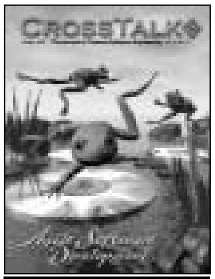
CrossTalk

October 2002 **The Journal of Defense Software Engineering** Vol. 15 No. 10



ON THE COVER
Cover Design by Kent Bingham.

CrossTalk

Sponsor	*Lt. Col. Glenn A. Palmer*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Bowers*
Associate Editor	*Chelene Fortier*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Kay Jensen*
Phone	(801) 586-0095
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk
CRSIP Online	www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 27.

Ogden ALC/MASE
7278 Fourth St.
Hill AFB, UT 84056-5205

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CrossTalk.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc. that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CrossTalk Editorial Department.

STSC Online Services: www.stsc.hill.af.mil
Call (801) 777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CrossTalk available free of charge.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

Agile Software Development Deserves An Open-Minded Approach

This month's CROSSTALK focuses on one of the potentially hot topics of debate in the software industry today – agile software development vs. disciplined, process-focused software development. There seems to be much confusion about what agile software development is and is not, and what agile implies. We hope that our selection of articles will provide some insight into agile software development methodologies and how they compare in application and use to the process improvement strategies embodied in capability maturity models that we have championed for many years. Perhaps these articles may even widen your understanding and perceptions of where agile development fits within today's software development challenges.

Our lead article by Jim Highsmith, *What Is Agile Software Development?* sets the stage for understanding agile software development. He discusses what he terms the *sweet spot problem* domain for agile approaches and then gives greater detail on the three dimensions he refers to as agile ecosystems: chaordic perspective, collaborative values, and barely sufficient methodology.

We next provide part one of a two-part article by Alistair Cockburn, *Learning From Agile Software Development – Part One*. Part one describes how agile and plan-driven teams make different trade-offs of money for information or for flexibility, and presents the first seven of 10 principles for tuning a project to meet various priorities, including cost, correctness, predictability, speed, and agility. Part two will be featured in the November issue of CROSSTALK.

In *Agile Methodologies and Process Discipline*, Mark C. Paulk addresses issues surrounding agile development as compared to rigorous software process improvement models such as the Capability Maturity Model® for Software (SW-CMM®). He summarizes and critiques the compatibility of agile methodologies with plan-driven methodologies as described by the SW-CMM, and concludes by making the point, "Perhaps the biggest challenge in dealing effectively with both agile and plan-driven methodologies is dealing with extremists in both camps who refuse to keep an open mind."

In his article, *Odyssey and Other Code Science Success Stories*, John Manzo tells of the successful delivery of an actual complex industrial automation application using an agile development methodology based on eXtreme Programming (XP). He includes some real-world insights from a developer's experience applying the development method, including a quantitative measure of XP's effectiveness since its inception.

Our supporting articles this month begin with *Integrating Systems and Software Engineering: What Can Large Organizations Learn From Small Start-Ups?* Author Paul E. McMahon explores variations in large and small engineering organizations and presents an alternative view of large projects that he claims may aid companies in their quest for more effective systems and software integration. He believes that using adaptive techniques and small informal teams inside large organizations can complement a formal organizational process focus that may already exist within a company.

Next, in *Highpoints From the Agile Software Development Forum*, Pamela Bowers summarizes the keynote talks from "Creating Competitive Advantage Through Agile Development Practices," a technology forum held at Westminster College in Salt Lake City in March. In *Agile Before Agile Was Cool,* Gordon Sleve describes a specific example of choice of an agile methodology for the development of a needed application, with quick turnaround to meet a customer's need. He notes that this was done long before agile methods were well known or popular and concludes by saying, "I see both methodologies coexisting and filling an important purpose: *to make our customers successful*."

We conclude this issue with two online articles: *Should You Be More Agile?* by Rich McCabe and Michael Polen, and *Agile Development: Weed or Wildflower?* by David Kane and Steve Ornburn. These authors believe that software projects, even in the defense community, can benefit from the techniques of agile development.

As noted in the lengthy references at the end of many of these articles, much is being written about agile software development at this time. We hope that this selection of thought-provoking articles will enable you to become more enlightened about the many perspectives of these new software development alternatives and will have a positive influence on opening your minds to new possibilities.

H Bruce Allgood

H. Bruce Allgood
Deputy Director, Computer Resources Support Improvement Program

What Is Agile Software Development?[1]

Jim Highsmith
Cutter Consortium

In the past two years, the ideas of "agile software development," which encompasses individual methodologies such as Crystal methods, eXtreme Programming, feature-driven development, and adaptive software development, are being increasingly applied and are causing considerable debate. This article attempts to answer the fundamental question on many people's minds: What is agile software development?

"Never do anything that is a waste of time – and be prepared to wage long, tedious wars over this principle," said Michael O'Connor, project manager at Trimble Navigation in Christchurch, New Zealand. This product group at Trimble is typical of the homegrown approach to agile software development methodologies.

While interest in agile methodologies has blossomed in the past two years, its roots go back more than a decade. Teams using early versions of Scrum, Dynamic Systems Development Methodology (DSDM), and adaptive software development (ASD) were delivering successful projects in the early- to mid-1990s.

This article attempts to answer the question, "What constitutes agile software development?" Because of the breadth of agile approaches and the people who practice them, this is not as easy a question to answer as one might expect. I will try to answer this question by first focusing on the sweet-spot problem domain for agile approaches. Then I will delve into the three dimensions that I refer to as *agile ecosystems*: barely sufficient methodology, collaborative values, and chaordic perspective. Finally, I will examine several of these agile ecosystems.

The Agile Problem Domain: Fitting the Process to the Project

All problems are different and require different strategies. While battlefield commanders plan extensively, they realize that plans are just a beginning; probing enemy defenses (creating change) and responding to enemy actions (responding to change) are more important. Battlefield commanders succeed by defeating the enemy (the mission), not conforming to a plan.

I cannot imagine a battlefield commander saying, "We lost the battle, but by golly, we were successful because we followed our plan to the letter." Battlefields are messy, turbulent, uncertain, and full of change. No battlefield commander would say, "If we just plan this battle long and hard enough, and put repeatable processes in place, we can eliminate change early in the battle and not have to deal with it later on."

A growing number of software projects operate in the equivalent of a battle zone – they are extreme projects. This is where agile approaches shine. Project teams operating in this zone attempt to utilize leading or bleeding-edge technologies, respond to erratic requirements changes, and deliver products quickly. Projects may have a relatively clear mission, but the specific requirements can be volatile and evolving as customers and development teams alike explore the unknown. These projects, which I call high-exploration factor projects, do not succumb to rigorous, plan-driven methods.

"Projects may have a relatively clear mission, but the specific requirements can be volatile and evolving as customers and development teams alike explore the unknown."

The critical issues with high-exploration factor projects are as follows: first, identifying them; second, managing them in a different way; and third, measuring their success differently. Just as winning is the primary measure of success for a battlefield commander, delivering customer value (however the customer defines it) measures success for the agile project manager. Conformance to plan has little meaning in either case. If we want to be agile, we have to reward agility.

There is, however, a critical difference between managing a battle and managing warehouse logistics. Battlefields are managed by constant monitoring of conditions and rapid course alterations – by *empirical processes*. Adapting to changing conditions is vital. Conversely, managing warehouse logistics is a process that can be described by calculations involving materials on hand, deliveries, and shipments; managing can be described as a *defined process*, one that involves a relatively high degree of predictability and algorithmic precision. Many manufacturing plants operate as defined processes.

The concepts and assumptions behind empirical and defined processes are fundamentally and irreconcilably different. The practices of agile software development – short iterations, continuous testing, self-organizing teams, constant collaboration (daily integration meetings and pair programming for example), and frequent re-planning based on current reality (rather than six-month- old plans) – are all geared to the understanding of software development as an empirical process.

On the other hand, the fundamental basis of the Capability Maturity Model® (CMM®) and CMM Integration℠ (CMMI℠) is a belief in software development as a defined process. As such, tasks can be defined in detail, *algorithms* can be defined, results can be accurately measured, and measured variations can be used to refine the processes until they are repeatable within very close tolerances.

For projects with any degree of exploration at all, agile developers just do not believe these assumptions are valid. This is a deep, fundamental divide – and not one that can be reconciled to some comfortable middle ground. It is part of having a chaordic (meaning a combination of chaos and order as coined by Dee Hock, founder and former CEO of Visa

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.
℠ CMM Integration and CMMI are service marks of Carnegie Mellon University.

International) perspective on the world as described in the next section.

While agile practices – refactoring, iterative feature-driven cycles, customer focus groups – are applicable to nearly any project, I believe the agile *sweet spot* is this *exploratory projects* problem category. The more volatile the requirements and the more experimental the technology, the more agile approaches improve the odds of success.

The Agile Ecosystem: Chaordic, Collaborative, and Streamlined

The agile movement covers a broader set of issues than the word *methodology* connotes, so I use the word *ecosystem* to include the three characteristics that define agile development: a chaordic perspective, collaborative values and principles, and barely sufficient methodology. A chaordic perspective arises from recognition and acceptance of increasing levels of unpredictability in our turbulent economy.

Two concrete ramifications of trying to manage in an unpredictable environment are that while goals are achievable, project details are often unpredictable, and that the foundation of many process-driven approaches (the goal of repeatable processes) is unattainable. In company after company, I have found successful projects that met the customer's vision, but in the end, looked nothing like the original plan.

Furthermore, truly repeatable processes would be almost mechanical in nature, and no mechanical process could possibly react to the infinite variety of variations that software projects encounter. The reason many projects attain their goals has little to do with repeatable processes and much to do with the skill and adaptability of the people who are working on the project.

Hock's chaordic style is similar to what I call leadership-collaboration or adaptive management, which is about creating an environment with the *requisite variety* to meet the challenge of extreme projects, particularly the challenge of high change.

Agile managers understand that demanding certainty in the face of uncertainty is dysfunctional. They set goals and constraints that provide boundaries within which creativity and innovation can flourish. They are macromanagers rather than micromanagers[2].

While an entrepreneurial Silicon Valley company has one culture, a space shuttle avionics team has another. One of the biggest problems in implementing software development methodologies over the years has been the attempted mismatch of culture and methodology. Rather than recognizing the inherent differences between people, project teams, and organizations, we denigrate those who have different cultures by labeling them unprofessional, immature, or undisciplined. Or conversely, we label them bureaucratic, rigid, and closed-minded. For example, you can call a well-oiled extreme programming team a lot of things, but after watching them practice test-first development, pair programming, constant refactoring, and simple design, the last thing you can call them is undisciplined, immature, or unprofessional.

The second characteristic of agile development is *collaborative* values and principles. Agile and rigorous organizations view people, and how to improve their performance, differently. Rigorous methodologies are designed to standardize people to the process, while agile processes are designed to capitalize on each individual's and each team's unique strengths: they adapt the process to the people[3].

Agile organizations focus on building individual skills and on fostering a high degree of interaction among team members and the project's customers. Agilists believe that with today's complex projects, understanding comes more from face-to-face interaction than from documentation. Agilists do not believe that a reliance on heavy processes makes up for lack of skill, talent, and knowledge.

The third aspect of agile ecosystems is the concept of a *barely sufficient* methodology that attempts to answer the question of how much structure is enough. To be agile, one must balance flexibility and structure, and barely sufficient does not mean insufficient. Bare sufficiency reduces costs through streamlining but even more importantly, it incorporates the chaordic perspective that creativity and innovation occur in a slightly messy environment, not a primarily *structured* one. Too many organizations operate on the unspoken assumption that "if a little process is good, then lots of process will be better."

Concepts drive action and behavior. Software inspections, or any other software engineering technique, will be implemented differently depending upon one's underlying conceptual framework. The underlying conceptual frameworks behind agile development and the CMM are different and will therefore drive organizations to different behaviors. The really important questions are about to what kind of core capabilities one's conceptual foundation leads. These are not always easy questions to answer, and organizations will have different answers for different stages in their evolution and for different projects in their portfolio.

An Agile Case Story

Jeff De Luca, project director of Nebulon, an information technology consulting firm in Melbourne, Australia, offers an example of an agile methodology's success using feature-driven development (FDD). De Luca's project was a complex commercial lending application for a large Singapore bank utilizing 50 people for 15 months (after a short initialization period). I tracked De Luca down looking for an FDD case story for my book, and subsequently spent several hours on the phone and exchanged many e-mails with him.

Previously, the Singapore lending project had been a colossal failure. Prior to De Luca's involvement in the project, a large, well-known systems integration firm had spent two years working on the project and finally declared it undoable. Its deliverables included the following: 3,500 pages of use cases, an object model with hundreds of classes, thousands of attributes (but no methods), and, of course, no code. The project – an extensive commercial, corporate, and consumer lending system – incorporated a broad range of lending instruments (from credit cards to large multi-bank corporate loans) and a breadth of lending functions (from prospecting to implementation to back-office monitoring). "The scope was really too big," said De Luca.

FDD arose, in name at least, in 1997-98 when Nebulon took over the Singapore project. De Luca had been using a streamlined, light-process framework for many years. Peter Coad, who was brought in to develop the object model for the project, had been advocating very granular, feature-oriented development but had not embedded it in any particular process model. These two threads came together on this project to fashion what was dubbed FDD.

Less than two months into the *new* project, De Luca's team was producing demonstrable features for the client. The team spent about a month working on the overall object model (the original model and what De Luca refers to as the previous team's useless cases were trashed). They spent another couple of weeks working on the feature decomposition and short iteration planning. Finally, to demonstrate the project's viability to a once-burned and skeptical client, De Luca and his team built a portion of the relationship management application as a proof of concept. From that point on, with about four months elapsed, they staffed to 50 people and delivered approximately 2,000 features in 15 months. The project was completed significantly under budget, and the client, the CEO of the bank, wrote a glowing letter about the success of the project.

While talking with De Luca, a couple of things struck me about this project. Certainly the FDD process contributed to the project's success, but when I asked De Luca what made the FDD successful, his first response was that the overriding assumption behind the FDD is that it embraces and accepts software development as a decidedly human activity. The key, he said, is having good people – good domain experts, good developers, good chief programmers. No process makes up for a lack of talent and skill.

My guess is that even if the first vendor's staff had used FDD as a process model, they would not have been successful because they just did not have the appropriate level of technical and project management talent. However, had they been using a FDD-like agile process, their inability to complete the project might have surfaced in less than two years. This is a clear example of why working code is the ultimate arbiter of real progress. In the end, thousands of use cases and hundreds of object model elements did not prove real progress.

A Sampler of Agile Approaches

There are a growing number of agile *methodologies*, or agile software development ecosystems (ASDEs), as I prefer to label them, and a number of agile practices such as Scott Ambler's agile modeling [1]. The core set of these includes lean development (LD), ASD, Scrum, eXtreme Programming (XP), Crystal methods, FDD, and DSDM. Authors of all of these approaches (except LD) participated in writing the Agile Software Development Manifesto [2] and so its principles form a common bond among practitioners of these approaches. While individual practices are varied, they fall into six general categories:

- Visioning. A good visioning practice helps assure that agile projects remain focused on key business values (for example, ASD's product visioning session).
- Project initiation. A project's overall scope, objectives, constraints, clients, risks, etc. should be briefly documented (for example, ASD's one-page project data sheet).
- Short, iterative, feature-driven, timeboxed development cycles. Exploration should be done in definitive, customer-relevant chunks (for example, FDD's feature planning).
- Constant feedback. Exploratory processes require constant feedback to stay on track (for example, Scrum's short daily meetings and XP's pair programming).
- Customer involvement. Focusing on business value requires constant interaction between customers and developers (for example, DSDM's facilitated workshops and ASD's customer focus groups).
- Technical excellence. Creating and maintaining a technically excellent product makes a major contribution to creating business value today and in the future (for example, XP's refactoring).

Some agile approaches focus more heavily on project management and collaboration practices (ASD, Scrum, and DSDM), while others such as XP focus on software development practices, although all the approaches touch the six key practice areas. The rest of this section delves into four of these approaches, illustrating different aspects of each.

Lean Development

The most strategy-oriented ASDE is also the least known: ITABHI, Inc. President Bob Charette's LD was derived from the principles of lean production used during the restructuring of the Japanese automobile manufacturing industry in the 1980s. In LD, Charette extends traditional methodology's view of change from a risk of loss to be controlled with restrictive management practices to a view of change as producing *opportunities* to be pursued using *risk entrepreneurship*. LD has been used successfully on a number of large telecommunications projects in Europe.

The goals of LD are completion in one-third the time, within one-third the budget, and with one-third the defect rate. While most other ASDEs are tactical in nature, Charette thinks that the major changes required to become agile must be initiated from the top of the organization. Organizational strategy becomes the context within which agile processes can operate effectively. Without this strategic piece, agile development – as all those who have tried to implement ASDEs in organizations can testify – is shunted aside by the organizational forces that seek equilibrium.

