CUTTER IT JOURNAL
The Journal of Information Technology Management

CUTTER CONSORTIUM
ACCESS TO THE EXPERTS

December 2001 Vol. 14, No. 12

The Great Methodologies Debate: Part 1

Resolved
Traditional methodologists are a bunch of process-dependent stick-in-the-muds who'd rather produce flawless documentation than a working system that meets business needs.

Rebuttal
Lightweight, er, "agile" methodologists are a bunch of glorified hackers who are going to be in for a heck of a surprise when they try to scale up their "toys" into enterprise-level software.

"Today, a new debate rages: agile software development versus rigorous software development."

Jim Highsmith, Guest Editor

Opening Statement
Jim Highsmith 2

Agile *Can* Scale: Inventing and Reinventing SCRUM in Five Companies
Jeff Sutherland 5

Agile Versus Traditional: Make Love, Not War!
Robert L. Glass 12

Business Intelligence Methodologies: Agile with Rigor?
Larissa T. Moss 19

Agility with the RUP
Philippe Kruchten 27

Extreme Requirements Engineering
Larry Wagner 34

Exclusion, Assumptions, and Misinterpretation: Foes of Collaboration
Lou Russell 39

opinion

Opening Statement

by Jim Highsmith

In the early 1980s, I participated in one round of methodology debate. Structured analysis and design champions such as Tom DeMarco, Ed Yourdon, and Tim Lister were on one side of the debate, while data-driven design aficionados like Ken Orr, Jean-Dominique Warnier, Michael Jackson (not the singer), and myself were on the other. It was debate, but a collegial one. Although we argued, we did so out of mutual respect and the desire to further the cause of software development. Tom, Ed, Tim, Ken, and I now serve together on the Cutter Business Technology Council — still debating.

There have been a variety of similar debates over the last 25 years — data-driven versus process-driven approaches, information engineering versus structured development, relational database design versus object database design, object everything versus non-object everything. Each of these debates was useful in advancing the cause of better software development. Whatever one's position, having to listen and respond to thoughtful discourse from the other side helped each side clarify its position.

Today, a new debate rages: agile software development versus rigorous software development. Agile approaches (Extreme Programming, Crystal Methods, Lean Development, Feature-Driven Development, Adaptive Software Development, SCRUM, and Dynamic Systems Development Methodology) populate one camp. The Software Engineering Institute, with its Capability Maturity Model (not considered a "methodology" per se by the SEI, but so considered by many others), and Rational Software, with its Rational Unified Process (RUP), populate the other. But even these lines are obscure. In the next few months, I am participating on two panel "debates," one ostensibly on "agile" versus RUP and another ostensibly on agile/RUP versus the CMM. There are articles about light — or agile — RUP. It seems that no one is willing to concede "agility" to the other side.

Proponents of XP, SCRUM, and other agile approaches have argued endlessly for their values and practices — often stridently, as have proponents of the CMM and the RUP. The point is that spirited debate and even some in-your-face jabs are good for software development. Healthy, collegial debate helps everyone think through the issues more clearly. Only by defining and debating will others be able to understand the similarities and differences and be able to apply the right mix to their own organization. Both the SEI and Rational have made wonderful contributions to software development, but it is important to understand the similarities and differences between them and the self-professed agile approaches. Only by purposeful debate will organizations be prepared to decide which approach, or mix of approaches, matches their particular culture and problem domains. Debates become heated at times, but that's an important part of a good debate. If people aren't passionate about what they believe, why bother?

One of the great services that Extreme Programming proponents are performing for software development is their determination to push "extreme positions" — for example, no documentation, no up-front design, limiting work weeks to 40 hours, and communal code — and *really* radical positions such as frequent testing of one's own code. By taking these positions and clearly defining the extremes, XPers have launched this healthy debate within the software development community. If Kent Beck and others had advocated moderate positions,

 ©2001 Cutter Information Corp.

everyone could just say, "Well, we really do just about the same thing." If Extreme Programming had been named Moderate Programming, it would not have generated such debate.

Jeff Sutherland brings up one of these key "extreme" points around which debate should ensue. In his article, Sutherland challenges the assumption "that software development is a predictable, repeatable process." I think this is one position that helps *distinguish* agile development from traditional development. It challenges one of the most fundamental assumptions upon which many project management and software development methodologies are built. It attacks the premise that plans, designs, architectures, and requirements are predictable and can therefore be stabilized. It attacks the premise that processes are repeatable — the very foundation upon which traditional, rigorous methodologies are constructed. It is a fertile area for debate and discussion.

