Management Challenges to Implementing Agile Processes in Traditional Development Organizations

Barry Boehm, *University of Southern California*
Richard Turner, *Systems and Software Consortium*

Managers face several barriers, real and perceived, when they try to bring agile approaches into traditional organizations. These strategies can help address them.

Our discussions with traditional developers and managers concerning agile software development practices nearly always contain two somewhat contradictory ideas. They find that on small, stand-alone projects, agile practices are less burdensome and more in tune with the software industry's increasing needs for rapid development and coping with continuous change. (The "Agile Processes" sidebar explains more about their characteristics and why you might find them useful.) However, they're frustrated with the difficulty of scaling up and integrating them into traditional, top-down systems development organizations. When pressed for reasons, the usual response is a daunting litany of barriers.

In March 2004, the University of Southern California Center for Software Engineering (USC-CSE) Affiliates Annual Research Review held the fourth in a series of annual workshops to identify as many of these barriers as possible and to discuss whether or not they represented real conflicts with traditional engineering (http://sunset.usc.edu). Participants included agile developers, traditional aerospace and telecommunications developers, agile method creators, and academics. The results were a collection of change-related challenges and a list of nearly 40 perceived barriers to agile implementation (see the "Categories of Barriers to Agile Processes" sidebar). Workshop participants arrived at a consensus that some of these perceived barriers were nonproblems (for more on these, see the "Nonproblems" sidebar).

They categorized the other barriers either as problems only in terms of scope or scale, or as significant general issues needing resolution. From these two categories, we've identified three areas—development process conflicts, business process conflicts, and people conflicts—that we believe are the critical challenges to software managers of large organizations in bringing agile approaches to bear in their projects.

Development process conflicts

This is the first and perhaps most obvious area of difficulty. How do you merge agile, lightweight processes with standard industrial

0740-7459/05/$20.00 © 2005 IEEE

processes without either killing agility or undermining the years you've spent defining and refining your systems and software engineering process assets? Forging alloys of such disparate materials is technically and managerially challenging, and managers must be creative to succeed.

The annual workshops have indicated that several large aerospace, manufacturing, and IT companies have begun experimenting with agile methods. Most have used enthusiastic agile early-adopters on small pilot programs consisting either of pure agile methods applied to isolated (or possibly failing) projects or a hybrid of agile and traditional methods in a generally low-risk application. While almost all these pilots have been successful, the companies have been only minimally able to extend, evolve, or interoperate the agile processes or products.

Variability

Managing variability in subsystems and teams has proven difficult. If both agile and traditional teams are developing software for the same product, they can develop radically different artifacts that might not integrate easily. Without some means of coordination, an agile team's domain assumptions, GUIs, or commercial off-the-shelf choices could vary significantly from other developers' counterpart assumptions. Product functionality might change in order of delivery, or the agile team might change its design specifics as properties emerge and the team incorporates customer feedback.

Consider interface definitions. If the agile team evolves its own interfaces, it might leave other parts of the team at risk for developing against a changing standard. However, the traditional approach of locking the interface specification early could encumber the agile team's need to refactor some part of their design.

Larger organizations must pay specific attention to identifying how to synchronize teams. Some organizations have experienced success with medium-sized teams-of-teams, but multiple 15-minute daily meetings can become untenable. Finding enough competent team leaders who possess the necessary mix of technical, people, and agility skills can also prove difficult.

Different life cycles

Working with different life cycles is also difficult. Agile processes focus on immediately delivering functionality, while traditional methods focus on optimizing development over a longer period. The traditional longer life cycles require adjustments to the agile processes. For example, the documentation traditional methods require for training and support isn't a natural output of agile methods.

On the other hand, test-driven design is an example of an agile process that readily supports work in the maintenance phase by automatically providing regression tests. In fact, agile development makes the entire development cycle much more like a maintenance phase by providing short, focused iterations.

Legacy systems

Applying agile processes to legacy systems, whether within maintenance or as new development, raises numerous issues. Legacy systems generally aren't easy to refactor or disassemble to accommodate agile replacements that need to build capability in increments. Legacy systems might also institutionalize awkward and often sclerotic business processes that are embedded in the culture and aren't easy to refactor away.

Requirements

Differences between how agile and traditional approaches perform the requirements process can also cause problems. Agile requirements tend to be primarily functional and reasonably informal. This might or might not work in your systems engineering verification and validation approach. Strengthening the agile requirements approach to provide additional information might be necessary. For example, you might need to add information to stories or other requirements statements to more specifically address nonfunctional requirements such as reliability or security.