LD is the *operational* piece in a three-tiered approach[4] that leads to change-tolerant businesses. It provides a delivery mechanism for implementing risk entrepreneurship. The key in business, according to Charette, is that the opportunity for competitive advantage comes from being more agile than the competitors in one's market.

LD's risk entrepreneurship enables companies to turn risk into opportunity. Charette defines change tolerance as "the ability of an organization to continue to operate effectively in the face of high market turbulence." A change-tolerant business not only responds to changes in the marketplace, but also causes changes that keep competitors off balance. "Most software systems are not agile, but fragile," said Charette. "Furthermore, they act as brakes on competitiveness." Every business must deal with change by building a change-tolerant organization that can impose change on competitors.

Charettes's work sends three key messages to agile developers and business stakeholders in information technology. First, the wide adoption of ASDEs will require strategic selling at senior levels within organizations. Second, the strategic message that will sell ASDEs is the ability to pluck opportunity from fast-moving, high-risk *exploration* situations. And third, proponents of ASDEs must understand and communicate to their customers the risks associated with agile approaches and, therefore, the situations in which they are and are not appropriate.

LD is as much a management challenge as a set of practices. Charette said, "You have to set the bar high enough to force rethinking traditional practices. LD initiatives focus on accelerating the speed of delivering software applications, but not at the expense of higher cost or defect rates. These three goals need to be achieved concurrently, or it isn't LD."

Adaptive Software Development

In 1992, I started working on a short interval, iterative, rapid application development process that evolved into ASD. The original process, developed in conjunction with colleague Sam Bayer, was used on projects in companies from Wall Street brokerage houses to airlines to telecommunications firms. During the next several years, Sam and I (together and separately) successfully delivered more than 100 projects using these practices. During the early to mid-1990s, I also worked with software companies that were using similar techniques on very large projects.

In the mid-1990s, my interest in complex adaptive systems began to add a conceptual background to the team aspects of the practices and was the catalyst for the name change to ASD. Complexity theory helps us understand unpredictability and that our inability to predict does not imply an inability to make progress. ASD works with change rather than fighting against it. In order to thrive in turbulent environments, we must have practices that embrace and respond to change – practices that are adaptable. Even more importantly, we need people, teams, and organizations that are adaptable and agile.

The practices of ASD are driven by a belief in continuous adaptation – a different philosophy and a different life cycle – geared to accepting continuous change as the norm. In ASD, the static plan-design-build life cycle is replaced by a dynamic speculate-collaborate-learn life cycle. It is a life cycle dedicated to continuous learning and oriented to change, re-evaluation, peering into an uncertain future, and intense collaboration among developers, management, and customers.

A Change-Oriented Life Cycle

A waterfall development life cycle, based on an assumption of a relatively stable business environment, becomes overwhelmed by high change. Planning is one of the most difficult concepts for engineers and managers to re-examine. For those raised on the science of reductionism (reducing everything to its component parts) and the near-religious belief that careful planning followed by rigorous engineering execution produces the desired results (we are in control), the idea that there is no way to "do it right the first time" remains foreign. The word plan, when used in most organizations, indicates a reasonably high degree of certainty about the desired result. The implicit and explicit goal of *conformance to plan* restricts a manager's ability to steer the project in innovative directions.

Speculation, the first conceptual concept, gives us room to explore, to make clear the realization that we are unsure and to deviate from plans without fear. It does not mean that planning is obsolete, just that planning is acknowledgeably tenuous. It means we have to keep delivery iterations short and encourage iteration. A team that speculates does not abandon planning; it acknowledges the reality of uncertainty. Speculation recognizes the uncertain nature of complex problems and encourages exploration and experimentation. We can finally admit that we do not know everything.

The second conceptual component of ASD is collaboration. Complex applications are not built; they evolve. Complex applications require that a large volume of information is collected, analyzed, and applied to the problem – a much larger volume than any individual can handle by himself or herself. Although there is always room for improvement, most software developers are reasonably proficient in analysis, programming, testing, and similar skills. But turbulent environments are defined in part by high rates of information flow and diverse knowledge requirements. Building an e-commerce site requires greater diversity of both technology and business knowledge than the typical project of five to 10 years ago. In this high-information-flow environment, in which one person or small group cannot possibly *know it all*, collaboration skills (the ability to work jointly to produce results, share knowledge, or make decisions) are paramount.

"A change-tolerant business not only responds to changes in the marketplace, but also causes changes that keep competitors off balance."

Once we admit to ourselves that we are fallible, then learning practices – the *learn* part of the life cycle – becomes vital for success. Learning is the third component in the speculate-collaborate-learn life cycle. We have to test our knowledge constantly, using practices like project retrospectives and customer focus groups. Furthermore, reviews should be done after each iteration rather than waiting until the end of the project.

An ASD life cycle has six basic characteristics: mission-focused, feature-based, iterative, time-boxed, risk-driven, and change-tolerant. For many projects, the requirements may be fuzzy in the beginning, but the overall mission that guides the team is well articulated. A mission provides boundaries rather than a fixed destination. Without a good mission and a constant mission refinement practice, iterative life cycles become oscillating life cycles – swinging back and forth with no progress.

The ASD life cycle focuses on results, not tasks, and the results are identified as application features. Features are the customer functionality that is to be developed during iteration.

The practice of time boxing, or setting fixed delivery times for iterations and projects, has been abused by many who use time deadlines incorrectly. Time deadlines used to bludgeon staff into long hours or to cut corners on quality are a form of tyranny; they undermine a collaborative environment. It took several years of managing ASD projects before I realized that time boxing was minimally about time – it was really about focusing and forcing hard trade-off decisions. In an uncertain environment in which change rates are high, there needs to be a periodic forcing function to get work finished.

As in Barry Boehm's spiral development model [3], analyzing the critical risks drives the plans for adaptive iterations. ASD is also change-tolerant, not viewing change as a problem but seeing the ability to incorporate change as a competitive advantage.

eXtreme Programming

XP, to most aficionados, was developed by Kent Beck, Ward Cunningham, and Ron Jeffries and has, to date, clearly garnered the most interest of any of the agile approaches. XP preaches the values of community, simplicity, feedback, and courage and is defined, at least in part, by its 12 practices: the planning game, small releases, metaphor, simple design, refactoring, test-first development, pair programming, collective ownership, continuous integration, 40-hour week, on-site customer, and coding standards.

There has been so much written about XP's practices that another rehash seems less important than discussing XP's impact on software development. The interest in XP generally comes from the bottom up, from developers and testers tired of burdensome processes, documentation, metrics, and formality. These individuals are not abandoning discipline, but excessive formality that is often mistaken for discipline. They are finding new ways to deliver high-quality software faster and more flexibly.

XP and other agile approaches are forcing organizations to re-examine whether their processes are adding any value to their organizations. Well over 400 individuals have signed the Agile Software Development Manifesto Web page, available at: <www.agilealliance.com>. These individuals reaffirm their desire to deliver high-quality software without the burdens of bureaucracy.

Other important contributions of XP proponents are their views on reducing the cost of change during a software's life and their emphasis on technical excellence through refactoring and test-first development. XP provides a *system* of dynamic practices, whose integrity as a holistic unit has been proven.

Some people think *extreme* is too extreme, that XP would be more appealing with a more moderate name. I don't think many people would get excited about a book on *moderate programming*. New markets, new technologies, new ideas aren't forged from moderation, but from radically different ideas and the courage to challenge the status quo. XP has led the way.

Dynamic Systems Development Method

The DSDM was developed in the United Kingdom in the mid-1990s. It is an outgrowth of, and extension to, rapid application development practices. The DSDM boasts the best-supported training and documentation of any ASDE, at least in Europe.

Each of the major phases of the DSDM development process – functional model iteration, design-and-build iteration, and implementation – are themselves iterative processes. The DSDM's use of three interactive iterative models and time boxes can be used to construct very flexible project plans.

The functional model iteration is a process of gathering and prototyping functional requirements based on an initial list of prioritized requirements. Nonfunctional requirements are also specified during this phase. The design-and-build iteration refines the prototypes to meet all requirements (functional and nonfunctional) and engineers the software to meet those requirements. One set of business functions (features) may go through both functional model and design-and-build iterations during a time box, and then another set of features goes through the same processes in a second time box. Implementation deploys the system into a user's environment.

The DSDM also addresses other issues common to ASDEs. First, it explicitly states the difference between the DSDM and traditional methodologies with respect to flexible requirements. The traditional view, according to the DSDM manual, is that functionality stays relatively fixed (after it is established in the original requirements specifications), while time and resources are allowed to vary. The DSDM reverses this viewpoint, allowing the functionality to vary over the life of the project as new things are learned. However, while functionality is allowed to vary, control is maintained by using time boxes.

The DSDM also addresses the issues of documentation – or lack thereof – a constant criticism of ASDEs. Because one of the principles of the DSDM relates to the importance of collaboration, it uses prototypes rather than lengthy documents to capture information. The DSDM recommends only 15 work products from its five major development phases, and several of these are prototypes. There is an interesting comment in the DSDM white paper on contracting:

> The mere presence of a detailed specification may act to the detriment of cooperation between the parties, encouraging both parties to hide behind the specification rather than seeking mutual beneficial solutions. [4]

With respect to work products, the DSDM, unlike rigorous methodologies, does not offer detailed documentation formats for its 15 defined work products. Instead, the DSDM work product guidelines offer a brief description, a listing of the purposes, and a half-dozen or so quality criteria questions for each work product.

Another area that the DSDM focuses on is establishing and managing the proper culture for a project. The manual describes, for example, the different emphasis of project managers and points out how difficult the transition can be for project managers steeped in traditional approaches. A passage from the DSDM manual illustrates this point:

> A traditional project manager will normally focus on agreeing a detailed contract with customers about the totality of the system to be delivered along with the costs and time scales. In a DSDM project, the project manager is focused on setting up a collaborative relationship with the customers. [4]

The focal point for a DSDM project manager shifts from the traditional emphasis on tasks and schedules to sustaining progress, getting agreement on requirement priorities, managing customer relationships, and supporting the team culture and motivation.

The Future of Agile Development

There are fundamental shifts driving economies, the structure of products that we build, and the nature of the processes we use to build products. "These changes in products, technologies, firms, and markets are not a passing phenomenon," according to Carliss Baldwin, Harvard Business School professor and Kim Clark, dean of the Harvard Business School faculty.

> These fundamental changes driven by powerful forces deep in the economic system, forces which moreover have been at work for many years ... we must be prepared to dig deep, for the forces that matter are rooted in the very nature of things, and in the processes used to create them. [5]

In the foreword to "Planning eXtreme Programming," Tom DeMarco makes the analogy between military history and software development as each swing from the relative advantages of armor to those of mobility. DeMarco says:

> In the field of IT, we are just emerging from a time in which armor (process) has been king. And now we are moving into a time when only mobility matters. [6]

Agile development is not defined by a small set of practices and techniques. Agile development defines a strategic capability, a capability to create and respond to change, a capability to balance flexibility and structure, a capability to draw creativity and innovation out of a development team, and a capability to lead organizations through turbulence and uncertainty.

Agile development does not abandon structure, but attempts to balance flexibility and structure – trying to figure out that delicate balance between chaos and rigidity. The greater the uncertainty, the faster the pace of change, and the lower the probability that success will be found in structure. Plan-driven methodologies have a definite place for some problem domains just as individual practices (configuration management for example) have a definite place in the most agile of software development projects. In a less volatile era, rigorous processes were applicable for a wide range of projects. In an era in which traditional management styles dominated, plan-driven software development approaches fit well.

But as Bob Dylan sang, "Times, they are a-changin'." Volatility and uncertainty increasingly defines today's business, and even today's military environment. Talented technical people want to work in an organization in which they have more control over how they work and how they interact with peers, customers, and management. Problems are changing, people are changing, and ideas are changing. While there are still opportunities for plan-driven style development and management, I believe growth lies in being agile and flexible.

Throughout the last three years, I have used agile methods with project teams in India, Canada, Norway, the United States, New Zealand, Poland, and Australia. Companies in these countries are struggling with exploratory projects that run the gamut, including an e-commerce infrastructure product, a clinical drug-trial monitoring application, 300,000 lines of embedded C code in a new cell-phone chip, a worldwide financial system product, a myriad of internal IT applications, the complete business system for a dot-com start-up (that is still in business), and an oil-field geophysical data gathering and analysis system.

These companies want to be more agile: They want to create change for their competitors and respond quickly to market conditions. They plan, but they are not blinded by those plans. They focus on delivering customer value, not adding up how many processes they have in place. They document, but they do not get lost in piles of paper. They rough out blueprints (models), but they concentrate on creating working software. They focus on individuals and their skills and on the intense interaction of development team members among themselves and with customers and management. They deliver results in a turbulent, messy, ever-changing, ever-exciting marketplace.◆

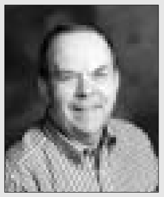
References

1. Ambler, Scott. Agile Modeling. New York: John Wiley, 2002.
2. AgileAlliance. "Agile Software Development Manifesto." 13 Feb. 2001 <www.agilemanifesto.org>.
3. Boehm, Barry. "A Spiral Model of Software Development Enhancement." IEEE Computer May 1988.
4. DSDM Consortium. Dynamic Systems Development Method. Version 3. United Kingdom <www.dsdm.org>.
5. Baldwin, Carliss Y., and Kim B. Clark. Design Rules – Vol. 1: The Power of Modularity. Cambridge: The MIT Press, 2000.
6. Beck, Kent, and Martin Fowler. Planning eXtreme Programming. Boston: Addison-Wesley, 2001.

Notes

1. This article is adapted from Jim Highsmith's book "Agile Software Development Ecosystems." Addison-Wesley, 2002. (Article quotes and examples taken from this book.)
2. For additional information, see James A. Highsmith. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York: Dorset House, 2000.
3. For extensive research in this area, see Buckingham, Marcus, and Curt Coffman. First, Break All the Rules: What the World's Greatest Managers Do Differently. New York: Simon & Schuster, 1999, and Buckingham, Marcus and Donald O. Clifton. Now, Discover Your Strengths. New York: Simon & Schuster, 2001.
4. The three tiers are Risk Leadership, Risk Entrepreneurship, and Lean Development.

About the Author



Jim Highsmith is director of the Cutter Consortium's Agile Project Management Practice, author of "Agile Software Development Ecosystems (2002)" and "Adaptive Software Development: A Collaborative Approach to Managing Complex Systems (2000)," and winner of the 2000 Jolt Award. He has more than 30 years experience as a consultant, software developer, manager, and writer. In the last 10 years, he has worked with information technology organizations, industrial product companies, and software companies in the United States, Europe, Canada, South Africa, Australia, Japan, India, and New Zealand to help them adapt to the accelerated pace of development in increasingly complex, uncertain environments.

Cutter Consortium
2288 North Coulter Drive
Flagstaff, AZ 86004
Phone: (781) 648-8700
E-mail: jim@jimhighsmith.com

Coming Events

October 14-16
20th Annual Pacific Northwest Software Quality Conference
THE PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE
Portland, OR
www.pnsqc.org

November 3-6
3rd Annual Amplifying Your Effectiveness (AYE) Conference 2002
Phoenix, AZ
www.ayeconference.com

November 4-8
Software Testing Analysis and Review Conference
Anaheim, CA
www.sqe.com/starwest

November 11-14
National Defense Industrial Association
Denver, CO
www.ndia.org

November 18-21
International Conference on Software Process Improvement
Washington, DC
www.software-process-institute.com

February 24-27, 2003
Software Engineering Process Group Conference
SEPG 2003
Boston, MA
www.sei.cmu.edu/sepg/

April 28-May 1, 2003
Software Technology Conference 2003
STC
Salt Lake City, UT
www.stc-online.org

May 3-10, 2003
International Conference on Software Engineering
Portland, OR
www.icse-conferences.org/2003

Learning From Agile Software Development – Part One

Alistair Cockburn
Humans and Technology

This two-part article compares agile, plan-driven, and cost-sensitive software development approaches based on a set of project organization principles, extracting from them ideas for pulling agile techniques into cost- and plan-driven projects. Part one describes how agile and plan-driven teams make different trade-offs of money for information or for flexibility, and presents the first seven of 10 principles for tuning a project to meet various priorities, including cost, correctness, predictability, speed, and agility. Part two, which will run in the November issue of CROSSTALK*, will present the last three principles, then pull the material together for actions that plan-driven and cost-sensitive project teams can use to improve their strategies and hedge against surprises.*

Being *agile* is a declaration of prioritizing for project maneuverability with respect to shifting requirements, shifting technology, and a shifting understanding of the situation. Other priorities that might override agility include predictability, cost, schedule, process-accreditation, or use of specific tools.