When the arguments are all about mushy, moderate, middle-of-the-road positions, then others have little information from which to analyze and draw their own conclusions. Spice and diversity are vital to useful debate. But so are thoughts from those who moderate the debate, who emphasize the similarities between positions, who say, "Well, they may not be as far apart as we first thought." In his wonderfully conciliatory article, Bob Glass suggests that we "Make Love, Not War." Glass, who has a long history of addressing the yin and yang of issues in his work, analyzes each of the Agile Manifesto values and principles. His analysis indicates the pros and cons of each of them, and he then indicates which "side" he thinks is more correct. The final tally? "Agile: 8, Traditional: 5, Tie: 3."

Larissa Moss' article describes the balancing act that most people and their organizations go through. Moss says, "The debate about rigor versus agility should not be about the choice between them, but rather about creating methods and guidelines for merging 'just enough' of both." Moss' article is about synthesizing rather than compromising — and she illustrates this synthesis through two wonderful phases — "the thrills of chaos" and the "dregs of structure." Moss reports that a dot-com group she worked with "did not reject structure per se, only the *rigor* that accompanied it and only when imposed by other people, such as their management." She comments, "The most valuable lesson learned from my dot-com experience is that agility and rigor not only *can* complement each other — they *should*."

In his article on the Rational Unified Process (RUP), Philippe Kruchten discusses the idea of tailoring, arguing that through customization, the RUP can adapt to a wide range of projects. He asks which is easier: "Starting from a blank slate, with a few key principles, and then building the

Cutter IT Journal®

Cutter Business Technology Council: Rob Austin, James Bach, Tom DeMarco, Jim Highsmith, Tim Lister, Ken Orr, Dick Nolan, Ed Yourdon

Editorial Board: Larry L. Constantine, Bill Curtis, Tom DeMarco, Peter Hruschka, Tomoo Matsubara, Navyug Mohnot, Roger Pressman, Howard Rubin, Paul A. Strassmann, Rob Thomsett

Editor Emeritus: Ed Yourdon
Publisher: Karen Fine Coburn
Managing Editor: Karen Pasley
Production Editor: Linda Mallon
Client Services: Christine Doucette

Cutter IT Journal® *(ISSN 1522-7383) is published 12 times a year by Cutter Information Corp., 37 Broadway, Suite 1, Arlington, MA 02474-5552 (Tel: +1 781 648 8700, or, within North America, +1 800 964 5118; Fax: +1 781 648 1950 or, within North America, +1 800 888 1816; Web site: www.cutter.com/consortium/).*

Cutter IT Journal® *covers the software scene, with particular emphasis on those events that will impact the careers of information technology professionals around the world.*

©2001 by Cutter Information Corp. All rights reserved. Cutter IT Journal® *is a trademark of Cutter Information Corp. No material in this publication may be reproduced, eaten, or distributed without written permission from the publisher. Unauthorized reproduction in any form, including photocopying, faxing, and image scanning, is against the law.*

Subscription rates are US $485 a year in North America, US $585 elsewhere, payable to Cutter Information Corp. Reprints, bulk purchases, past issues, and multiple subscription and site license rates are available on request.

process from the bottom up? Or starting from a rich knowledge base and choosing, shrinking, modifying, and evolving existing recipes to fit the problem at hand?" Kruchten describes the concepts behind the RUP and characterizes it as an open framework with a rigorous underlying process model. "Some of its detractors call RUP a 'heavyweight' process and depict it as a behemoth that forces you to do zillions of useless and unnatural things," he says. "We see it more as a rich palette of knowledge from which to choose what you need."

Larry Wagner's article broaches the subject of agility versus rigor from an XP and a CMM perspective, using requirements engineering as the topic for comparison. Wagner uses his extensive experiences with clients to discuss when he thinks the XP practice of user stories is appropriate and when he believes that other techniques would be more suitable. Addressing the current debate, he emphasizes "what you can leverage from the controversy."

Finally, Lou Russell's article reminds us of the need, whatever the debate, to include and embrace diversity — whether in ideas or culture or gender. It is somewhat ironic that the Agile Alliance, whose principle statement includes focusing on individuals and collaboration among individuals, has been criticized for exclusivity. Lou reports on these allegations that "the Agile Manifesto is a white male supremacy initiative," which remind us that collaboration and communication are activities that are much easier to talk about than do.