Suggestions

The following approaches can help organizations integrate agile practices into their traditional processes.

Do some serious preparation up front. Conduct a significant analysis of existing and proposed processes to identify mismatches in process requirements and expectations.

Build up processes rather than tailoring them down. Look at the project's needs and select only those process assets that seem indispensable. Of course, watch for instances in which

How do you merge agile, lightweight processes with standard industrial processes without killing agility?

Agile Processes

In general, agile methods are lightweight processes that employ short iterative cycles, actively involve users to establish, prioritize, and verify requirements, and rely on a team's tacit knowledge as opposed to documentation. A truly agile method must be iterative (take several cycles to complete), incremental (not deliver the entire product at once), self-organizing (teams determine the best way to handle work), and emergent (processes, principles, and work structures are recognized during the project rather than predetermined). Figure A shows an example of an agile process flow.

Key concepts and practices

The practices espoused to support these values vary with the method. They belong to three general areas:

- communication (for example, metaphor and pair programming),
- management (for example, planning game and fast cycle/frequent delivery), and
- technical (for example, simple design, refactoring, and test-driven design).

Examples of these agile concepts and practices include the following:

- Embracing change: Seeing change as an ally rather than an enemy. Change allows for more creativity and quicker value to the customer.
- Fast cycles, frequent delivery: Scheduling many releases with short time spans between them forces implementation of only the highest priority functions, delivers value to the customer quickly, and speeds requirements emergence. Timeboxing, for example, establishes specific time frames that are then filled with as much prioritized functionality as can be developed.
- Simple design: Designing for the battle, not the war. The motto is YAGNI (You Aren't Going to Need It). The anti-motto is BDUF (Big Design Up Front). Strip designs down to cover just what you're developing. Since change is inevitable, planning for future functions is a waste of effort.
- Refactoring: Restructuring software to remove duplication, improve communication, simplify, or add flexibility without changing its behavior; just-in-time redesign.
- Pair programming: A style of programming in which two programmers work side by side at one computer, continually collaborating on the same design, algorithm, code, or test.
- Retrospective or reflection: A post-iteration review of the effectiveness of the work performed, methods used, and estimates. The review supports team learning and estimation for future iterations.
- Tacit knowledge: Establishing and updating project knowledge in the participants' heads rather than in documents (explicit knowledge).
- Test-driven development: Developers and customers incrementally write module or method tests before and during coding. Supports and encourages very short iteration cycles.

Why implement agile approaches?

Obviously, integrating agile approaches and philosophies into traditional environments is difficult. Otherwise we wouldn't be discussing the challenges. If it's so difficult, then why should companies go to all this trouble? Here are a few reasons we've come upon in our research:

- Working software is often a better measure of progress than a force-fit earned-value ratio.
- You can often identify unnecessary or low-value functions early and therefore not spend time implementing them.

you might need more rigor and add necessary process components.

Define specific functionality or responsibilities that you're going to address with agile approaches. Small, GUI-intensive applications with short life cycles are an example. As we mentioned earlier, maintenance can also be a good place to experiment with agile approaches.

Develop architectures that support compartmentalization of agile and traditional teams. Identify an "agility level" characteristic for your architectural evaluations. Look for areas where requirements will likely change rapidly or where the design approach is unclear and designate these as agile opportunities.

Realign or redefine traditional milestone reviews to better fit an iterative approach. Fashion the standard reviews to be more like the Life Cycle Anchor Point milestones in the Win-Win Spiral[1] or Rational[2] processes. Move the preliminary design review out to where it makes sense to talk about the design based on developed code and prototypes.