Most managers run a portfolio of projects having a mix of those priorities. They need to prioritize agility, predictability, and cost sensitivity in varying amounts and therefore need to mix strategies. This article focuses on borrowing ideas from the agile suite to fit the needs of plan-driven and cost-sensitive programs.

Our industry now has enough information to sensibly discuss such blending. The agile voices have been heard [1, 2, 3, 4, 5, 6, 7], the engineering voices have been heard [8, 9, 10], two articles in this issue [11, 12] illustrate the differences in world view, and some authors have discussed the question of their coexistence and principles underlying successful development strategies [3, 8, 13].

Buy Information or Flexibility

Many project strategies revolve around spending money for either information or flexibility [3, 14].

In a *money-for-information* (MFI) proposition, the team can choose to expend resources now to gain information earlier. If the information is not considered valuable enough, the resources are applied to other work. The question is how much the team is willing to expend in exchange for that information.

In a *money-for-flexibility* (MFF) proposition, the team may opt to expend resources to preserve later flexibility. If the future is quite certain, the resources are better spent pursuing the most probable outcome, or on MFI issues.

Different project strategies are made by deciding which issues are predictable, unpredictable but resolvable, or unresolvable, deciding which of those are MFI or MFF propositions, and how best to allocate resources for each.

Predictable issues can be investigated using breakdown techniques. Such an issue might be creating a schedule for work similar to that successfully performed in the past.

Unpredictable but resolvable issues can be investigated through study techniques such as prototypes and simulators. Such issues include system performance limits. These are also MFI propositions. Agile and plan-driven teams are likely to use similar strategies for these issues as part of basic project risk management.

"This is a MFI [money-for-information] situation: It is worth spending a lot of money now to discover ... where those next defects are located."

Unresolvable issues tend to be sociological, such as which upcoming standard will gain market acceptance, or how long key employees will stay around. These issues cannot be resolved in advance, and so are not MFI propositions, but are MFF propositions. Agile and plan-driven teams are intrinsically likely to use different strategies for these issues. Agile teams will set up to absorb these changes, while plan-driven project teams must, by definition, create plans for them.

Teams will differ on which issues are resolvable, and how much money should be spent in advance on predictable issues. A plan-driven team is more likely to decide that creating the project plan is basically a predictable issue, and that a good MFI strategy is to spend resources early to make those predictions.

In contrast, an agile team might decide that the project plan is fundamentally unresolvable past a very simple approximation. There being no effective MFI strategy, it adopts an MFF approach, making an approximate plan early and allocating resources for regular re-planning over the course of the project.

Both agile and plan-driven developers might agree that the question of system performance under load is an important MFI issue, and so both might agree to spend money early to build a simple system simulator and load generator to stress-test the design.

They are likely to spend money differently on design issues. The plan-driven team, viewing it as a sensible MFI proposition, will spend money early to reduce uncertainty about the future of the design. Agile teams are more likely to view design as either being inexpensive to change (a poor MFI candidate) or unresolvable (making it an MFF proposition). They are therefore more likely to choose a design early and allocate money to adjust it over time. This difference on design issues is fundamental, since the two groups view the matter from different decision arenas.

Ten Principles

The following 10 principles have shown themselves useful in setting up and running projects. Most of these are known in the literature [3, 4, 8, 21]. My phrasing of them may be slightly different.

1. Different projects need different methodology trade-offs.
2. A little methodology does a lot of good; after that, weight is costly.
3. Larger teams need more communication elements.
4. Projects dealing with greater potential damage need more validation elements.
5. Formality, process, and documentation are not substitutes for discipline, skill, and understanding.

6. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
7. Increased communication and feedback reduces the need for intermediate work products.
8. Concurrent and serial development exchange development cost for speed and flexibility.
9. Efficiency is expendable in non-bottleneck activities.
10. Sweet spots speed development.

The first seven principles are discussed in this article, the last three will be addressed in part two.

1. Different Projects Need Different Methodology Trade-offs

This should be obvious, but it seems to need re-stating at frequent intervals [15, 16, 17, 18].

Figure 1, adapted from Boehm and Port [8], shows one particular aspect of these differences. In this figure, the two diminishing curves show the potential damage to a project from not investing enough time and effort in planning. The two rising curves show the potential damage to the project from spending too much time and effort in planning.

The lines crossing on the left indicate a project for which potential damage is relatively low with under-planning, and relatively high with over-planning. Much commercial software, including Web services fall into this category. The lines crossing on the right indicate a project for which potential damage is relatively high with under-planning, and for which much more planning would have to be done before damage would accrue from delays due to planning. Safety-critical software projects fall into this category.

The curves should make it clear that when there is risk associated with taking a slow, deliberate approach to planning, then agile techniques are more appropriate. When there is risk associated with skipping planning or making mistakes with the plan, then a plan-driven approach is more appropriate. The curves illustrate clearly the home territory of each.

Figure 2 shows a different characterization of project differences [3]. The hori-zontal axis captures the number of people needing to be coordinated, rising from one on the left to 1,000 on the right. The idea is that projects need more coordina-tion elements to their methodology as the number of people increases.

The vertical axis captures the potential damage caused by undetected defects in the system, from loss of comfort to loss of life. The idea is that projects need more validation elements as the potential damage increases.

Each box in the grid identifies a set of projects that might plausibly use the same combination of coordination and validation policies. The label in the box indicates the maximum damage and coordination load common to those projects (thus, D40 refers to projects with 20-40 people and potential loss of discretionary monies). Projects landing in different boxes should use different policies.

The different planes capture the idea that projects run to different priorities, some prioritizing for productivity, some for legal liability, and others for cost, predictability, agility, and so on.

Any one methodology is likely to be appropriate for only one of the boxes on one of the planes. Thus, at least 150 or so methodologies are needed (Capers Jones identifies 37,000 project categories [17]). That number is increased by the fact that technology shifts change the methodologies at the same time.

2. A Little Methodology Does a Lot of Good; After That, Weight is Costly

Figure 3 (see page 12) relates three quantities: the weight of the methodology being used, the size of the problem being attacked, and the size of the team. (*Problem size* is a relative term only. The problem size can drop as soon as someone has an insight about the problem. Even though problem size is highly subjective, some problems are clearly harder for a team to handle than others.) This figure illustrates that adding elements to a team's methodology first helps then hinders their progress [3].

The dashed line shows that a small team, using a minimal methodology, can successfully attack a certain size of problem. Adding a few carefully chosen elements to the methodology allows them to work more effectively and attack a larger problem. As they continue to add to the methodology, they increase the bureaucratic load they put on themselves and, being only a small team, start expending more energy in feeding the methodology than solving the problem. The size of the problem they can successfully attack diminishes.

The curve is similar for a large team (the solid line), but not as abrupt. The large team needs more coordination elements to work optimally, and has more people to feed the methodology as it expands. Eventually, even the larger team starts being less productive as the methodology size grows and solves the larger problems less successfully.

3. Larger Teams Need More Communication Elements

Six people in a room can simply talk amongst themselves and write on white boards. If 200 people were to try that, they would get in each other's way, miss tasks, and repeat each other's work. The larger team benefits from coordination. This is the slower rise in the large-team curve in Figure 3. The smaller team needs

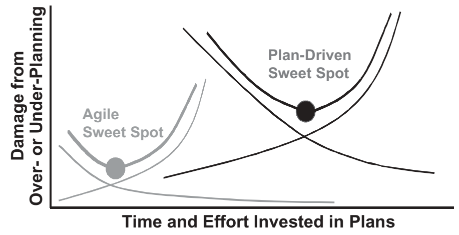
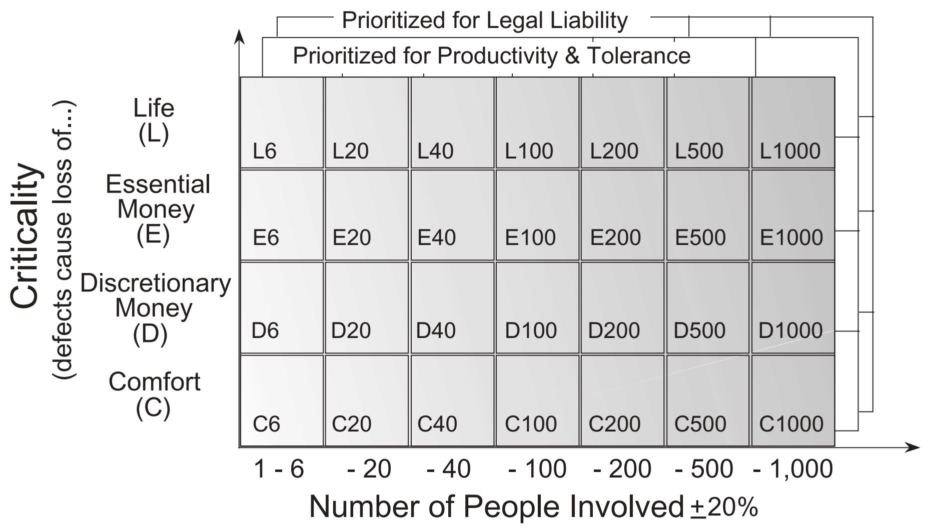


Figure 1: *Balancing Discipline and Flexibility with the Spiral Model and MBASE*

Figure 2: *Projects by Communication, Criticality, and Priorities* [3]


Prioritized for Legal Liability
Prioritized for Productivity & Tolerance
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
- 20
- 40
- 100
- 200
- 500
- 1,000
Number of People Involved ±20%

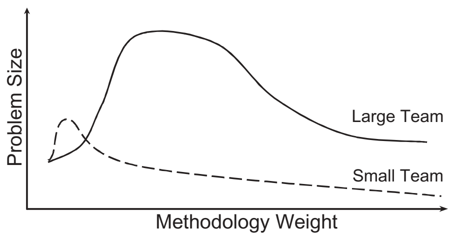


Figure 3: *A Little Methodology Goes a Long Way*

fewer coordination mechanisms and can treat them with less ceremony than can the larger team.

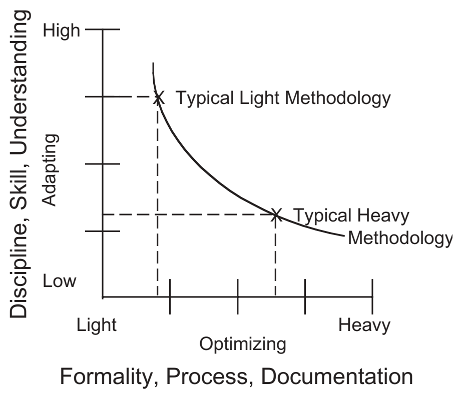
Although this principle should be obvious, many process designers try to find a single set of coordination elements to fit all projects.

4. Projects Dealing with Greater Potential Damage Need More Validation Elements

A team of developers charged with creating a proof-of-concept system does not have to worry about the damage caused by a system malfunction in the same way that a team charged with developing a final production system to be produced in vast quantities does. Atomic power plants, automated weapons systems, even cell phones or automobiles produced in the millions have such economic consequences that it is well worthwhile spending a great deal more time locating and eliminating each additional remaining defect. This is an MFI situation: It is worth spending a lot of money now to discover information about where those next defects are located.

For a system in which remaining defects have lower economic consequences (such as ordering food online from the company cafeteria), it is not worth spending as much money to discover that information. The team will consequently find it appropriate to use fewer and lighter validation techniques on the project

Figure 4: *Differences Between Adapting and Optimizing Approaches*



5. Formality, Process, and Documentation Are Not Substitutes for Discipline, Skill, and Understanding

Highsmith [4] points to the difference between discipline and formality, skill and process, understanding and documentation.

Discipline is an internal quality of behavior; formality is an externally visible result. Many of the best developers are very disciplined in their actions without using formal methods or documents.

Skill is an internal quality of action, typically of a single person, while process is an externally declared agreement, usually between several people. Individuals operating at high levels of skill often cannot say what process they follow. Processes are most useful in coordinating the flow of work between people.

"An agile project manager relies on discipline, skill, and understanding, while requiring less formality, process, and documentation."

Understanding is an internal realization; documentation is external. Only a small part of what people know can be put into external documentation, and that small part takes a lot of time.

Process designers often forget these differences, thinking that enough formality will impart discipline, enough process will impart skill, and enough documentation will impart understanding. An agile project manager relies on discipline, skill, and understanding, while requiring less formality, process, and documentation (Figure 4). This allows the team to move and change directions faster.

6. Interactive, Face-to-Face Communication Is the Cheapest and Fastest Channel for Exchanging Information

Understanding passes from person to person more rapidly when two people are standing next to each other, as when they are discussing at a white board. At that white board, they can use gestures, facial expressions, proximity cues, vocal inflection and timing, cross-modality (aural-visual) timing, and real-time feedback along modalities to discover what each knows, needs to know, and how to convey it [3, 19]. They use the white board as an external-marking device not just to draw, but also to hold some of their discussion points in place so they can refer back to them later.

As characteristics of that situation are removed, the communication effectiveness between the two people drops (Figure 5). On the phone, they lose the entire visual channel and cross-modality timing. With e-mail or instant messaging, they lose vocal inflection, vocal timing, and real-time question and answer. On videotape, they have visuals, but lose the ability to get questions answered. On audiotape, they again lose visuals and cross-modality timing. Finally, communi-cating through documents, they attempt to communicate without the benefit of gestures, vocal inflection and timing, cross-modality timing, proximity cues, or question-and-answer.

This principle suggests that for cost and efficiency improvements, a project team employ personal, face-to-face communication wherever possible. A decade-long study at MIT's Sloan School of Management in the 1970s and a recent research compilation both concluded that physical distance matters a great deal [20, 21].

The cost of imposing distances between people can be seen with a simple calculation. Suppose that a developer earns $2 per minute, and two people working side-by-side on the same problem exchange questions and answers at the rate of 100 questions each per week. Thus, for each minute on average that gets interposed between thinking the question and hearing the answer adds $200 of salary cost to the project per person per week, or about $10,000 per year. For a 10-person project, that one-minute average delay costs the organization $100,000 per year. Two offices being a few meters apart creates a one-minute delay. For offices around the corner or up a flight of stairs, the average delay is more on the order of five minutes ($500,000 per year).

The salary cost is actually the smaller cost. The larger cost is that when two people are more than about half a minute's travel apart, they simply do not ask each other many of those questions. Instead, they guess at the answers. Some percentage of those guesses are wrong, and those mistakes end up as defects in the system that must be found through debugging, external test, integration test, or even through system use.

7. Increased Communication and Feedback Reduces the Need for Intermediate Work Products

Intermediate work products – those not required by the final users of the system or the next team of developers -– tend to have two forms: a) promises as to what will eventually be constructed, and b) intermediate snapshots of the developers' knowledge (design descriptions).

This understanding, as we have already seen, moves faster through interactive than paper-based communication. Increasing the use of interactive communications will never entirely eliminate the need for archivable design documentation, but it can reduce it, particularly during the design and development stages of the project. Eventually, external documentation will be needed when none of the original designers are around, but that does not count as *intermediate* documentation.

Users who regularly get to see the developing system stop needing elaborate promises of what they will be given. This is an MFI issue. If the users are not going to get to see the result for a year or two, then it is worth a lot to create the most accurate promise possible. If on the other hand, the users get to see results every few days or weeks, then a better use of the project's money is to simply build the system and show it to the users.

There is, however, a MFF issue at play here as well since there are diminishing returns on the MFI issue of creating that promise. No amount of care in crafting a detailed promise can capture the unpredictable reaction of the users on seeing the final product in their own environment as they perform their work assignments. The time and money spent on guessing at the users' response to the delivered system would be better allocated to deal with their response on seeing the real system.

Mock-ups, prototypes, and simulations deal with the MFI aspects of the situation. They are an expenditure of resources to discover information sooner. The MFF aspects of the situation are handled through incremental delivery with iterative re-work, allocating resources for the inevitable surprises resulting from real delivery.

Interim Summary

The natural tension between agility-focused, plan-driven, and cost-sensitive project teams is explained in part by their interpretations of what counts as a *money-for-information* proposition, what counts as a *money-for-flexibility* proposition, and how much money to spend on each. We have seen how people with various priorities use those economic strategies differently.

It is particularly important, in working with the first seven principles, that each be used to tune a project's running rules, of particular importance is that each project team declares its priorities as well as its communication and validation requirements. With those in place, the team can orient itself to the amount of face-to-face communication it can manage, and the extra methodology weight it should appropriately set in place.

The principles are intended to be used as slider scales. Too much toward each end of the sliding scale brings its own sort of damage.