Debate can be an energizing and positive force in software development and project management. This debate over "agile" versus "rigorous" or "traditional" methodologies is healthy, particularly when approached from a perspective of collegiality and respect. We should value the ideas of those who advocate both ends of this debate as well as those who try to bring a leavening perspective to it. I think this issue has a wealth of lively and thought-provoking articles. In fact, this debate has generated such intense interest that we will continue it next month with another round of enlightening articles. My thanks to all the authors for their great contributions.

next issue

The Great Methodologies Debate: Part 2

Guest Editor: Jim Highsmith

Is the RUP really "rich *and* light"? Can a self-described "spy" in the house of agile turn double agent? And why would one of the agile movement's foremost proponents confess that "agility shows up in the execution — or it doesn't"? In the January 2002 issue, we'll continue our methodologies debate with articles by such luminaries as Ivar Jacobson, Stephen Mellor, and Alistair Cockburn.

Tune in next month for more lively opinions from both sides of the methodological divide.

 ©2001 Cutter Information Corp.

Agile *Can* Scale: Inventing and Reinventing SCRUM in Five Companies

by Jeff Sutherland

In recent months, a wide range of publications — *Software Development*, *IEEE Software*, *Cutter IT Journal*, *Software Testing and Quality Engineering*, and even *The Economist* — have published articles on agile software development methodologies, reflecting a growing interest in these new approaches to software development (Extreme Programming [XP], Crystal Methodologies, SCRUM, Adaptive Software Development, Feature-Driven Development, and Dynamic Systems Development Methodology among them). In addition to these "named" methodologies, scores of organizations have developed their own "lighter" approaches to building software. The formation of the Agile Alliance by a group of expert consultants and authors on development process has fueled increasing interest in ways to deliver quality software in short, fixed delivery schedules, under severe time-to-market pressures [8].

The goal of SCRUM is to deliver as much quality software as possible within a series of short time-boxes called "sprints," which last about a month. SCRUM is characterized by short, intensive, daily meetings of every person on a software delivery team, usually including product marketing, software analysts, designers, and coders, and even deployment and support staff. SCRUM project planning uses lightweight techniques such as Burndown Charts, as opposed to Gantt charts. A Gantt chart is only as good as the assumptions inherent in the critical path represented on the chart. In agile development, the critical path usually changes daily, rendering any given Gantt chart obsolete within 24 hours. The solution is using a technique to calculate the velocity of development. The neural networks in the brains of team members are used on a daily basis to recalculate the critical path. This allows the plan to be recalculated and the velocity of "burndown" of work to be computed. Team efforts to accelerate or decelerate the velocity of burndown allow a team to "fly" the project into a fixed delivery date.

A typical Burndown Chart is illustrated in Figure 1. It consists of the cumulative time it takes to complete outstanding tasks for deliverable software for a SCRUM sprint. Each developer breaks down tasks into small pieces and enters into an automated backlog system two variables on each active task every day. The automated system then estimates daily the remaining work for each task and sums the work remaining for each task to generate the cumulative backlog. Requiring only one minute of each developer's time each day to update two data items for active tasks, the automated system produces the Burndown Chart. It shows how fast the

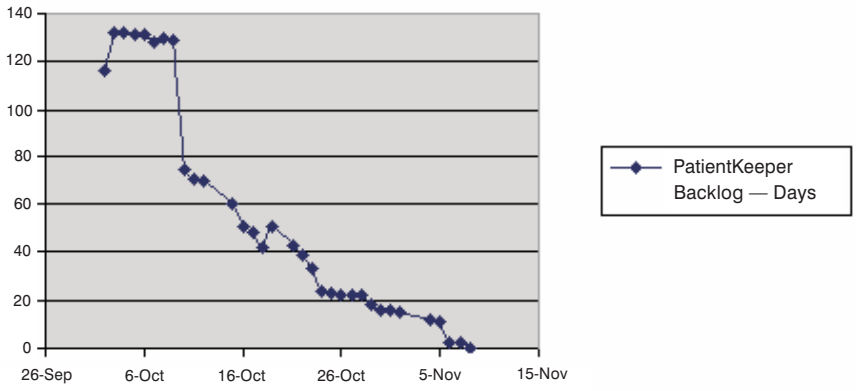

Figure 1 — Burndown chart. (Source: Advanced Development Methodologies)

outstanding backlog is decreasing each day. In the daily SCRUM meetings, the team members determine what actions taken that day will maximize the downward movement of the cumulative backlog. This is equivalent to the team manually recalculating the critical path of the project during a 15-minute daily meeting. Experience has shown that SCRUM project planning will consistently produce a faster path to the end goal than any other form of project planning reported to date, with less administrative overhead than any previously reported approach.