Implement agile practices that support exist-

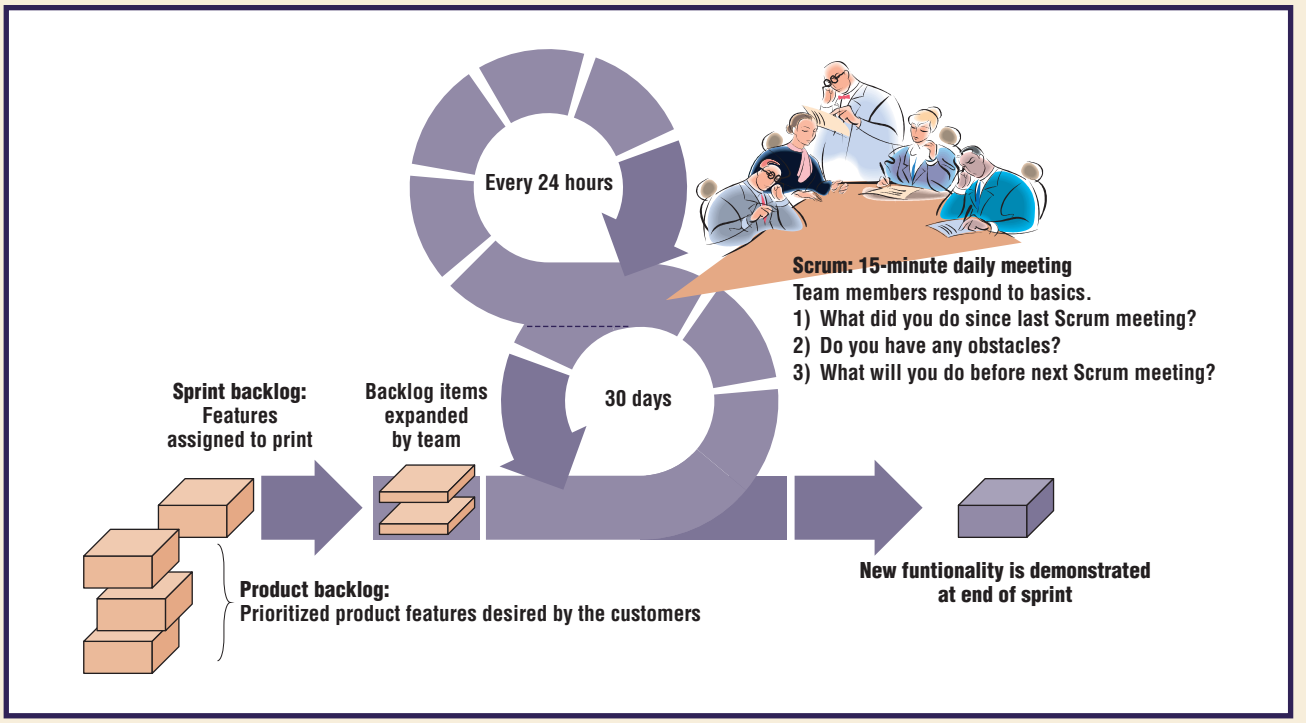

Figure A. The Scrum process: An example of an agile process flow. (Source: Advanced Development Methods, Inc., www.ControlChaos.com).

- Short cycles force you to focus on specific capabilities, can require more specific descriptions of functionality, and can identify misconceptions between the customer and developer earlier.
- Agile is more suited to emerging requirements and capability-based specifications than traditional top-down approaches.
- Agile provides rapid value to the customer, often delivering capability while traditional methods are still sorting out plans.
- Agile methods empower developers who might be suffering from the over-constraint of heavy processes.
- Agile practices aren't new, have been proven over time, and generally work as well as or better than some currently accepted practices.

ing processes or new organizational priorities. Practices that support existing processes might include prioritizing requirements (helpful in keeping on schedule when new requirements emerge) or test-first and continuous integration (helpful in finding problems earlier rather than later). An example of a new organizational priority that could benefit from agile methods is rapid application development. It can be aided by such practices as continuous integration, pair programming (pairs typically spend 20 percent more effort but 40 percent less calendar time), and timeboxing or its large-system counterpart, Schedule as Independent Variable.

Evaluating risks is the best overall approach to determining how much agility (or any attribute, for that matter) is enough. For each project decision, consider the risks of too much versus too little agility and the counterpart risks of doing too much planning and architecting. For example, our COCOMO II-based analysis[3] of how much architecting is enough indicates that for a 10 million source line of code project, devoting 30 to 40 percent of the project schedule to planning and architecting is best. On the other hand, a five percent investment is usually enough for a 10,000 SLOC product. The percentage is closer to zero if the product is

Categories of Barriers to Agile Processes

Participants at the 2004 USC-CSE Annual Research Review identified three categories of real and perceived barriers to implementing agile processes.