The second part of this article will present the final three principles, then pull from the collected information to suggest specific actions that leaders of plan-driven and cost-sensitive projects can take to either improve their strategies, or at least hedge their bets against future surprises.◆

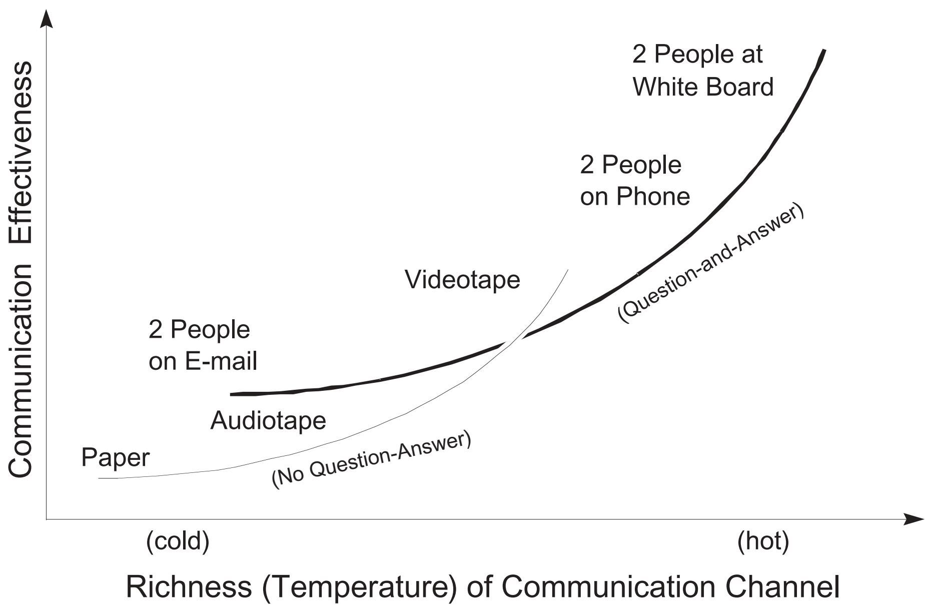


Figure 5: *Increasing Communication Effectiveness from Richer Communication Channels*

References

1. Beck, Kent. eXtreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.
2. Coad, P., E. Lefebvre, and J. De Luca. Java Modeling In Color With UML: Enterprise Components and Process. Upper Saddle River: Prentice Hall, 1999.
3. Cockburn, Alistair. Agile Software Development. Boston: Addison-Wesley, 2001.
4. Highsmith, Jim. Agile Software Development Ecosystems. Boston: Addison-Wesley, 2002.
5. Schwaber, K., and M. Beedle. Agile Software Development with Scrum. Upper Saddle River: Prentice Hall, 2001.
6. Highsmith, Jim, and Alistair Cockburn. "Agile Software Development: The Business of Innovation." IEEE Software 34.9 (2001): 120-122.
7. Cockburn, Alistair, and Jim Highsmith. "Agile Software Development: The People Factor." IEEE Software 34:11 (2001): 131-133.
8. Boehm, Barry, and D. Port. "Balancing Discipline and Flexibility with the Spiral Model and MBASE." CROSSTALK Dec. 2001: 23-30. Available at: <www.stsc.hill.af.mil/crosstalk/2001/dec/boehm.pdf>.
9. Software Engineering Institute. Capability Maturity Model® Integration℠ V1.1 (CMMI℠). Pittsburgh: SEI, 2002. Available at: <www.sei.cmu.edu/cmmi>.
10. Humphrey, Watts. A Discipline for Software Engineering. Boston: Addison-Wesley, 1997.
11. Paulk, Mark C. "Agile Methodologies and Process Discipline." CROSSTALK Oct. 2002: 15-18.
12. Highsmith, Jim. "What Is Agile Software Development?" CROSSTALK Oct. 2002: 4-9.
13. Cockburn, Alistair. "Agile Software Development Joins the Would-Be Crowd." Cutter IT Journal Jan. 2002: 6-12.
14. Sullivan, K., P. Chalasani, S. Jha, and V.

Sazawal. "Software Design as an Investment Activity: A Real Options Perspective," in Real Options and Business Strategy: Applications to Decision Making. L. Trigeorgis, ed. London: Risk Books. Dec. 1999.
15. Mathiassen, L. "Reflective Systems Development." Scandinavian Journal of Information Systems. Vol. 10, No. 1, 2. Gothenburg, Sweden: The IRIS Association, 1998: 67-117.
16. Hohmann, L. Journey of the Software Professional. Upper Saddle River: Prentice-Hall, 1997.
17. Jones, Capers. Software Assessments, Benchmarks, and Best Practices. Boston: Addison-Wesley, 2000.
18. Cockburn, Alistair. "Selecting a Project's Methodology." IEEE Software 17.4 (2000): 64-71.
19. McCarthy, J., and A. Monk. "Channels, Conversation, Cooperation and Relevance: All You Wanted to Know About Communication But Were Afraid to Ask." Collaborative Computing 1.1 (Mar. 1994): 35-61.
20. Allen, T.J. Managing the Flow of Technology. Cambridge, MIT Press, 1977.
21. Olson, G. M., and J. S. Olson. "Distance Matters." Human-Computer Interaction 15 (2001): 139-179.

About the Author

Alistair Cockburn, an internationally recognized expert in object technology, methodology, and project management, is a consulting fellow at Cockburn and Associates. He is author of "Surviving Object-Oriented Projects," "Writing Effective Use Cases," and "Agile Software Development," which have won Jolt Productivity Book Awards. He is one of the original authors of the Agile Software Development Manifesto and founders of the AgileAlliance, and is program director for the Agile Development Conference held in Salt Lake City. Cockburn has more than 20 years experience leading projects in hardware and software development.

1814 Fort Douglas Circle
Salt Lake City, UT 84103
Phone: (801) 582-3162
Fax: (775) 416-6457
E-mail: alistair.cockburn@acm.org

Web Sites

AgileAlliance

www.agilealliance.org/home

The AgileAlliance is a nonprofit organization dedicated to promoting the concepts of agile software development, and helping organizations adopt those concepts, which are outlined by the Agile Software Development Manifesto and can be found on this Web site. The AgileAlliance was designed to be lightweight, initially consisting of a board of directors, one administrator, and a set of bylaws. Just like agile processes, all work and operations within the AgileAlliance is intended to emerge from subsets of members that self-organize into programs.

Agile Development Conference

http://agiledevelopmentconference.com

The Agile Development Conference is a conference on delivering fit-for-purpose software under shifting conditions, using people as the magic ingredient. A number of techniques, practices, and processes have been identified to do this, and more will be found in the future. This conference will discuss people working together to create software, and the tools, techniques, practices, and issues involved. Come here to learn, or, even better, to name them. This conference has recently been funded and is still being organized. Read the conference vision and structure and the request for participation.

Crystal Methodologies

www.alistair.cockburn.us/crystal

Crystal collects together a self-adapting family of "shrink-to-fit," human-powered software development methodologies based on these understandings:

- Every project needs a slightly different set of policies and conventions, or methodology.
- The workings of the project are sensitive to people issues, and improve as the people issues improve, individuals get better, and their teamwork gets better.
- Better communications and frequent deliveries communication reduce the need for intermediate work products.

This site is a resource for people wanting to understand those ideas, to find more about improving skills and teaming, and to identify some project policies to use as a starting point. This site is set up as a museum of information, with exhibit halls, exhibit rooms, exhibits with notes, and a discussion area.

North Carolina State University

www.csc.ncsu.edu

The North Carolina State University's (NCSU's) Computer Science Department cites strengths in the areas of software systems, communications and performance analysis, theory and algorithms, and computer architecture. Founded in 1967, NCSU's is one of the oldest computer science departments in the country and the only one at a state-assisted Research I University. The university hosts a small workshop, "Agile Software Development Methodologies: Raising the Floor or Lowering the Ceiling" at <http://collaboration .csc.ncsu. edu/agile>.

XBreed

www.xbreed.net/index.html

XBreed is the product of mixing SCRUM, eXtreme Programming (XP) and Alexanderian ideas. Information technology is the result of developing multiple applications and shared components as fast as humanly possible. Combining Scrum and XP was very natural: Scrum provides a solid management framework, while XP provides a basic but complete set of engineering practices. The result is a lean but very effective way to run software projects. In addition, Scrum practiced at the application team level – provided a shared resources team is involved – can lead to re-usability. XBreed is a free method. This Web site includes everything you need to know to run XBreed projects.

Agile Methodologies and Process Discipline

Mark C. Paulk
Software Engineering Institute

Agile methodologies have been touted as the programming methodologies of choice for the high-speed, volatile world of Internet and Web software development. They have also been criticized as just another disguise for undisciplined hacking. The reality depends on the fidelity to the agile philosophy with which these methodologies are implemented, and the appropriateness of the implementation for the application environment. This article addresses these issues and summarizes and critiques the compatibility of agile methodologies with plan-driven methodologies as described by the Capability Maturity Model® for Software.

Agile methodologies, such as eXtreme Programming (XP), have been touted as the programming methodologies of choice for the high-speed, volatile world of Internet and Web software development. They have also been criticized as just another disguise for undisciplined hacking. Although creators of agile methodologies usually espouse them as disciplined processes, some have used them to argue against rigorous software process improvement models such as the Capability Maturity Model® (CMM®) for Software (SW-CMM®) [1].

Many organizations moving into e-commerce (and e-government) have existing CMM-based initiatives (and possibly customers demanding mature processes) and desire an understanding of whether agile methodologies can address CMM practices adequately. Usually, the reality depends on 1) the fidelity to the agile philosophy with which these methodologies are implemented, and 2) the appropriateness of the implementation for the application environment.

This article recaps the Agile Software Development Manifesto and its underlying principles. The compatibility of agile methodologies with plan-driven methodologies as described by the CMM is summarized and critiqued. Although agile methodologies can be characterized as *lightweight methodologies* that do not emphasize process definition or measurement to the degree that models such as the CMM do, a broad range of processes can be considered valid under the CMM. The conclusion is that agile methodologies advocate many good engineering practices, although some practices may have an extreme implementation that is controversial and counterproductive outside a narrow domain.

For those interested in process improvement, the ideas in the agile movement should be thoughtfully considered. When rationally implemented in an appropriate environment, agile methodologies address many CMM Level 2 and 3 practices. The ideas in the agile movement should be carefully considered for adoption where appropriate in an organization's business environment; likewise, organizations considering agile methodologies should carefully consider the management and infrastructure issues of the CMM.

Agile Methodologies

Many names have been used for the agile methods, including Internet-speed, lightweight, and lean methodologies. Similarly, plan-driven methodologies have been described as rigorous, disciplined, bureaucratic, heavyweight, and industrial-strength. Some of these descriptors can be considered derogatory, e.g., lightweight or bureaucratic. *Agile* is the term preferred by the AgileAlliance, a group of software professionals dedicated to promoting the concepts of agile software development. *Plan-driven* was coined by Barry Boehm [2] to characterize the opposite end of the planning spectrum from agile methodologies.

"The conclusion is that agile methodologies advocate many good engineering practices, although some practices may have an extreme implementation that is controversial and counterproductive outside a narrow domain."

Any discussion of agile methodologies should begin with the fundamentals of agile as expressed by its proponents. The Manifesto of the AgileAlliance found at <www.agilemanifesto.org> states:

> We are uncovering better ways of developing software by doing it, and helping others do it. Through this work we have come to value:
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value on the items on the right, we value the items on the left more.

The principles behind the agile manifesto are as follows:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Agile methodologies are usually targeted toward small- to medium-sized teams building software in the face of vague and/or rapidly changing requirements. Agile teams are expected to be co-located, typically with less than 10 members.

Process Discipline in the Agile Methods

Why would we challenge the principles of agile methodologies? Do not most professionals share the objectives espoused in the agile movement? In one sense, the values expressed in the agile manifesto should be captured in any modern software project, even if the implementation may differ radically in other environments. Customer satisfaction, communication, working software, simplicity, and self-reflection may be stated in other terms, but without them, non-trivial projects face almost insurmountable odds against success.

In effective CMM-based improvement, when defining processes, organizations should capture the minimum essential information needed, use good software design principles (such as information hiding and abstraction) in structuring the definitions, and emphasize usefulness and usability [3]. One of the consequences of the Level 1 to Level 2 culture shift is demonstrating the courage of convictions by becoming realistic in estimates, plans, and commitments.

Much of the formalism that characterizes most CMM-based process improvement is an artifact of large projects and/or severe reliability requirements, especially for life-critical systems. The hierarchical structure of the SW-CMM, however, is intended to support a broad range of implementations within the context of the 18 key process areas and 52 goals that compose the requirements for a fully mature software process. It is true, however, that the CMM emphasizes explicitly capturing knowledge via documentation and measurement – a process emphasis. The challenge for many in adopting agile methodologies lies in the (perceived) problems in de-emphasizing processes, documentation, contracts, and planning.

Individuals Over Process

Agile methodologies assume the programmers are generalists rather than specialists. Competent generalists are hard to come by, but this is an endemic problem in any technically demanding discipline. Specialist knowledge in a domain can be needed, however, which may lessen the effectiveness of practices such as pair programming.

The foundation of software engineering in the SW-CMM at Level 1 is competent people, sometimes doing heroics, all too frequently working to overcome *the system* to do professional work. In spite of heroics, the foundation that is assumed in the CMM is competent professionals.

"Software professionals want to take pride in their work, but how can they when managers say, 'I'd rather have it wrong than have it late. We can always fix it later.' "

Without competent professionals, the best software process is ineffective – because the work we do in software projects is human-centric and design-intensive, and the process is what we do.

Software professionals want to take pride in their work, but how can they when managers say, "I'd rather have it wrong than have it late. We can always fix it later." When program managers acknowledge that making the schedule is the primary consideration in raises and promotions, what is the impact on motivation, quality, and professionalism? These are fundamental management issues, which is why the focus at Level 2 of the SW-CMM is on project management, which empowers competent professionals to do quality work.

It is interesting to note that, although agile methodologies emphasize individuals over process, the set of practices in an agile methodology addresses the same kind of planning and commitment issues as the focus on basic project management at CMM Level 2. An *agile methodology* that ignored customer collaboration and incremental development would almost certainly fail. Agile practices are synergistic, and the success of agile methodologies depends on the emergent properties of the set of practices as a whole.

Working Software Over Documentation

The agile emphasis on tacit rather than explicit knowledge, as externally captured in documentation, can be a high-risk choice in many environments such as government contracting, but it can be an effective choice if rationally made. When agile advocates denigrate the value of documentation, they lessen their credibility in the eyes of many experienced professionals. The tone in arguing against non-essential documentation differs from arguing that documentation is an inefficient waste.

eXtreme Programming expert Bob Martin said at the 2001 XP Universe conference that he ran into someone who said his organization was using XP. Martin asked him how pair programming was viewed, and the reply was, "We don't do that." Martin asked how refactoring was working out, and the reply was, "We don't do that." Martin asked how well the planning game was working, and the reply was, "We don't do that." "Well," Martin asked, "then what are you doing?" "We don't document anything!" was the answer.

Success carries the seeds of failure, and the agile methodologists are concerned that some adopting these new ideas do not really understand what an agile methodology is – and it is not ad hoc, chaotic programming.

When considering process documentation, the element that is missing from agile methodologies, which is crucial for the SW-CMM, is the concept of *institutionalization*, i.e., establishing the culture that "this is the way we do things around here."

Although implicit in some agile practices such as the peer pressure formed by pair programming, infrastructure is important for institutionalizing good engineering and management practices. The key process areas in the CMM are structured by common features that deal with implementing and institutionalizing processes. The institutionalization practices for each key process area map to all the goals within the area, so a naïve agile implementation that ignored these cultural issues would fail to satisfy any CMM key process area.

As implementation models that focus on the development process, these issues are largely outside the focus of the agile methodologies, but they are arguably crucial for their successful adoption.

Over-documentation is a pernicious problem in the software industry, especially in Department of Defense (DoD) projects. Software maintainers have long known that the only documentation you can really trust is the code (and those of us with experience debugging compiler and run-time defects doubt even that). Having said that, an architectural description of the system that provides a tour of the top-level design can be invaluable to maintainers.

From a technical perspective, as projects become larger, emphasizing a good architectural *philosophy* becomes increasingly critical to project success. Major investment in the design of the product's architecture is one of the practices that characterizes successful Internet companies [4]. Architecture-based design, designing for change, refactoring, and similar design philosophies emphasize the need for dealing with change in a systematic fashion.

One of the compromises that agile methodologists are likely to be required to make as they move into larger projects and applications that are life- or mission-critical is a stronger emphasis on documenting the architecture and the design of the system. In turn, plan-driven methodologists must acknowledge that keeping documentation to a minimum, useful set is also necessary. What benefit do we really get from detailed designs where the programming design language is nearly as large as the code?

Much of the controversy with respect to the technical issues centers on what happens as projects scale up. Practices that rely on tacit knowledge and highly competent professionals may break down in larger teams with their rapidly expanding communication channels and coordination challenges. However, replacing those practices with ones appropriate for large teams may result in losing the emergent properties of the agile methodology.