Details of the SCRUM approach have been carefully documented elsewhere. SCRUM is the only agile methodology that has been formalized and published as an organizational pattern for software development [2]. The process assumes that requirements will change during the period between initial specification and delivery of a product. It supports Humphrey's Requirements Uncertainty Principle [9], which states that for a new software system, the requirements will not be completely known until after the users have used it. SCRUM allows for Ziv's Uncertainty Principle in software engineering, which observes that uncertainty is inherent and inevitable in software development processes and products [15]. And it accounts for Wegner's mathematical proof (lemma) that it is not possible to completely specify an interactive system [14]. Most software systems built today are object-oriented implementations, and most of those object-oriented systems depend on environmental inputs to determine process outputs (i.e., they are interactive systems).

Traditional, heavyweight software methodologies assume that requirements can be specified in advance, that they will not change during development, that the users know what they want before they see it, and that software development is a predictable, repeatable process. These assumptions are fundamentally flawed and inconsistent with the mathematical lemmas and principles cited above. As a result, 31% of software projects, usually driven by a variant of the waterfall methodology, are terminated before completion [3].

This article serves as a short retrospective on the origins of SCRUM, its evolution in five companies, and a few key learnings along the way. It will provide a reference point for further investigation and implementation of SCRUM for those interested in using a proven, scalable, lightweight development process that supports the principles of the Agile Alliance as outlined in the "Manifesto for Agile Software Development" (see www.agile alliance.org).

EASEL CORPORATION: THE FIRST SCRUM

SCRUM was started in 1993 for software teams at Easel Corporation, where I was VP of object technology. In the initial SCRUM, we built the first object-oriented design and analysis tool that incorporated round-trip engineering. A second SCRUM implemented the first product to completely automate object-relational mapping in an enterprise development environment. I was assisted by two world-class developers — Jeff McKenna, now an Extreme Programming consultant, and John Scumniotales, now a development leader for object-oriented design tools at Rational Corporation.

In 1995, Easel was acquired by VMARK. SCRUM continued there until I joined Individual in 1996 as VP of engineering to develop Personal Newspage (now www.office.com). I asked Ken Schwaber, CEO of Advanced Development Methodologies, to help me incorporate SCRUM into Individual's development process. In the same year, I took SCRUM to IDX Systems when I assumed the positions of senior VP of engineering and product development and CTO. IDX, one of the largest US healthcare software companies, was the proving ground for multiple team SCRUM implementations. At one point, almost 600 developers were working on dozens of products. In 2000, SCRUM was introduced to PatientKeeper, a mobile/wireless healthcare platform company where I became CTO. So I have experienced SCRUM in five companies that varied widely in size. They were proving grounds for SCRUM in all phases of company growth, from startup, to initial IPO, to mid-sized, and then to a large

 ©2001 Cutter Information Corp.

company delivering enterprise systems to the marketplace.

"All-at-Once" Software Development

There were some key factors that influenced the introduction of SCRUM at Easel Corporation. The book *Wicked Problems, Righteous Solutions* [5] by Peter DeGrace and Leslie Hulet Stahl reviewed the reasons why the waterfall approach to software development does not work for software development today. Requirements are not fully understood before the project begins. The users know what they want only after they see an initial version of the software. Requirements change during the software construction process. And new tools and technologies make implementation strategies unpredictable. DeGrace and Stahl reviewed "All-at-Once" models of software development, which uniquely fit object-oriented implementation of software and help to resolve these challenges.

All-at-Once models of software development assume that the creation of software is done by simultaneously working on requirements, analysis, design, coding, and testing and then delivering the entire system all at once. The simplest All-at-Once model is a single super-programmer creating and delivering an application from beginning to end. All aspects of the development process reside in a single person's head. This is the fastest way to deliver a product that has good internal architectural consistency, and it is the "hacker" model of implementation. The next level of approach to All-at-Once development is handcuffing two programmers together, as in the XP practice of pair programming [1]. Two developers deliver the entire system together. This has been shown to deliver better code (in terms of usability, maintainability, flexibility, and extendibility) faster than work delivered by larger teams. The challenge is to achieve a similar productivity effect in the large with an entire team and then with teams of teams.

Our team-based All-at-Once model was based on both the Japanese approach to new product development, Sashimi, and SCRUM. We were already using production prototyping to build software. It was implemented in slices (Sashimi) where an entire piece of fully integrated functionality worked at the end of an iteration. What intrigued us was Hirotaka Takeuchi and Ikujiro Nonaka's description of the team-building process in setting up and managing a SCRUM [13]. The idea of building a self-empowered team in which everyone had the global view of the product on a daily basis seemed like the right idea. This approach to managing the team, which had been so successful at Honda, Canon, and Fujitsu, resonated with the systems thinking approach being promoted by Peter Senge at MIT [12].