Nonproblems

- Quality assurance systems
- Agile inadequate for managing defects
- Refactoring is rework
- Agile is monolithic
- Quantitative management
- Extension/effectiveness of automatic testing to acceptance/system integration
- Perception that agile is extreme or a fad; not responsible
- Agile projects are unmanaged

Problems only in terms of size or scope

- Configuration management
- Earned value tools: Agile focuses on features and business value, traditional focuses on activities
- Stakeholder sign-off requirements
- Planning documentation
- Deployment, life cycle support (training): Long-term life cycle sustainment, decay rate of tacit knowledge
- Risk management
- Contracted/planned inch-pebble milestones
- Process QA/standard processes
- Process standards (IEEE, DoD, EIA)
- Designing for the battle, not the war

Significant issues

- Resource loading, slack, timekeeping, capital evaluation
- Required colocation, customer access
- Nonfunctional requirements
- Documentation
- Critical design reviews (milestones)
- Contractual and source selection issues
- Interfacing/integration with other methodologies/disciplines
- Predictability, perfect knowledge
- Statutory/regulatory constraints
- HR policies and processes
- System interface control
- Roles, responsibilities, and skills
- Agile work on legacy systems
- Formal requirements
- System engineering V-process model
- Maturity assessments
- Traditional engineering measurements
- Cost estimation

being developed on top of a mature architectural framework and middleware package.

As a final comment on this challenge, we believe that it represents a part of the larger issue of integrating advances in software development approaches into the traditional systems engineering approach. We address this larger issue in the "Opinion: Managing the Integration of Software Engineering and Systems Engineering" sidebar.

Business process conflicts

An often-overlooked difference between agile and traditional engineering processes is the way everyday business is conducted. Estimation, resource loading, and slack calculations can vary significantly. The level of uncertainty and ambiguity that exists in any evolutionary or iterative process, particularly in long-term estimates, is most likely higher with agile approaches. We're still caught in the certainty/ambiguity conundrum; our business processes and infrastructure require near-perfect predictions of difficult-to-estimate tasks rather than encouraging experimentation and evolution with knowable short-term results but accepting of long-term haziness.

Human resources

Organizations must learn to accommodate human-resource issues such as timekeeping, position descriptions, team-oriented versus individual rewards, and required skills. Agile development team members often cross the boundaries between standard development position descriptions and might require significantly more skills and experience to adequately perform. Paradoxically, HR departments and procedures frequently get in the way of empowering people to pursue nontraditional approaches that require organizations to revisit legally vetted and audited policies and procedures.

Progress measurement

Traditional contracts, milestones, and progress measurement techniques might be inadequate to support agile processes' rapid pace. Contracts and payments tend to be based on MIL-STD-1521 milestones like preliminary and critical design reviews, which have almost no meaning in an agile environment. Traditional earned-value processes are difficult if not impossible to apply to agile work because of work breakdown structure inadequacies and the flexibility timeboxing requires. In some cases, agile measurements, such as require-

ments burndown (the number of requirements or functions in the backlog list that have been accomplished) or story completion, have been successfully used as reasonable substitutes for more traditional measures.

Process standard ratings

One area of conflict for mature organizations will be in how agile processes will affect their ratings with respect to CMMI, ISO, or other process standards. We feel that agile is in line with much of the Level 5 concept of constantly adapting to improve performance. Unfortunately, most agile methods don't support the degree of documentation and infrastructure required for lower-level certification; it might, in fact, make agile methods less effective. It's possible that enlightened appraisers can find ways to include agile methods as alternative practices in many instances, although safety-critical areas or components requiring some form of certification call for rigorous appraisals.

Suggestions

You can address business process issues in numerous ways. Also, several research topics promise possibilities for solutions.

Address HR issues when you begin your pilot project so that you can test their impact on traditional processes. This might cause initial delays, but it can save considerable confusion and hard feelings if issues arise mid-project.

Apply throughput accounting rather than cost accounting in software development projects. David Anderson discusses this extensively in *Agile Management for Software Engineering*.[4]

Develop management and architectural practices for hybrid agile and plan-driven methods. Establish some standard characteristics and patterns that support separating stable components from those with evolving requirements or environments.

Investigate and update contracting practices to support agile concepts. Disburse payments upon delivery of running software or demonstration of progress rather than completion of artifacts or reviews. Develop contracting provisions and incentives for client satisfaction that support agile development.

The Nonproblems

A number of the barriers participants identified and considered during the USC workshop were deemed to be nonproblems. This indicated that either the perception was false (a myth) or that organizations could easily eliminate the issue without significant changes to either the agile or traditional ways of doing business.