Customer Collaboration Over Contracts

The degree of trust implicit in relying on customer collaboration rather than a contract is not justified in many customer-supplier relationships. Even when the relationship begins with the best of intentions and the highest of expectations on both sides, one of the main difficulties in learning from experience is "the use of unaided memory for coding, storing, and retrieving outcome information" [5], with the consequence that "change can make liars of us, liars to ourselves" [6]. As time goes by, as things change, our unaided memories become unreliable.

The reliance of agile methodologies on tacit knowledge is therefore vulnerable to perception shifts over time, yet tacit knowledge may be much more effective than external, explicit knowledge in setting expectations and driving behavior. In a government-contracting context, federal acquisition regulations establish a context for ensuring fair play – even if it is not necessarily an effective and efficient environment. This can be considered a problem in expectations management. The agile methodologies manage customer expectations by insisting on an ongoing customer interaction and rapid iteration.

"One of the most significant barriers to implementing an agile methodology is likely to be an inability to establish and maintain close and effective customer collaboration."

Ignoring possible regulatory issues, the *stories* in XP, in conjunction with an evolutionary life cycle and ongoing customer-supplier communication [7], document requirements and commitments in a manner that could satisfy the goals of requirements management and software project planning in the SW-CMM. Will such a set of stories satisfy a DoD customer that the requirements are adequately stated and that commitments *as driven by the customer* are being met? Or will the natural desire for a more comprehensive requirements statement drive the customer towards a requirements specification that lacks the dynamic capability desired for an agile methodology?

Perhaps an honest answer to this type of question reveals more about the comfort levels of both customer and supplier in an *agile relationship*. One of the most significant barriers to implementing an agile methodology is likely to be an inability to establish and maintain close and effective customer collaboration – and this barrier is likely to be erected on the customer's side of the relationship.

Responding to Change Over Planning

Dwight Eisenhower is quoted as saying that planning is more important than the plan. And one of the great military axioms is that no battle plan survives contact with the enemy. That said, planning – and preparation – are prerequisites to success. Planning for change is quite different from not planning at all.

Agile methodologies, with their rapid iterations, require continual planning. Customer collaboration and responsiveness to change are tightly linked, if perhaps inconsistent with typical government-contractor relationships. One of the shifts in acquisition strategy in recent years has been toward prototyping, evolutionary development, and risk-driven life cycles. With their emphasis on addressing requirements volatility, agile methodologies could be a powerful synthesis of practices that DoD contractors could leverage to make planning more responsive to change.

Stepping Up to the Plate

The greatest challenge in taking advantage of the virtues of agile methodologies may lie in convincing acquisition agencies to *step up to the plate* and use agile methods where appropriate. Hardly lesser is the challenge in convincing agile advocates to consider modifying the agile methodologies to suit new arenas. We have to decide where to place the *balance point* in documentation and planning to alleviate the concerns of the stakeholders (and regulatory requirements) while achieving the flexibility and benefits promised in the agile philosophy.

Agile methodologies may wind up being the preferred process in many environments, yet be inappropriate in contexts such as life-critical systems or high-reliability systems. Modifications to the agile methodologies needed for those environments may be great enough that the synergistic effects of the set of practices in an agile methodology are lost. Emergent properties in a system are sensitive to interdependencies. Arguing that agile methodologies are not suitable for all environments is not the same as saying they are suitable for none.

Contractual commitments explicitly based on evolutionary or incremental life cycles are desirable. Plans with *miniature milestones* that are detailed in the short term and conceptual in the long term are possible. Processes that capture the minimum essential information needed to reliably and consistently perform the work and documentation that captures useful information are feasible. Just because these objectives are desirable, possible, and feasible does not, however, mean they are easily realized. Selecting an appropriate balance point requires an open mind from both agile and plan-driven methodologists on both the supplier and customer sides of the equation.

Conclusions

Agile methodologies imply disciplined processes, even if the implementations differ in extreme ways from traditional software engineering and management practices; the extremism is intended to maximize the benefits of good practice [8]. The SW-CMM tells *what* to do in general terms, but does not say *how* to do it; agile methodologies provide a set of best practices that contain fairly specific how-to information – an implementation model – for a particular kind of environment.

Even though agile methodologies may be compatible in principle with the discipline of models such as the CMM, the implementation of those methodologies must be aligned with the spirit of the agile philosophy and with the needs and interests of the customer and other stakeholders. Aligning the two in a government-contracting environment may be an insurmountable challenge.

As we learn empirically what works well in the agile methodologies and how far they can be extended into different environments, we should expect software engineering to adapt and adopt the useful ideas of the visionaries in the agile movement. This will include using data to separate the wheat from the chaff as we identify what is useful, and what is limited in its application. Laurie Ann Williams, for example, has integrated pair programming into an extension of the Personal Software Process[SM] called the Collaborative Software Process, and demonstrated that performance improves [9].

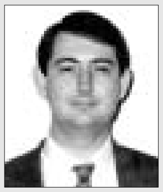
Many of the practices in the agile methodologies are good practices that should be thoughtfully considered for any environment. While the merits of any of these practices can be debated in comparison with other ways of dealing with the same issues, none of them should be arbitrarily rejected. Perhaps the biggest challenge in dealing effectively with both agile and plan-driven methodologies is dealing with extremists in both camps who refuse to keep an open mind.◆

[SM] Personal Software Process is a service mark of Carnegie Mellon University.

References

1. Paulk, Mark C., Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. The Capability Maturity Model®: Guidelines for Improving the Software Process. Boston: Addison-Wesley, 1995.
2. Boehm, Barry. "Get Ready for Agile Methods, With Care." IEEE Computer Jan. 2002.
3. Paulk, Mark C. "Using the Software CMM with Good Judgment." ASQ Software Quality Professional June 1999.
4. MacCormack, Alan. "Product Development Practices That Work: How Internet Companies Build Software." MIT Sloan Management Review Winter 2001.
5. Einhorn, Hillel J., and Robin M. Hogarth. "Confidence in Judgment: Persistence of the Illusion of Validity." Psychological Review 85.3 (1978).
6. Dawes, Robyn M. Rational Choice in an Uncertain World. Orlando: Harcourt Brace Jovanovich, 1988.
7. Beck, Kent. eXtreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.
8. Paulk, Mark C. "Extreme Programming From a CMM Perspective." IEEE Software 34.11 (2001).
9. Williams, Laurie Ann. "The Collaborative Software Process." Diss. University of Utah, Aug. 2000.

About the Author



Mark C. Paulk is a senior member of the Technical Staff at the Software Engineering Institute. He has been with the SEI since 1987. Paulk was the "book boss" for Version 1.0 of the Capability Maturity Model® for Software and the project leader during the development of Software CMM® Version 1.1. His current interests center on high maturity practices and statistical control for software processes.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Phone: (412) 268-5794
Fax: (412) 268-5758
E-mail: mcp@sei.cmu.edu

Odyssey and Other Code Science Success Stories

John Manzo
AgileTek L.L.C.

Code Science® is an agile software development method based on eXtreme Programming (XP). This article describes the success achieved using code science to develop a complex industrial automation application. With a brief review of XP as background, code science is described in terms of refinements made to XP in applying it to a wide variety of application domains and industries over a period of almost four years. Included are real-world insights from the developers' experience in applying this agile development method, concluding with a quantitative measure of the effectiveness of XP since its inception almost four years ago.

Using Code Science®, an agile software development methodology based on eXtreme Programming (XP), we[1] recently delivered an application (code-named *Odyssey*) consisting of 400,000-plus executable source lines of code (ESLOC) to one of the world's premier industrial automation companies. The application was written in C++ by as many as 17 developers (including some of our customer's staff) in approximately 15 months. We are geographically remote from this customer.

Odyssey was delivered a month and a half ahead of schedule with a productivity rate of 43 ESLOC per coding hour. During the project duration, approximately 2,400 defects were found and fixed, yielding a *captured* defect density of six defects per thousand lines of code (KLOC). During a thorough, more than six-week customer-conducted acceptance test, only about 200 defects were found (none severe), yielding a *delivered* defect density of 0.5/KLOC. The customer is delighted with the product and is confident of the competitive edge achieved.

The Application

The Odyssey program consists of two distinct but related scalable vector graphics applications. The first is a run-time application that issues real-time commands from a touch screen panel to devices known as programmable logic controllers, which are used in manufacturing assembly processes. The second is a panel design application that enables human-machine interface engineers to develop the graphical equivalent of a hardware panel made up of buttons, gauges, and other control and monitoring devices.

Why Agile Methods

We began experimenting with XP several years ago, and actually began our first XP project a few months before Kent Beck published his first book on the subject [1]. Before that time, we used several traditional waterfall and rapid application design-based methods. We were impressed at how quickly our first XP project was completed.

In a side-by-side comparison of XP and waterfall on the very same project, the XP team delivered their final product when the other team was less than 50 percent complete. Since then, we refined our initial XP approach to encompass successive refinements that became known as Code Science.

"In a side-by-side comparison of XP and waterfall on the very same project, the XP team delivered their final product when the other team was less than 50 percent complete."

Code Science is *largely* based on the twelve tenets of XP. These are as follows:

1. Customer at the Center of the Project. The customer is treated as a full-fledged member of the development team with access to all the information that the rest of the team is privy to (e.g., defect logs, issue lists, etc.).
2. Small Releases. Simple releases are put into production early and updated frequently on a very short cycle (two to three days). New versions are released at the end of each iteration (three to five weeks).
3. Simple Design. A program built with XP should be the simplest program that meets the *current* requirements.
4. Relentless Testing. XP teams focus on validation of the software at all times. Programmers develop software by writing tests first followed by software that fulfills the requirements reflected in the tests. Customers provide acceptance tests that enable them to be certain that the features they need are provided.
5. Refactoring. The system design is improved throughout the entire development process. This is done by keeping the software clean, without duplication, as simple as possible, and yet complete – ready for any change that comes along. (Martin Fowler defines refactoring as "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" [2]).
6. Pair Programming. XP programmers write all production code in pairs: Two programmers work together at one machine.
7. Collective Ownership. All the code belongs to the all the programmers. This enables the team to work at full speed. When something needs changing, it can be changed without delay. It is important to note that an effective configuration management discipline is an important enabler of this practice.
8. Continuous Integration. The software system is integrated and built multiple times per day (ideally, every time a task is finished). Continual regression testing prevents functional regressions when requirements change. This also keeps the programmers on the same page and enables very rapid progress.
9. 40-Hour Workweek. Tired programmers make more mistakes. XP teams do not work excessive overtime, which keeps them fresh, healthy, and effective.
10. On-Site Customer. An XP project is steered by a dedicated individual who is empowered to determine requirements, set priorities, and answer questions.
11. Coding Standards. For a team to work effectively in pairs and to share ownership of all the code, programmers need to write the code in the same way with rules that ensure the code communicates clearly.
12. Metaphor. Development is guided with a

® Code Science is registered in the U.S. Patent and Trademark Office.

simple shared story of how the overall system works. XP was originally used to develop a payroll program at Chrysler Corporation [3]. The team used the metaphor of an assembly line to describe the process of *building* a payroll check.

The key tenet in XP is iterative development and the unforgiving honesty of working code. The concept of iterative development has been around for a long time. However, XP does have some limitations such as scaling – the ability to add large numbers of developers to a project that requires them. (Most XP practitioners consider six to 12 developers to be the practical limit.) It was necessary to modify XP to develop a methodology that would work on large projects, across multiple application domains, and for clients with diverse and sometimes very specialized needs, for example, regulatory environments such as the Food and Drug Administration (FDA), where there is a strong need for extensive documentation.

The Code Science Difference

A way to quickly understand Code Science is to think of it as XP with a delta (a set of differences). Some of the differences are additive (+), some are subtractive (-) and some are simply modifications or refinements (▲). Following are a set of differences defined.

+ Business Process Analysis

In employing XP there is an implicit assumption that the client basically knows what it wants and, therefore, the requirements gathering process can begin with user stories. Although this is often the case, many of our clients need to focus and solidify their ideas and, most importantly, determine with clarity what they *need* rather than what they *want*. To accomplish this, we developed a process that helps bring focus and understanding to the client's business needs, prioritizing features and functions in terms of the business value they represent. This first step, which is formally absent from XP, is a step we can take when *necessary* to ensure that the story-gathering effort produces stories based on a real vs. perceived need.

+ Delphi Estimation

The Delphi method of estimating involves three or more participants who discuss the work and provide anonymous estimates of the time for completion (usually in units of *perfect programmer hours* – i.e., an ideal, no interruption, period of time). These estimates are tallied and a mean and standard deviation is made known to the participants. Discussion ensues among the participants as to the differences in the estimates (which remain anonymous). This continues for successive rounds (usually three) until the standard deviation (a measure of uncertainty) is made sufficiently narrow. Once the number of perfect programmer hours is known, a loading factor is applied to convert this estimate to *real* programmer hours.

+ Componentized Architecture

For complex systems, it is especially important to assure conceptual integrity in the final product. Also, because complex systems can be large, it is also important to enable the system to be developed in an environment of distributed ownership. Among the least understood areas of XP is the notion of *design-as-you-go* through refactoring. To some, especially those who equate design and architecture, this means no up-front architecture, and, by implication, any architecture that the delivered system may have is a defacto one at best.

"The best architectures are isomorphic (one-on-one) mappings between problem and program space."

Architecture of a system simply means identifying the constituent components of the system and defining the interrelationship(s) between them. The best architectures are isomorphic (one-to-one) mappings between problem and program space. This ensures that a system's underlying structure and components mirror the problem being solved. This means that for the *program* to change requires that the *problem* changes and, therefore, you are *change-proofing* your program. While there may be more efficient ways to solve a problem (e.g., creating one module to perform similar functions by invoking it in a context sensitive way), this efficiency will almost always come at the expense of time spent debugging and later modifying the program if one or more of the functions change.

However, it also means something more. By defining the relationships between the various components, one has gone most of the way toward establishing agreements for the interfaces. The power of interface agreements is that they serve as restrictive liberators. In other words, the individuals working on various system components are free to design the internals of those components without regard for potential untoward effects on the rest of the system – so long as the interface agreements are honored.

By spending a relatively small amount of time up front, one can ensure both a product with conceptual integrity and a project that can scale.

+ Automated Contract and Regression Testing

Given that XP is premised upon the need to embrace change, making it easy to perform regression testing is an important part of any XP project. We have taken this to the next level by implementing the capability to perform contract testing, which checks for the existence of predefined pre-conditions, post-conditions and class invariants. (As an example, an overdrawn flag in your checking account is invalid if there is a positive balance remaining after the last transaction.)

+ Story Actors

We have added to the notion of *stories* the concept of story *actors*. Actors are personifications of the various categories of users the system will encounter. Thinking of the requirements in terms of actors brings the requirements to life as well as unmasks nuances that would otherwise remain invisible to both the developers and the customer.

+ Wall Gantts

Frequently used in project management, a Gantt chart provides a graphical illustration of a schedule that helps to plan, coordinate, and track specific tasks in a project. We have taken the concept one step further and adapted it to agile methods by creating a physical construct using twine, pushpins, and index cards. The twine is used to create a line on a wall. Tasks, written on cards, are folded in half and hung on the line (one line for each project participant). Index cards with dates (one for each day of an iteration, which usually lasts three to five weeks) are pinned across the top of the chart.

Physically constructing the Gantt chart makes it very easy to move tasks around, drive out dependencies, and load balance. Because the chart is wall size, it is easy for the team to stand around the chart to discuss the state of the project in near real-time (each day starts with a stand-up meeting). The wall Gantt also provides clear ownership for development efforts, encourages accountability, and serves as the team's *war room* and center of the project universe.

+ Automatic Document Generation

Through a tool we have built called Doc-It (similar to JavaDoc), we are able to reduce the burden and streamline the process of generating documentation that describes the inner workings of the code. Experience shows that it is a poor practice to separate documentation from the code that it describes. Updating source code documentation is difficult enough, but once the documentation is separated from the code, it is "out of sight, out of mind." To deal with this, a programmer simply needs to tag a comment in the source code and Doc-It creates automatic HTML Application Program Interface documentation with every build.

Doc-It traverses source code directories, creating a navigable hierarchy (directory, class, method) and creates a Web page for each source file. This makes the documentation easily accessible to new and existing team members. It also makes the documentation easily accessible to clients during co-development or during knowledge transfer phases.

▲ *Pair Programming*

Although our experience proves pair programming to be extremely effective, for many routine programming tasks, pair programming has not shown itself to be cost effective. On the other hand, for anything either algorithmically or logically complex, pair programming is a must. The default is to program in pairs, but the team gets to decide which modules will be coded solo.

