateful to Dave Thomas for the comment that got me started thinking about this issue, and to him, Chris Morris, and Tom McGraw for crucial early discussions. Deborah Vanderburg, Mike Clark, and David Butler performed valuable, careful reviews of drafts of this essay, as did Brian Marick and the anonymous reviewers.

I recently learned that William Wake has also discussed the relationship between XP practices and time scales[9].

REFERENCES

[1] Beck, K. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 1999.

[2] Boehm, B. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[3] Brooks, F. No Silver Bullet. Reprinted in *The Mythical Man-Month (Anniversary Edition).* Addison-Wesley, Boston, 1995.

[4] Cockburn, A. *Agile Software Development.* Addison-Wesley, Boston, 2002.

[5] Jacobson, I., Booch, G., and Rumbaugh, J. *The Unified Software Development Process.* Addison-Wesley, Reading, MA, 1999.

[6] Kirkpatrick, S., Gelatt Jr., C.D., and Vecchi, M.P. Optimization by Simulated Annealing. *Science, 4598* (13 May 1983), 671–680.

[7] Schwaber, K. and Beedle, M. *Agile Software Development with Scrum*. Prentice Hall, Upper Saddle River, NJ, 2002.

[8] Smith, D. Technical Debt. *Portland Pattern Repository.* January 20, 2005. <http://c2.com/cgi/wiki?TechnicalDebt>

[9] Wake, W. Extreme Programming as Nested Conversations. *Methods and Tools, 10:4* (Winter 2002), 2–12. <http://www.methodsandtools.com/PDF/dmt0402.pdf>