Most of the myths stem from misunderstanding. One reason for this is a lack of clarity about the intent and actual operation of agile practices such as refactoring or pair programming. Often, engineers in particular react viscerally rather than cerebrally to ideas that are self-proclaimed as "extreme." These myths also often arise from encounters with counterfeit agile methods—for example, poor software developers claiming to use agile methods when in reality they're simply hacking.

Nonproblems that are easy to eliminate include those having to do with quality and management. Generally, organizations can deal with these by adopting broader definitions of traditional terms. Measurement is critical in agile methods, particularly in establishing development velocity values critical to timeboxing. However, organizations might need to reinterpret these measures to fit easily into existing processes. Quality and defect management are also intrinsic to most agile methods, but the activities don't necessarily match one to one with some of the traditional approaches. With a bit of imagination and good will, pair programming, test-driven development, and daily builds can be part of most traditional quality and acceptance programs. Management in agile methods is often defined as protecting the developers from the remainder of the organization or as coaching—certainly aspects of good management in any discipline.

Identify incompatible assumptions (model clashes) and compatible assumptions (synergies) between agile and traditional methods within your organizational processes. Work to eliminate as many clashes as possible while encouraging synergies.

Conduct empirical studies of which classes of change are more unpredictable and therefore suited for agile methods and which are more predictable and suitable for traditional plan-driven methods.

Research how to modify or reconceive legacy systems to enable and help agility-compatible re-engineering and maintenance, replacement, or extension.

Establish guidelines for safe and agility-compatible process maturity assessments. The SEI and other process improvement centers need to quickly address this. They have acknowledged agile methods, but haven't established a standard for lead assessors to handle them.

Opinion: Managing the Integration of Software Engineering and Systems Engineering

In the March 2000 issue of *Computer*, one of us (Barry Boehm) described the need to "change from slow, reactive, adversarial, separated software and system engineering processes to unified concurrent processes" to address "rapid development of dynamically changing software-intensive systems."[1] In the five years since that column, the software world has moved considerably toward more agile, concurrent processes, but the hardware-intensive systems engineering discipline has largely continued to use sequential document-driven processes.

Sequential approaches (based on the waterfall concepts of relatively complete system requirements allocated down to the hardware and software developers, who then provided complete configuration items back to integration and validation teams in a MIL-STD-1521 review-based development cycle) worked well in a stable environment where software was a small part of the system and software development was somewhat slower paced. In today's environment, where software and user interaction drives the majority of system functionality, requirements tend to emerge with system use and to evolve rapidly, and hardware is less expensive and also evolving, the idea of establishing system requirements and specifications without close interactivity with software is risky.

We're in a transition period where software engineering has aggressively embraced more iterative and concurrent methods. Unfortunately, these methods are causing significant issues for systems engineering, and only recently has the hardware-intensive mainstream of systems engineering recognized that although waterfall-type approaches are required for hardware production, planning, and preparation, they need to be synchronized with the software spirals. The disconnects are most visible in large network-centric systems-of-systems, where software is responsible for large percentages of functionality and infrastructure.

Various organizations and standards groups (including the Capability Maturity Model Integration) have made some efforts to modernize and integrate the two disciplines. However, they do not fully consider or mitigate software risks. For example, the attempt to develop systems-of-systems by integrating best-of-breed systems developed under total systems performance responsibility guidelines usually results in belated discovery of incompatible commercial off-the-shelf, GUI, and architectural decisions. Integrated product teams that produce unvalidated and poorly coordinated PowerPoint or UML architectural or high-level design solutions leave major risks to discover late in the development cycle. Or, consider the case of iterative development in large systems. The traditional disciplines generally establish increments based on functionality and focus on one build at a time, often without an overarching architecture. This runs a risk of suboptimized early builds that result in large amounts of breakage and rework in later increments where more stringent requirements are expected from crosscutting attributes such as safety, security, and scalability.

Essential critical success factors are beginning to emerge. First, expectations management is a key component of any evolving system. Agreeing to unrealistic schedules or budgets

People conflicts

People issues are by far the most critical in improving management of engineering and development personnel. Addressing them is vital to the adoption and integration of agile methods and practices into your processes. People issues are at the heart of the agile movement, and much of the paradigm change is aimed at empowering individuals by supporting reasonable goals, shorter feedback cycles, ownership, and flexibility.