- 40-Hour Workweek

While we strive to provide the highest quality of life for all our staff members, it is unrealistic to expect that our client's time-critical requirements will not sometimes necessitate sustained periods of activity. Treating a 40-hour workweek as a hard requirement is often impractical.

- Metaphor

Metaphor is not included in Code Science. While we concede that it has benefits, so far we have not found a need to incorporate the use of metaphor in our methodology.

+ Flexibility to Meet Client's Special Needs

Some of our clients have special needs that are not accounted for by pure XP (e.g., in highly regulated environments such as biomedicine, the FDA requires specialized documentation and traceability for certain types of software). Code Science eliminates this XP limitation by incorporating a *special needs* provision in our methodology.

Application to Odyssey

Code Science is used on all Geneer software development projects. The Odyssey project was no exception. However, no two projects are the same. Each emphasizes certain of the specific tenets described above to differing degrees. In the interest of brevity, we describe some of our developers' more salient experiences and insights in applying these tenets.

The Customer Is at the Center

XP talks about having the customer on-site. While this is ideal, our experience in using XP/Code Science over the last four years is that it is seldom practical unless your customer is internal. In the case of Odyssey, the customer was located hundreds of miles away.

More important than physical location, however, is putting the customer at the center of your project as described earlier. In an XP/Code Science project, there is no attempt to hide information from the customer. While we did not insist the customer be physically in our facility, we did request that they be present during iteration planning, periods of critical knowledge transfer, or to approve test plans and validate their results – usually at the beginning/end of iteration. When this was impractical, or for routine communications, we used e-mail, conference calls, WebEx sessions, or videoconference. With active customer participation, the resulting product can be everything that the customer expects it to be.

Refactoring

Refactoring does not mean re-working. Do not partially write a feature with the *intent* of refactoring to get it complete later. Keep the changes simple, but keep them atomically complete.

Pair Programming

Pair programming was especially useful in ramping up a new staff member. It was also quite useful for chasing down complex defects. For simple modules, the team found it more expedient to use the white board in pairs for 15 minutes, then program solo.

Continuous Integration

The team performed builds at least daily, more often, two or three times a day. With a good automated build program, you cannot build too often. Our builds are generated with a custom, home-grown application that creates builds at 4 a.m. and again at 3 p.m. This gives the team a fresh build every morning and also one to work on in the afternoon. Besides performing the physical build, we are also informed if the build is broken (e.g., cannot compile because of a syntax problem, or a configuration management issue – checked in one file and not another, etc.).

40-Hour Week

During a long period of peak activity, the team found it helpful to make their work environment homier. By making their workplace a more dorm-like environment, they significantly eased the stress of the long, often intense, workdays and nights.

Componetized Architecture

A high-level architecture was defined at the beginning of the first iteration, and as more information became available, more detail was added to successive iterations. Team leads would spend perhaps two days with their teams using white boards for a four to six-week iteration. We found that the biggest mistake one can make here is to attempt to get too detailed about something for which there is insufficient information.

Story Actors

Because most of the team never worked on an industrial automation application, actors helped the team get familiar with the client's domain. When the team took a field trip, they could identify the user types by their actor names (representative of their role-play, more than their job title). It helped the team understand the business and how the product would be used in stories. The requirements were written in terms of how the system would be used, vs. desired functions. By associating who is doing what, it helps conceptualize and compartmentalize the functions.

Wall Gantts

Wall Gantts make load balancing easy and kept the project on track. The whole team sees the actual size of the function based on the task cards and there is great satisfaction in putting completion stickers on each card. An extremely useful management tool, Wall Gantts also helped to reveal issues and expose risks.

Large Team Experience

Although the overall team was divided into subteams, stand-up meetings were typically with the entire team. Team leads summarize and add detail with other team members as needed. As tasks are completed, people can move from one team to another.

Conclusion

During a period of almost four years, XP/Code Science has been employed on 14 projects across a wide variety of application domains and industries such as aerospace, telecommunications, banking and finance, pharmaceuticals, consumer goods, and even pari-mutuels. These projects ranged in size from 10 KLOCs for a Personal Data Assistant client, to more than 400 KLOCs

CONTINUED ON PAGE 30

Integrating Systems and Software Engineering: What Can Large Organizations Learn From Small Start-Ups?

Paul E. McMahon
PEM Systems

In an effort to integrate more effective systems and software engineering, many companies today are examining their internal processes. Recent research conducted on distributed development efforts may provide insight that could aid today's systems and software integration initiatives. Drawing material from his book, "Virtual Project Management: Software Solutions for Today and the Future [1]*," the author explores variations in large and small engineering organizations and presents an alternative view of large projects that may aid companies in their quest for more effective systems and software integration.*

Today, many companies are examining their internal processes in an effort to integrate more effective systems and software engineering. Many of the challenges faced in integrating systems and software engineering exhibit similarities to challenges observed on large distributed efforts.

In this article, engineering organizational variations are first explored, not to judge but to recognize the existence of variation and to note a common characteristic observed in all successful organizations regardless of size or structure. The article then discusses how the identified characteristic is achieved in different organizations.

This preliminary investigation sets the stage for a closer examination of what systems and software integration means in practice. An alternative view of a successful large project is presented that may challenge current published literature. The information presented in this article is the result of research initially conducted on large distributed projects [1].

Organizational Variation

If you were to ask a manager or senior engineer working today in a large advanced technology software-intensive organization to examine the chart in Figure 1, it is likely he (or she) would nod his (or her) head up and down, reflecting familiarity with the functional organizational structure and terminology employed on the chart.

Each rectangle on Figure 1 underneath the engineering manager is implemented through a department each with its own manager and pool of skilled engineers. At this level, similarity across diverse organizations is evident. Nevertheless, while many organizations have a similar top level structure, we have seen that inside these organizations implementation can vary greatly.

For example, in some organizations the Systems Engineering department is totally responsible for producing the software requirements specification (SRS). In other organizations, the Systems and Software Engineering departments collaborate on the production of the SRS with each producing specific *piece-parts* of the final SRS. We have also witnessed a third organizational variation where the Software Engineering department produces the complete SRS, while the systems group provides a review and approval role.

"... it is important to note that what we are asking engineers to do in departments by the same name, but in different organizations, can differ greatly."

Note that in all three cases described, the organizational chart referenced in Figure 1 could be used to describe the organizational structure. At the same time, it is important to note that what we are asking engineers to do in departments by the same name, but in different organizations, can differ greatly.

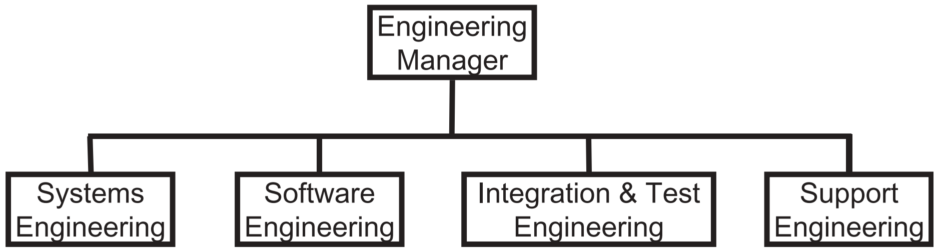
Common Successful Key Characteristics

It is not the intent here to judge the merits of particular organizational approaches, but rather to acknowledge their existence and to point out a key characteristic we have observed common to all successful organizations. In each case, when an organization functions successfully to produce an end product, individuals within that organization *understand and accept their specific role*. That is, they understand the organization's expectations of them.

This mutual understanding of roles, responsibilities, and expectations leads to operational efficiency with minimal duplication of effort. The successful organization appears from the outside to function as a single unit. Its piece-parts may vary on the inside, but in each case they come together without major surprises into a final integrated product.

It is worth noting here that in our experience we have found in many successful organizations that the *definition of* and *responsibility for* each piece-part is oftentimes not written down or described formally. It has been our experience working with large software intensive organizations with long histories of development and evolution that this knowledge may have been written down at some point in time but due to organizational evolution, its current state is most often passed on through less formal means.

Figure 1: *Traditional Functional Engineering Organization*



Communicating Expectations in Large Organizations

In large organizations we tend to see expectations communicated through structure and what we refer to as a process focus. While many large firms have over the past few years undergone organizational streamlining, our experience indicates that sizable written command media with phase-related exit and entry criteria continues to be relied upon.

The process employed inside many of these large organizations can be referred to as predictive, or repeatable. While written command media aids repeatability and communication, we have found that new engineers in many large organizations cannot rely totally on this written word to fully comprehend organizational expectations. Frequently, local cultures are also relied upon to aid communication of expectations.

Communicating Expectations in Small Organizations

Unlike many large organizations, small organizations most often see little structure, little process, and little established culture. The focus of most small organizations is on surviving. We find many small organizations to be heavily reliant on specific individuals. Given this situation, on the surface it would appear there is little a large organization could learn from a small one. However, let us take a closer look at the small organization.

The Small Organization Super Programmer Model

The communication of expectations in many small organizations is simple to describe. We refer to it as the *super programmer* model that implies a *do-it-all* expectation. The problem with this model is that, while expectations are clear, those expectations often lead to over-reliance on, and burnout of, individuals. We frequently find organizations that live by the super programmer model also live by the *code-and-fix* methodology.

During the past few years, we have known colleagues who have given up the relative security of the large, established organization in favor of the increased opportunity afforded by small start-ups. Unfortunately, many have also found that the demands of the small start-up require great personal sacrifice. While some have returned to the more *predictable* large corporate environment, others have found increased job satisfaction through an alternative small-company model that is rapidly gaining in popularity: the small team model.

The Small Team Model

Today, many small organizations are moving away from the super programmer model of operation in favor of a *small team* development approach. This change also often reflects a move away from a code-and-fix methodology, or no process, to what is referred to as an adaptive or lightweight process or method [2]. When we use the term *adaptive method* in this article we mean a method that supports rapid change initiated through small teams.

Lightweight processes can be thought of as *just enough* process, or process without a process-focus. Examples include eXtreme Programming (XP), which have been described by Kent Beck [3] and Jim Highsmith [4].

Characteristics of many lightweight processes include the following:

- Code and test focus.
- Continual iterative design.
- Pair programming.
- Continual planning and integration.

Upon first learning about the characteristics of XP, our reaction was, "This is fluff camouflaging a traditional code-and -fix methodology." However, after observing and interacting with a number of small teams embracing this approach, we have reached a different conclusion.

XP Fundamentals in Practice

While it is true that XP focuses on today, we have found that organizations that take this methodology seriously do not do so at the expense of planning. In fact, teams that follow this process often find themselves planning continuously. Planning in an XP environment is different from traditional planning conducted on many large projects. Plans are short in length, contain specific attainable goals, and often focus on periods of only a few weeks in duration.

This notion might not even sound like planning to those familiar with the process as currently implemented in many large organizations. It also might sound short-sighted and non-optimizing (not looking to the future for improvement), but the immediate feedback provided through short repeated cycles of planning and execution is proving to be effective from the perspective of the engineer in the trench.

This is the first fundamental difference we noticed when dealing with a small company employing an adaptive methodology. The second came when talking to software engineers working on an XP team: We found a surprising level of awareness and ownership of schedule and budget. The software engineers were aware of the schedule and budget and felt ownership of it because they had participated in its development. On the other hand, in many large organizations, we have found that it is not uncommon for engineers to have little insight into the project schedule and budget.

Upon first learning about XP, we found its code-focus to be difficult to accept. However, after observing an XP team in action, it left us with a different impression.

The notion that programmers using XP do not design is a misunderstanding. One member of a small XP team explained it to us this way: "Just because we focus on the code doesn't mean we don't give each task considerable forethought. We just don't write down the result formally, and we don't use a formal design tool." Then after he hesitated, he added, "But we might draw a sequence diagram or two, if we think it might help." Another member of the same team said, "We keep our designs as simple as possible, and we try not to spend any unnecessary time in design."

We had an opportunity to witness the design process in action with a small team, and it reminded us of an informal brainstorming session you might see in any organization. The meeting had not been scheduled ahead of time. It just happened because one member of the team wanted some help. Within a few minutes, three team members had gathered around a white board, and 25 minutes later there was a design solution sketched out on the board. We suggested that someone capture the diagram more formally.

Another interesting aspect of XP is pair programming. When we first heard about pair programming, we expressed agreement with the concept to a young client. Our thinking was that this technique would provide a backup in case one team member was pulled off the project or got sick. But the client was quick to explain that pair programming had nothing to do with having a backup.

We have since learned that some pair programming team members are adamant about having both team members present side by side during 100 percent of the programming activity. That is right! Not only does it take two people to complete one program, but also if one of the two is missing, in some cases, the other does not want to move on. Doesn't that sound incredibly inefficient?

But then the client went on to explain: "It's the dialogue that I don't want to miss. By having my teammate right next to me, it forces me to verbalize my thought process at the moment I type the code in. Often through this process, errors are detected at the same moment when they are about to be created."

As I listened to this process being described, a light bulb was flicking on in my head – this process is what peer reviews and early error detection was always meant to be!

Tailoring XP

When reading about a new methodology, you often envision something different from the way organizations actually apply it. XP, according to the book, includes 12 key practices. However, not all companies that claim to employ XP follow all the practices exactly as outlined by Beck [3]. For example, while many small organizations have recognized the value of pair programming, others have also recognized that what their engineers actually do extends beyond programming itself.

Often we find in practice that the pair recognizes that one of the members has more of a *systems* inclination (i.e., customer interface, requirements management), while the other prefers the traditional programmer role. These recognitions are usually based on individual strengths and desires. As a result, we tend to see a *systems focus* and a *software focus* being supported within the small team and small company environment, but without the formal system and software department boundaries that are prevalent in large organizations.

Effective Systems and Software Integration in Practice

We have witnessed large organizations communicating expectations through defined organizational structures, supported by *heavyweight* command media (policies, practices, and procedures). We have also seen the key role of culture in communicating expectations in large organizations.

In small organizations applying lightweight methodologies, on the other hand, communication occurs through short-range planning that leverages individual teammate strengths. Given what we have witnessed inside both large and small organizations, we are led to a question: "What does effective systems and software integration mean in practice?"

We believe that the answer to this question should be independent of the size and structure of the organization. We propose the following definition: Effective systems and software integration means that the right interactions are occurring at the right time, the right questions are being asked at the right time, and the right factors are being considered and acted upon at the right time.

Systems Engineering Inside Large Organizations

We have, on a number of occasions, taken the opportunity to ask an engineer in a large organization, "What is systems engineering?" Often the answer received has equated to, "whatever the systems engineering group does," in that particular organization.

Unfortunately, this answer can be problematic for two reasons. First, from an educational viewpoint, how are we to prepare systems engineers in our universities when the expectations of a systems engineer can vary dramatically from one organization to another?

Second, when task expectations are too tightly coupled to an organizational structure or department charter, the increased likelihood for tasks to fall through cracks exists. This is because no organization is perfect. We have also found that a tight coupling of task responsibilities to organizational partitioning tends to give rise to the "it's not my job" syndrome in large organizations.

Systems Engineering Inside Small Organizations

On the other hand, when we have asked, "What is systems engineering?" to an engineer who has experienced only the small organization environment, the most common response has been a puzzled look on his/her face. This is because in most small organizations, engineers do not think in terms of distinct systems and software tasks; rather, they think in terms of getting the job done.

One reason small organizations do not tend to exhibit the task-related difficulties we have observed in large organizations is because they have not artificially partitioned detailed tasking responsibility based on organizational boundaries. Stated differently, in small organizations that use adaptive methods, the team works out the task responsibilities knowing that together the team is responsible for everything.

Applying a Lightweight Approach to a Large Project

Published literature available today on lightweight methodologies [3, 4] indicates these approaches may not be scalable to large projects. While we do not recommend XP practices be applied in full on large projects, a number of the most successful large projects we have witnessed tend to already embrace many of these same practices. Although it is unwritten, i.e., not found in any formal corporate command media, a number of the most successful large projects we have observed also tend to exhibit an *unspoken adaptive subculture*.

Characteristics of these projects include incremental development, plans that focus on today, and a code focus. A code focus may seem unusual to a large project especially in large disciplined organizations, but in practice it has proven to be particularly effective when heavy reuse is involved. Testing candidate reuse code early, peer reviewing proposed changes early and often, integrating early, and staying integrated through a series of incremental builds have proven to be effective techniques on projects of all sizes.

A Key to Success

This article recommends managing a large project as a collection of small adaptive projects. This recommendation is not meant to imply that the adoption of such methods alone is sufficient to ensure large project success. On the contrary, if small teams within a large team were allowed to continually adapt their plan independently, then chaos would certainly result.

On one large project that we worked as a team member, the software work was partitioned across the country at three distinct sites. But before the work partitioning took place, a small group of senior project personnel established overall system architecture with very specific constraints, including computer platforms, compilers, tools, and interfacing requirements.

As the project evolved, there also evolved a number of small teams each with approximately seven to 10 engineers. Each small team had its own unique development issues. The project leader empowered these lower level informal small teams to make key decisions constrained only by the project requirements and the defined project architecture.

Many of the characteristics we have seen in successful small companies employing adaptive methods are also evident within small informal teams inside large projects. You will not necessarily find the small teams we are referring to on a *formal* organizational chart.

In-the-large XP practices can work, but only if implemented within the context of a higher level framework.

In short, we want our small teams inside large teams to take on increased responsibility, but the key to success on large projects is ensuring small team adap-

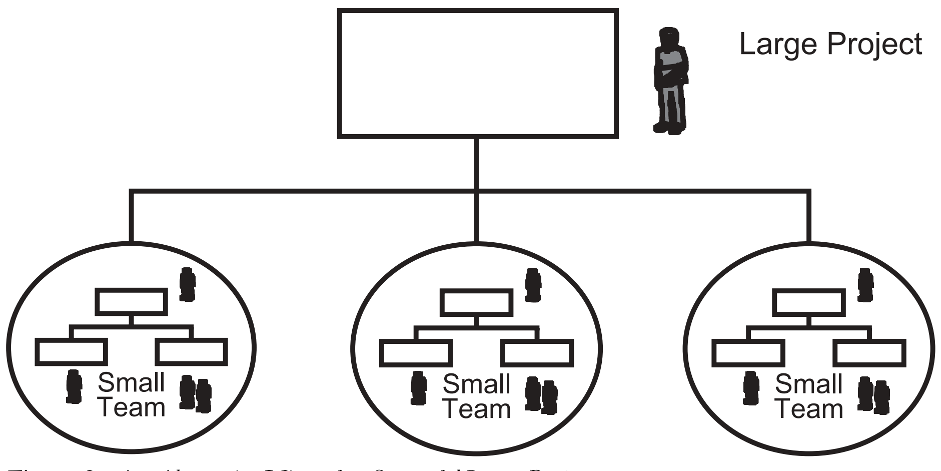


Figure 2: *An Alternative View of a Successful Large Project*

tations are consistent with well-defined, well-communicated system architecture.

An Alternative View of a Large Project

Through the use of a small architecture group and other related techniques to communicate architectural decisions and responsibilities [1], large projects can effectively be managed as a collection of small adaptive projects (Figure 2). While many large companies may not formally describe their operating structure in these terms, many successful large projects operate in this manner today.

It is also worth noting that we do not recommend that the notions of small teams and pair programming be overly formalized in large organizations. This would, in fact, undo the value we seek. Informality is an essential ingredient to the success of the small team in any organization.

Pair programming can be effective in an organization of any size, but it is also important to realize that some people just do not team well, and we do not believe that forcing small teams or pair programming makes sense in any organization.

Different companies have different cultures and differing past experiences and beliefs surrounding their workspace. Some believe in open workspaces, while others pride themselves in the private offices they provide their professional personnel. Nevertheless, we are witnessing a definite movement within the software community toward more group activities in support of increased productivity.

One new engineer in a large company told us that he sat in his cubicle for the first three months of his new job continually wanting to ask questions, but not wanting to be perceived as a nuisance. Other new engineers in the same company, we found out later, felt the same way.

Soon thereafter, an engineering lab environment was set up. It was not long before all the new software engineers were spending more and more time together in the lab. One engineer said the lab environment made it much easier to ask questions and to listen to answers to questions asked by others. Progress on the project increased at lightning speed shortly thereafter.

Conclusions

A few months after providing assistance to a small company utilizing XP, we asked one of their engineers a simple question: "Had anything changed over the past few months?" The engineer responded: "If I had to point to one thing, I've noticed that I'm spending less time dealing with urgent and unimportant matters, and more time on the things that count."

Adaptive methods not only can work in large organizations, we have found they are often key to large project success. We recommend large organizations that are not operating as effectively as desired consider the selective adoption of adaptive techniques.

What really first caught our attention with adaptive methods was the focus on the engineer in the trench. Too often, well-intentioned process improvement initiatives never seem to reach the real workers.

In reality, the short cycles of planning are not shortsighted, rather they are based on the length of time into the future where we have control over where we are going.

It is also important to note that it is not that the process we see today in large companies is not working, but it works at its own pace. Adaptive techniques and small informal teams inside large organizations can complement a formal organizational process focus, and can also be an effective method to facilitate change in an organization that is not evolving at the pace needed to remain competitive in today's world.

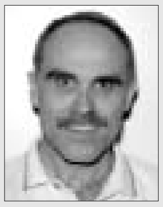
Write plans that work today, and do not discourage your team from continually updating their plans to reflect increased knowledge tomorrow. Encourage small teams to leverage your organization's strengths regardless of where those strengths lie inside your organization.

Small teams and adaptive methods can not only help your systems and software integration efforts, but they can also prepare your organization to be more effective in tomorrow's collaborative world.◆

References

1. McMahon, Paul E. Virtual Project Management: Software Solutions for Today and the Future. Boca Raton: St. Lucie Press, An Imprint of CRC Press LLC, 2001.
2. Fowler, Martin. "Put Your Process on a Diet." Software Development Magazine Dec. 2000: 32-36.
3. Beck, Kent. eXtreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.
4. Highsmith, James A. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. New York: Dorset House Publishing, 1999.

About the Author



Paul E. McMahon is an independent contractor providing technical and management leadership services to large and small engineering organizations. Before initiating independent work as PEM Systems in 1997, McMahon held senior technical and management positions at Hughes and Lockheed Martin. Today he employs his 28 years of experience to help organizations deploy high quality software processes integrated with systems engineering and project management. He has taught software engineering at Binghamton University in New York, conducted software process and management workshops, and has published more than 20 articles and a book on virtual project management.

118 Matthews St.
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org

Highpoints From the Agile Software Development Forum

Pamela Bowers
CrossTalk

There is much confusion in the software industry about what agile software development is and is not, and what it implies. This article reports on the keynote talks at the "Creating Competitive Advantage Through Agile Development Practices" technology forum held at Westminster College in Salt Lake City in March. Nearly 110 attendees at this initial annual forum participated in work sessions and networking breaks, and heard speakers and panel discussions about increasing return on investment, decreasing time to market, increasing innovation, and more.

Agile software development is not new. It has been around since the beginning of software development, but did not show a competitive advantage in the 1970s and 1980s, said Alistair Cockburn, a consulting fellow at Cockburn and Associates. However, it did win the development races in the turbulent 1990s, he said, and methodologies that began appearing in 1993-95 were Rapid Application Development, eXtreme Programming (XP), Scrum, Dynamic Systems Development Method, Crystal, and adaptive.

Cockburn was one of the speakers at the "Creating Competitive Advantage Through Agile Development Practices" technology forum held recently at Westminster College in Salt Lake City. More than 150 attendees learned about agile software development and networked with peers during breakout sessions. The Westminster College Gore School of Business and Wasatch Digital IQ co-sponsored the forum.

In his talk, Cockburn explained that agile software development is about putting value on "maneuverability." He said, "Its [agile development] being able to respond quickly became its advantage, however, agility is a value statement." Cockburn stressed that agile is not appropriate for every project. Some projects want predictability and repeatability, and cost and agility go against each other, he said. "With agile, it costs more to deliver products faster. There are trade-offs, and each project will make its own appropriate value decisions."

Jim Highsmith, director of Cutter Consortium's Agile Project Management Advisory Service, cited two main reasons for the popularity of agile software development today. "Agile addresses a problem domain where speed and flexibility are paramount," he said. "And it addresses a culture and workplace we would like – that Dilbert would like to work in – a community."

Cockburn noted that agile methods make greater use of the following: individuals and interactions, working software, customer collaboration, and responding to change. Plus, agile means different techniques in different situations, he said. "Within agility, different tactics fit different situations. Agile is not just a re-titling of eXtreme Programming."

Highsmith noted that agile works best in "exploration" environments. He compared it to drilling for oil: Drilling to known oil reserves requires very different techniques to be cost effective than does exploration drilling to find oil. In each case, projects are managed differently and success is measured differently, he said. "Agile software development finds a way around problems to get successful projects."

"Agile software developers are not hackers. Agilists plan regularly, test according to project priorities, and re-check results with users often. They talk to each other and customers as a matter of practice."

Highsmith defined an exploratory project in the software world as "one to complete large projects that are both frontier (research-like) and mission critical in a turbulent business and technology environment. The characteristics include early product release, high customer involvement, and frequent testing."

What led Symantec to move to XP was not delivering the product that its customer needed, said Russell Stay, vice president of Product Delivery. The company was using a modified Waterfall technique consisting of up-front design then execution. Tight project management resulted in delivering the product on time and in budget, said Stay, but it was the wrong product. "We needed to adopt a new process."

Agile software development is a resource-limited cooperative game of invention and communication, said Cockburn. The key is adapting to reality with players – people – who are non-linear, unpredictable, spontaneous, and bring weaknesses and strengths to the game, which never repeats itself, he said. "Software succeeds when people notice errors and have enough pride in their work to step out of their job description to see that it gets fixed."

An important consideration, said Cockburn, is that people communicate most effectively interactively, i.e., face to face. "The richest form of communication is two people at a white board. The least effective form is on paper." Much is said in the communication that occurs with body language, voice inflection, facial expression, etc., he said.

Added to this is the fact that the project methodology gets restructured around the ecosystem details, which are always changing. The key is to pair workers who complement each other to achieve the desired results. For example, said Cockburn, if "Bill" only has the patience to take a project through the requirements stage, then he should be teamed with "Mary" who excels in implementing the process through to project completion, he explained. "This way, information gets from the marketplace to the programmers."

"Gone are the days of the saviors and cowboys," said Stay. Pair programming was stressed up front when Symantec began its agile implementation. Stay said that he was willing to accept a 15 percent attrition rate due to this change. However, after giving it a try, he said that fewer than 8 percent of

Successful Methodology Ensures Reuse

In the real world, organizations can mandate that Personal Software Process^{SM} (PSP^{SM}) be used on a project and install the trainers to make sure it is used. However, software developers can either refuse to use PSP, or find many ways to subvert it, warned Alistair Cockburn, a recognized expert on software project management in an interview with CROSSTALK.

"Since 1991, my views on methodology have been that programmers can at anytime opt not to use this [process] either overtly or covertly," said Cockburn, a consulting fellow at Cockburn and Associates. "Therefore, the definition of the successful methodology for me includes that the people agree to use it the next time."

Cockburn said that he is looking for the way [methodology] that "puts the least requirements for consistency and discipline on software teams, yet beats the odds." He pointed to Crystal as a solution. There are three core principles in Crystal that make it successful. First, it works in increments, which allow you to recover from almost any catastrophe. Second, Crystal calls for reflection after every increment to discuss what to keep and what to change, which develops a process that adapts to change. Third, the team must tailor itself to create its own process. He added that Crystal also has a strong emphasis on personal communications, tacit knowledge, close worker proximity, and frequent delivery of running deliverables. It is all these elements, Cockburn said, that allow you to "beat the odds."

When asked, "What are the three things that most ensures agile success?" Cockburn said that experienced management is the No. 1 factor. "You need to have a project manager who is alert, sensing whether something is right or wrong, and with enough experience to steer the group. Second is having access to real users; developers need reliable information for requirements, to have someone handy to show results, and to get feedback on a reliable basis. Third most important is physical proximity. It is important to have people close enough together that they can talk to each other, he said.

Cockburn also advises management to pay attention to their fears. "They could be well founded." The key is to find out which fears are unfounded, he said. For example, he pointed to the fear of "hacking." In XP, the process check is that there are always two people working together, so it's a lot harder for one of them to hack, he said. Programmers also write their acceptance check before they write actual code, and this takes a lot of thinking, he added. A final rule is that any two people sitting together can change anybody else's work, as long as they agree, he said. "Call it common ownership."

With agile development, said Cockburn, "Programmers cannot just say, 'Go away and leave us alone'. Agile takes collaboration among project managers, users, programmers and testers. There is no privacy in the code."

employees left the company.

Stay stressed that it is important for the team to buy into the agile method and that coaching should be brought in house. Co-residence is also critical so that leaders and developers can work side by side, he said. "Individual office cubes get used less and less. Our average use is about one hour per day. The rest of the time, the development area is the hub of activity."

Unfortunately, the agile message is often misconstrued, said Cockburn. "Agile software developers are not hackers." Unlike hackers, agilists do plan regularly, he said. Agilists test according to project priorities, and re-check results with users often. They talk to each other and customers as a matter of practice. They expect management to provide priorities and to participate jointly in making project adjustments.

Stay concurred. The process at Symantec is "highly managed but flexible," he said. They use the "queing" theory of breaking down the process into small parts; iterations are done biweekly. Also, exit criteria and test procedures are defined first, before writing code, he said, and test automation is a priority.

It is also not true that agile only works with the best developers, Cockburn said. The critical success factor is to have at least one experienced and competent lead person, who can then carry four or five "average or learning" people. With that skill mix, agile techniques have been shown to work many times when the deck is "stacked" as follows:

- Hire good people.
- Seat them close together to help each other out, close to customers and users.
- Arrange for rapid feedback on decisions.
- Let them find fast ways to document their work.
- Cut out the bureaucracy.

"Agile software development has a lot to do with how much trust and communication is set up," said Cockburn.◆

Agile Before Agile Was Cool

Gordon Sleve
Robbins Gioia LLC

Success can be achieved by many means. Sometimes it is obvious which road to take, other times it does not really matter. Look at individual circumstances before choosing one path over the other.

People go years, possibly their entire lives, exhibiting certain behaviors, sometimes knowingly but often unaware of any notable pattern. Sometimes an event will occur where a name is given to their particular behavior. A man who takes his work problems out on his family may be in an anger management class and be informed that he exhibits *misplaced aggression*. For a woman whose husband is an alcoholic, yet buys liquor for him, might be labeled an *enabler*.

This identification can lead to an epiphany for the subject. This sudden realization is exactly what happened to me when a colleague of mine inquired as to my willingness to write this article on agile programming.

I had never before heard the term agile programming, but my associate was familiar with the concept and also with my work, so I was willing to trust in his judgment. During research into the concept of agile programming, it quickly became evident that this was an acceptable label for the coding practices I have used routinely for more than 13 years.

Unwitting Agile Programmer

In my work as an analyst for a program management firm, I use a fourth generation language (4GL), control and analysis tool to build reports and graphs. My customers and I use these documents to perform analysis on schedule related data. The information derived from this data is used to point out past mistakes and potential problems, thus saving both time and money. This is my goal as a program management analyst, and the goal of my firm, to make our *customers successful*.

Occasionally I am called upon to develop related applications or modules using 4GL. Some applications are written to analyze existing data and some to capture new data. The latest major development effort involved a resource-forecasting tool used to project future work requirements, and provide *what-if* scenario modeling. Because the customer wanted to keep the existing analysts at the site working on their current assignments, a separate contract was written and programmers were hired to perform the work. This project followed a traditional software development methodology because this methodology worked for this project. The project finished on time and under budget.

In 1994, the Air Force was awarded the Navy FA-18 programmed depot maintenance (PDM) contract. This was historic in that it was the first time that one branch of the armed forces was contracted to repair a weapon system used by a different branch. The accepted proposal called for the work to be performed at Hill Air Force Base (AFB) in Ogden, Utah.

Since the fall of 1993, I had been working as a program analyst on the Programmed Depot Maintenance Support System contract in the Aircraft Directorate at Hill AFB. When the new workload arrived, the program management team provided precedence network schedules and tracked the work against the plan, as had been done for the other aircraft work at Hill AFB.

Not long thereafter, the Aircraft Directorate was audited by an independent audit agency. Their findings required Hill AFB to provide an auditable *unplanned work* approval tracking system. The F-18s were brought to Hill AFB for a PDM, which is basically an overhaul of the entire aircraft, according to a specific set of operations to be performed, called planned operations. There are also provisions for problems encountered either during aircraft inspection or while the mechanics are performing planned work. These problems are defined as unplanned work and require extra time not accounted for in the planned operation package.

Given that the FA-18 is a Navy weapon system and each aircraft has spent a good deal of time on aircraft carriers or at coastal Naval Air Stations, much of the unplanned work is corrosion removal. This unplanned work accounted for more than half of the total hours on a FA-18 PDM. The independent audit agency required a way to show that hours billed to this contract workload were not being used to work on F-16s or C-130s, which were located in the same hangars as the FA-18. Without such an auditable system, the workload would be pulled from Hill AFB and given back to the Navy.

The Aircraft Directorate quickly established a manual process that may have satisfied the requirements set forth by the independent audit agency. The only worry was, with hand carrying of thousands of documents to and from the hangars, some were bound to get lost and therefore the system might fail the audit.