Management attitudes

Migrating from traditional to agile management attitudes can be difficult. Large-scale management processes such as earned value and statistical process control evolved from a manufacturing paradigm and tend to cast employees as interchangeable parts. Managers also tend to associate employees with specific roles that might cause difficulty in the multitasking characteristics of agile team members. Project managers in most agile methods play two primary roles: protector and coach. They act as a barrier between the organization and the team to minimize unnecessary perturbation during a sprint or development cycle and provide experienced technical help when necessary. Many traditional managers also fill these functions, but agile methods focus on them.

Logistical issues

Some logistical issues directly affect people in agile environments. Agile teams must nearly always be colocated. The typical agile workspace requires pair-programming stations,

only to continuously revise them upward is a lose-lose position in the long run.

Second, systems and software engineering must proceed concurrently but in a controlled, coordinated manner. This coordination has been likened to an asynchronous machine of engineering activities that requires periodic synchronization. This synchronization can be formal or informal, but a mechanism must exist for defining reasonable outcomes for relatively short periods of time. It's critical to establish ways to adjust activities for risks and issues, analyzing downstream impacts and using the results to replan for the next set of outcomes.

Third, the development of systems that will evolve must be architecture driven. Prototypes and experimentation are crucial to defining and characterizing an architecture that provides multiple development teams sufficient guidance to support the concurrency and coordination we mentioned earlier. This is particularly true for systems whose components are outsourced, legacy, independently evolving, or built by subcontractors. The architecture should be solidly in hand before any contractual agreements are established to avoid unaffordable rework and incompatible components.

Compatibility of organizational structures and business processes is the fourth factor. The lines of communication among component developers, systems engineers, and architects must be conceptually high bandwidth and readily accessible. Integrated product teams must not be allowed to evolve into independent product teams. Management must not lose sight of the importance of teamwork, flexibility, agility, and broad vision in preventing stovepiping and local optimization. Contracts and incentive structures must reflect this understanding as well.

Finally, as always, managing *by* risks is more critical than the managing *of* risks. Risks exist at every level of the system hierarchy. Many of those risks surface rapidly and, if management doesn't identify and respond to them, they can quickly become serious program-threatening problems. Programs must actively seek out risks and discuss them across the development hierarchy. They must intentionally watch for cross-cutting risks. Mitigation plans and activities must undergo constant evaluation for effectiveness and be quickly changed if they're not performing as expected.

The time is ripe for software and systems engineers to compare notes and learn from each other. Software is further along in dealing with unpredictable requirements changes, while systems engineering has a long history of successfully integrating existing components. We believe we should move intentionally toward a common set of lifecycle definitions and processes that incorporate both disciplines' needs and capitalize on their strengths.

Reference

1. B. Boehm, "Unifying Software Engineering and Systems Engineering," *Computer*, vol. 33, no. 3, 2000, pp. 114–116.

walls for status charts and assignments, a layout that allows team members to easily converse to share information, and sufficient equipment to support continuous integration and regression testing. (To those of us of a certain age, this may be reminiscent of engineering bullpens similar to those in the HBO series *From the Earth to the Moon*—only with better tools.)

Handling successful pilots

The negative impacts of how organizations handle the success of pilot projects are often overlooked in reporting outcomes. During the USC workshop, Alistair Cockburn described his experience with an all too frequently observed response to a successful agile pilot: fire or promote the manager and/or split up the team. Of course, this does three things: it destroys team relationships, both technical and personal, it dilutes the knowledge gained and lessons learned, and it sends the message that trying new things might be hazardous to your career.

Change management

Change management experts often describe the organizational antibodies that begin to gather as soon as something new appears in the existing culture. Concerns of inadequacy or obsolescence surface, jealousy about assignments and business accoutrements is aroused, and defense mechanisms rapidly deploy. This can result in several destructive behaviors, including the cultural crucifixion of change agents or early adopters and the deliberate sabotage of

Clearly, agile concepts will continue to migrate into traditional organizations (and vice versa) through planned or clandestine vectors.

projects through direct or indirect methods.

Then there's the problem of dealing with employees who simply refuse to use new methods (we've identified them as Cockburn level-ones[3]). Having a nonplayer (or someone playing for the opposing side) can disable any team but is particularly damaging in an agile context. Agile teams rely heavily on trust and shared tacit knowledge to support pair programming and shared ownership. This also makes moot any efforts to measure the results.