An Air Force major working in the Aircraft Directorate approached our firm about automating this process. It was easy to show a good potential return on investment (ROI) based on the amount of man-hours involved in processing work cards in the manual system. This also fell into the scope of our firm's program management charter. Automating this system would provide the ability to add the hours generated by this unplanned work into the schedule as soon as the requests were approved, thereby extending the schedule in a real time fashion. Since all parties were in agreement that this was a *win-win* situation, the next decision was how would we proceed with development?

To bring in more analysts would require writing a new contract or making an amendment to the existing one. Since the customer wanted the system in place by the time the audit agency returned, this option would take too long. They decided instead to reallocate the existing resources of the program management team and place all three analysts on full-time development of the new application. Due to the time constraint, there was no development of a formal plan or extensive requirements, which led to the use of methods now described as agile.

A Perfect Agile Fit

Traditional programming methodologies were put in place mainly to prevent *requirements creep*. In agile programming, potential for these problems is diminished by getting the application to the users quickly. Surely if it takes two years to develop an application, changes will arise in the organization or process that will drive new requirements and cause delays. Agile methodologies exist that are specifically tailored to long-term projects, but my experience has been with short-term projects. The unplanned work module was not extensive and we were confident that we could provide a quick turnaround. Even though we did not have agile methodology guidelines to go by, this project was a perfect candidate for just such a philosophy. The Agile Software Development Manifesto states:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while there is value in the items on the right, we value the items on the left more. [1]

Although we did not know of agile methodologies, in retrospect we practically followed them to the letter. We focused almost entirely on those *items on the left* and for all intents and purposes, ignored the *items on the right*. In this case, circumstance rather than conscious thought led us to perform in this manner. We were under the gun to get a working product in place in a short amount of time. There was little time for planning, contract negotiation, or documentation.

The lack of planning shows a slight departure from agile, but remember, we did not have the agile methodology with which to work. The available resources and their expertise locked in our tools. Due to the fact that very little of the traditional methodologies were available to us, we were forced to draw heavily from what is now an agile approach. I like to look at these two approaches as the Scarecrow and the Tin Man, agile being the Scarecrow (flexible) and traditional being the Tin Man (rigid.) They both want to get Dorothy to Oz, but they each have unique abilities and weaknesses. In our case, we had to rely on a Scarecrow named "agile."

The initial requirements were to automate the manual process and add reports. The bulk of the detailed requirements were obtained during development by working closely with civilian counterparts involved in the unplanned work process. These individuals were planners, schedulers, and the approval authority, each with knowledge of how their piece of the process worked.

We met daily, but on an informal basis, usually with the point of contact (POC) most familiar with the section being worked in a one-on-one setting. This would almost always take place at the developer's desk. We showed progress from the previous day and verified with the POC that their requirements were being met. Then we would identify any changes that needed to be made and start working to that end. As we moved through the development phase, we would identify those *nice to have* features and record those requirements for follow-on work. Through this process the programmer learned much more about his customer's needs than he could by reading thousands of pages of requirements documents. It also showed a commitment to individuals and interactions over processes and tools.

As changes were made to the application, corroboration was sought from the user before continuing further. Sometimes the user would sit through several iterations of code changes right at the programmer's desk, commenting and critiquing. By working closely with the user community, there was very little need for documentation and training. By the time development was completed, the users had been trained through their constant involvement. There was no need for a formal training class after implementation. Since we knew there would be a good deal of follow-on changes, we decided to wait on documentation.

The main application was completed with great success in plenty of time for the next audit. The only cost associated with the program was the temporary loss of productivity toward the original program management objectives. Our team was awarded a certificate of appreciation from the Aircraft Directorate citing a savings of 250 direct/indirect man-hours per aircraft. There was a dollar figure placed on both the cost (approximately $28,800) and the savings (approximately $1 million) resulting in an actual ROI of more than 30 to 1.

Given the limited planning work, there was no way to predict such a fantastic windfall. However, had more time been spent in planning, documentation, contracting, and processes the ROI ratio would have been less impressive. Follow-on changes included expansion for both F-16 and C-130 workloads, and a *master write-up* module that provided the users with a pick list of frequently used write-ups.

Specified Successful Use Continues

Although the F-18 contract only lasted one year before the workload was returned to the Navy, the use of this application on the F-16 and C-130 programs continues to this day. This program is still in a textual interface format but will soon be converted to an Oracle/Web-based format to provide better accessibility and ease of use. The user community has taken ownership of this application and they like it because of that very fact; it is their own creation. From time to time, minor bugs appear, which will be the case when omitting configuration management and strict coding practices, but the users are happy with that trade-off for flexibility.

One might argue that agile software development *flies in the face* of program management. While the irony of a program management professional touting "responding to change over following a plan" is not lost on me, I see both methodologies coexisting and filling an important purpose: to *make our customers successful*.

Yes, I am a proponent of agile programming but only in those cases where it is the best solution. Unless given a specific set of circumstances, I cannot say which is best. I have experienced success with both agile and traditional software development methodologies, and success is the ultimate goal.◆

References

1. AgileAlliance. "Agile Software Development Manifesto." 13 Feb. 2001 <www.agilemanifesto.org>.

About the Author

Gordon Sleve is a senior program analyst for Robbins Gioia LLC working in the ICBM program office at Hill Air Force Base (AFB) in Ogden, Utah. He was previously a site manager at Letterkenny Army Depot in Chambersberg, Penn. Sleve was selected as Robbins Gioia's Senior Program Analyst of the Year for Dayton-based operations in 1995 for his support of the implementation of Programmed Depot Maintenance Support System at Hill AFB.

Robbins Gioia LLC
OO-ALC/LMSO
6014 Dogwood Ave.
Bldg. 1258 Rm. 14
Hill AFB, UT 84056-5816
Phone: (801) 775-5943
Fax: (801) 586-4835
E-mail: gordon.sleve@hill.af.mil

Should You Be More Agile?

Rich McCabe and Michael Polen
Software Productivity Consortium

Agile software development techniques are an effective response to many problems still plaguing development projects. Although there are a number of issues to consider, almost any project can become more agile to its benefit. What exactly does it mean to be more agile? Words like predictable, cost-effective, and mature are more often used to characterize desirable software development processes. Agile development has come into focus recently due to the popularity of its most widely known interpretation, eXtreme Programming, but some of its foundations go back as far as 20 years. This article addresses some of the questions about agile: What is agile? Who needs to be agile? How can any project not creating small business applications seriously consider agile development? Is agile development an "all or nothing" proposition?

Agile Development: Weed or Wildflower?[1]

David Kane
SRA International

Steve Ornburn
GBC Group, Inc.

The Software Engineering Institute's Capability Maturity Model® (CMM®) has been a major force for software process and acquisition improvement in the federal government's civil and defense communities for the past decade. Major investments have been made by the government, their contractors, and many other organizations to make software development more consistent and reliable. The CMM provided an alternative to the cowboy programmer archetype. Amid this backdrop of progress, a new trend in software development has emerged – agile development, which aims to build software faster and more flexibly than traditional approaches. Agile values "individuals and interactions over processes and tools" [1]. *For organizations that have invested in a CMM, do agile methods represent the rebirth of the cowboy – a weed to be stamped out? Or, are agile methods a reasonable way to build software in a world in which needs are changing at an ever-increasing pace – a wildflower to be nurtured? This article looks at whether there is a home for agile methods in communities that have have embraced the CMM.*

Editor's Note: Due to space constraints, CROSSTALK was not able to publish these articles in their entirety. However, they can be viewed in this month's issue on our Web site at <www.stsc.hill.af.mil/crosstalk> along with back issues of CROSSTALK.

CONTINUED FROM PAGE 21

for the industrial automation project described herein. The languages were mostly C++ but also included C, HTML, VB, and SQL. Productivity ranged from a low of 21 to a high of 48 lines of code per coding hour, averaging 35 lines of code per coding hour.

Compared to projects conducted before adopting this intensely practical and agile software development discipline, our cost per line of code and defect rates were drastically reduced while our development velocity was significantly increased. Our most recent audit revealed an *overall* average productivity index of 22 [4]. This index is a management scale corresponding to the overall process productivity achieved by an organization during the main software build. An index of 25 is considered among the highest ever recorded.◆

References

1. Beck, Kent. eXtreme Programming Explained: Embrace Change. Boston: Addison-Wesley, 1999.
2. Fowler, Martin, et. al. Refactoring: Improving the Design Of Existing Code. Boston: Addison-Wesley, 1999.
3. C3 Team. "Chrysler Goes to Extremes." Distributed Computing. Oct. 1998 <www.xprogramming.com/publications/distributed-computing .html>.
4. Putnam, Lawrence H., and Ware Myers. Measures of Excellence: Reliable Software on Time, Within Budget. Upper Saddle River: Prentice Hall/Yourdon Press, 1992.

Note

1. "We" as mentioned throughout this article refers to the Geneer company.

About the Author

John Manzo has spent more than three decades of his career in software engineering, and has contributed to and made significant accomplishments in the development of software, computer, and telecommunications solutions. Manzo comes to AgileTek from Geneer where he was chief technology officer, and brings with him a legacy of broad and deep experience in agile development methods. Earlier in his career, Manzo was recognized for his development of the Fire Control software for the Navy's highly successful AEGIS system – one of the largest and most complex software developments ever delivered to the Department of Defense. He has served as a representative to the President's National Science Advisory Board, and served as an adjunct faculty member of Harvard University where he developed, and for several years taught, "The Management of Software Engineering."

AgileTek
934 South Golf Cul de Sac
Des Plaines, IL 60016
Phone: (847) 840-3765
Fax: (847) 376-8308
E-mail: jmanzo@agiletek.com

The 12-Step Program for Software Weight Watchers

In the February issue of CrossTalk, I sensed an intellectual melee in the air. Six months later, software publications and conferences were abuzz. In an industry where plans are inadequate and planning is essential, we are starting to question the strong hold of predictive, process oriented, model-based software development.

Grassroots intrigue with lightweight methods like eXtreme Programming (XP) and Scrum has triggered software developers to question their approach, methods, and focus. Although questions currently outweigh answers, this debate has broken the assiduous fixation on such heavyweight methods like the Capability Maturity Model® Integration^{SM} (CMMI^{SM}) and Software Process Improvement and Capability dEtermination.

I am, however, concerned for individuals, teams, and organizations that are ensnared and laden by incompatible methods or approaches. For some these processes and methods have become addicting and will be hard to break.

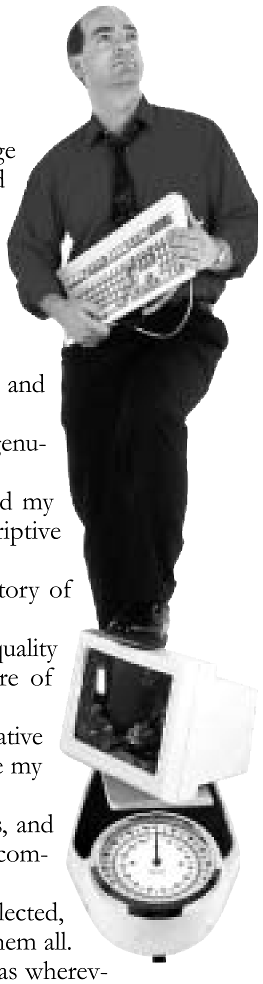
For those who are troubled, I offer the time-tested 12-Step program used by millions of people to successfully transform their lives and recover from obsessive-compulsive behaviors. With modification, I hope these steps successfully amend software development habits and aid in the recovery from career threatening behaviors.

If you are clinging to the sanctuary of models, processes, and theoretical control, and find these heavy methods too restrictive for your small-to-medium projects, unstable requirements, competent team, and short deadlines then repeat after me: "I, (state your name) am a heavyweight zealot. I promise to follow the 12 Steps for Heavyweights."

12 Steps for Heavyweights

1. I admit I was powerless over predictability, and my life had become unimaginative.
2. I believe that a power greater than a process, model, or best practice could restore my sanity.
3. I made a decision to turn my will and my career over to the care of vigilance, acumen, proficiency, and ingenuity.
4. I searched and made a fearless inventory of my respect for colleagues and customers.
5. I admit to customers, myself, and to colleagues the exact nature of my wrongs.
6. I am entirely ready to have pair programming, coding standards, and iterative deliveries remove my defects.
7. I humbly ask my customer, collaborative colleague, and test programs to reveal my shortcomings.
8. I made a list of customers I harmed, and I am willing to make amends to them all.
9. I will make amends to customers wherever possible, except when to do so would injure them or others.
10. I will continue to take personal inventory, and when I am wrong will promptly admit it and refactor.
11. I will seek through collective ownership to improve my conscious contact with customers asking only for knowledge of their will for me and the power to carry that out.
12. Having had a creative awakening, I will carry this message to heavyweight zealots and practice these principles in all my affairs.

If your proclivity for collaboration, rapid development, and empirical control has led you to the fast paced world of XP, Scrum, Crystal, adaptive software development, feature-driven development, or dynamic systems development method, and you find these light methods are inadequate for your large project, dependable requirements, and motley team then repeat after me: "I, (state your name) am a lightweight infidel. I promise to follow the 12 Steps for Lightweights."



12 Steps for Lightweights

1. I admit I was powerless over change, and my life had become unmanageable.
2. I believe that a power greater than ingenuity could restore me to sanity.
3. I made a decision to turn my will and my career over to the care of prescriptive processes and best practice models.
4. I searched and made a fearless inventory of my process areas and maturity.
5. I admit to my manager, SEPG, and quality assurance department the exact nature of my wrongs.
6. I am entirely ready to have quantitative management and optimization remove my defects.
7. I humbly ask assessments, inspections, and quality assurance to reveal my shortcomings.
8. I made a list of all process areas I neglected, and I am willing to make amends to them all.
9. I will make amends to my process areas wherever possible, except when to do so would injure the budget, in which case I will request a waiver.
10. I will continue to assess my maturity and when I deviate, I will promptly conform.
11. I will seek through documentation and meetings to improve my conscious contact with "The Model" asking only for knowledge of its will for me and the power to carry that out.
12. Having had a disciplined awakening, I will carry this message to lightweight infidels and to practice these principles in all my affairs.

Since the majority of the industry has considerable investment in heavyweight methods and will likely dismiss the agile movement as a return to callow software programming, I offer you the following athletic afflatus.

In planning for one of the most grueling events, the Tour de France, Lance Armstrong starts preparations a year in advance. He does not prepare for a specific race but instead prepares for several race scenarios and his ability to adapt to them. That is important because as soon as the race starts, the event will take its own course and any planning will be history. He knows the course, conditions, and competitors are unpredictable and his success depends on knowing what is going on and responding quickly.

Software development, although not the Tour de France, is far from predictable and would benefit from the insight of one of the greatest athletes in the world. Although not a panacea, these pellucid ideas, if allowed to imbue the mind, will ameliorate your software organization. Go ahead. Break from the peloton.

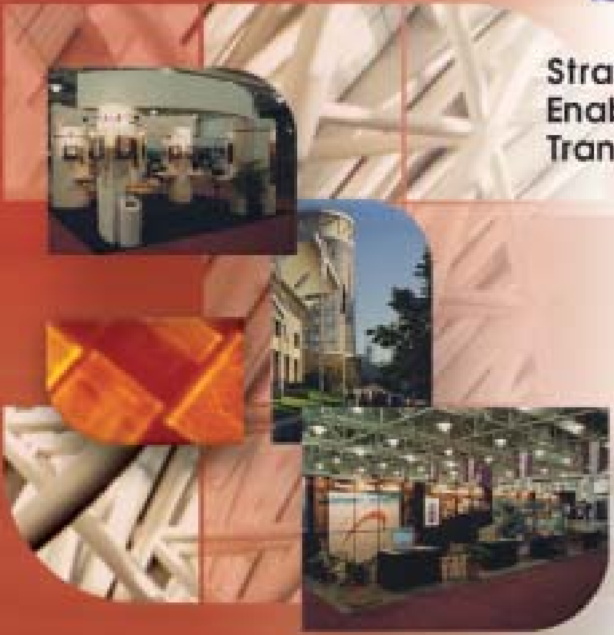
— Gary Petersen
Shim Enterprise, Inc.

The Fifteenth Annual

Software Technology Conference

28 April - 1 May 2003 • Salt Lake City, UT

STC 2003



Strategies & Technologies: Enabling Capability-Based Transformation

Reasons to Exhibit at STC 2003

- Participate in the premier software technology conference - endorsed by the Department of Defense (DoD)
- Interact with over 2,500 leaders and decision makers in the DoD, Government, Industry, and Academia
- Choose your level of involvement by becoming a platinum, gold, or silver trade show sponsor

Register to exhibit today!
www.stc-online.org
or 800-538-2663

Source code: CT3

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

Published by the Software Technology Support Center (STSC)

CrossTalk / MASE
7278 4th Street
Bldg. 100
Hill AFB, UT 84056-5205

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737