People within the organization aren't the only ones affected by the introduction of agile methods. Stakeholders, particularly customers, might need to play significantly different roles. Many agile methods require (or at least strongly suggest) onsite customers, significant customer interaction and feedback, and customer input for acceptance testing. Attention to process matching and customer education is necessary to smooth the transition.

Suggestions

The following practices can help in addressing people issues.

Understand how communication occurs within development teams. This is key to incorporating agile practices and teams. You can find a good discussion in Alistair Cockburn's *Agile Software Development*.[5]

Educate stakeholders. Countering mythology through education is an ancient, honored tradition. Learn as much as you can and share it with customers, managers, and practitioners. Engaging speakers to discuss experiences or specific methodologies can help, but beware the possibly negative impact of true believers.

Translate agile and software issues into management and customer language. Engaging upper management and customers in fruitful discussion of software issues is often difficult because of the "eyes glaze over" response. When you discuss technical issues with not-as-technical-as-you people, remember to describe issues in terms that the audience can easily connect with.

Emphasize value. Software engineering has traditionally been value-neutral—every requirement, test case, object, or defect has been essentially equally important. Agile methods emphasize value in two ways. First, they negotiate and prioritize requirements so that expectations are managed and timeboxing can work. Second, they acknowledge the value of each team member, the team as an entity, and the products the team produces to the organization and the customer.

Pick good people and reward the results of pilot projects. You don't need to create a dream team, but definitely eliminate the level-ones in your pilots. Show your appreciation for the team's work, regardless of the outcome. The team members put their reputations on the line for the organization, leaving themselves vulnerable to the organizational antibodies. Don't minimize that effort.

Reorient reward systems to recognize both individual and team contributions.

We've identified some barriers to integrating agile and traditional methods as perceptual, not technical. However, we've also seen that many technical barriers do exist. We don't believe that these are insurmountable and are confident that organizations can overcome them with diligence, patience, and work. Clearly, agile concepts will continue to migrate into traditional organizations (and vice versa) through planned or clandestine vectors. Research is needed in several areas to provide new approaches and to harmonize the methods. For example, the Systems and Software Consortium is finalizing a document called *Disciplined Agility*, which describes an approach to implementing agile methods in high-maturity and process-compliant environments Most importantly, though, is the need to capture metrics data and lessons learned from your experiences. Data is critical to validate the integration activity's value and the return on investment. Lessons learned can support more rapid integration, eliminate the repetition of ineffective approaches and practices, and disseminate experience across the organization and throughout the community.

References

1. B.Boehm, "Anchoring the Software Process," *IEEE Software*, vol. 13, no. 4, 1996, pp. 73–82.
2. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
3. B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2004.
4. D. Anderson, *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*, Prentice Hall, 2003.
5. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2001.

About the Authors

Barry Boehm is the TRW Professor of Software Engineering and director of the Center for Software Engineering at the University of Southern California. His research interests include software process modeling, software requirements engineering, software architectures, software metrics and cost models, software engineering environments, and value-based software engineering. His contributions to the field include the Constructive Cost Model (COCOMO), the Spiral Model, and the Theory W (win-win) approach to software management and requirements determination. He is a fellow of the ACM, AIAA, the IEEE, and INCOSE and is a member of the US National Academy of Engineering. He received his PhD from UCLA in mathematics. Contact him at the Center for Software Eng., Univ. of Southern California, 941 W. 37th Place, SAL Room 337, Los Angeles, CA 90089-0781; boehm@cse.usc.edu.

Richard Turner is a director at the Systems and Software Consortium. His research interests include establishing empirical profiles for software development and acquisition best practices; employing spiral, risk-driven methods in large system-of-systems acquisitions; harmonizing software engineering, systems engineering, and acquisition life-cycle models; and developing ways to measure return on investment for software and systems engineering processes and process improvement. He was on the original author team for Capability Maturity Model Integration. He coauthored *Balancing Agility and Discipline: A Guide for the Perplexed* (Addison-Wesley, 2004), cowritten with Barry Boehm, and *CMMI Distilled* (Addison-Wesley, 2000, 2004). He received his DSc in engineering management from George Washington University. He's a member of the IEEE and INCOSE. Contact him at the Systems and Software Consortium, 2214 Rock Hill Rd., Herndon, VA 20170-4227; turner@systemsandsoftware.org.