We were also impacted by recent publications in computer science. As I alluded above, Peter Wegner at Brown University demonstrated that it was impossible to fully specify or test an interactive system, which is designed to respond to external inputs (Wegner's Lemma) [14]. Here was mathematical proof that any process that assumed known inputs, as does the waterfall method, was doomed to failure when building an object-oriented system.

Building a self-empowered team in which everyone had the global view of the product on a daily basis seemed like the right idea.

We were prodded into setting up the first SCRUM meeting after reading James Coplien's paper on Borland's development of Quattro Pro for Windows [4]. The Quattro team delivered one million lines of C++ code in 31 months, with a four-person staff growing to eight people later in the project. This was about 1,000 lines of deliverable code per person per week, probably the most productive project ever documented. The team attained this level of productivity by intensive interaction in daily meetings with project management, product management, developers, documenters, and quality assurance staff.

Software Evolution and "Punctuated Equilibrium"

Our daily meetings at Easel were disciplined in the way we that we now understand as the SCRUM pattern [2]. The most interesting

effect of SCRUM on Easel's development environment was an observed "punctuated equilibrium" effect. A fully integrated component design environment leads to rapid evolution of a software system with emergent, adaptive properties, resembling the process of punctuated equilibrium observed in biological species.

It is well understood in biological evolution that change occurs sharply at intervals separated by long periods of apparent stagnation, leading to the concept of punctuated equilibrium [6]. Computer simulations of this phenomenon suggest that periods of equilibrium are actually periods of ongoing genetic change of an organism. The effects of that change are not apparent until several subsystems evolve in parallel to the point where they can work together to produce a dramatic external effect [10]. This punctuated equilibrium effect has been observed by teams working in a component-based environment with adequate business process engineering tools, and the SCRUM development process accentuates the effect.

By having every member of the team see every day what every other team member was doing, we began to see how we could accelerate each other's work. For instance, one developer commented that if he changed a few lines of code, he could eliminate days of work for another developer. This effect was so dramatic that the project accelerated to the point where *it had to be slowed down*. This hyperproductive state was seen in several subsequent SCRUMs, but never one so dramatic as the one at Easel. It was a combination of (1) the skill of the team, (2) the flexibility of a Smalltalk development environment, and (3) the way we approached production prototypes that rapidly evolved into a deliverable product.

The first SCRUM worked from a unique view of a software system. A project domain can be viewed as a set of packages that will form a release. Packages are what the user perceives as pieces of functionality, and they evolve out of work on topic areas (see Figure 2). Topic areas are business object components. Changes are introduced into the system by introducing a unit of work that alters a component. The unit of work in the initial SCRUM was called a Synchstep.

System evolution proceeds in Synchsteps (see Figure 3). After one or more Synchsteps have gone to completion and forced some

By having every member of the team see every day what every other team member was doing, we began to see how we could accelerate each other's work.

refactoring throughout the system, a new package of functionality emerges that is observable to the user. These Synchsteps are similar to genetic mutations. Typically, several interrelated components must mutate in concert to produce a significant new piece of functionality. This new functionality appears as a punctuated equilibrium effect to builders of the system. For a period of time, the system is stable with no new behavior. Then when a certain (somewhat unpredictable) Synchstep completes, the whole system pops up to a new level of functionality, often surprising the development team.

The key to entering a hyperproductive state was not just the SCRUM organizational pattern. We did

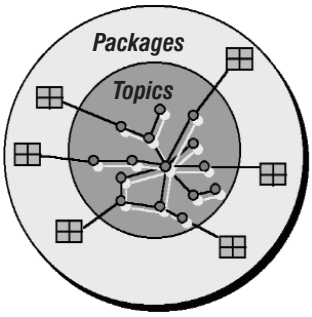

Figure 2 — Initial SCRUM view of a software system.

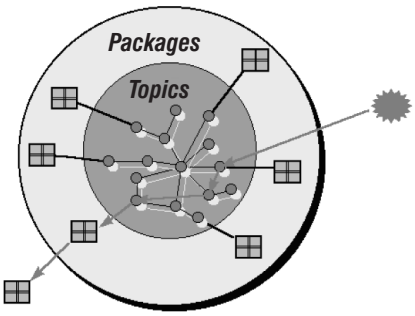

Figure 3 — Firing a Synchstep.

 ©2001 Cutter Information Corp.

constant component testing of topic areas, integration of packages, and refactoring of selected parts of the system. These activities have become key features of XP [7].

Furthermore, in the hyperproductive state, the initial SCRUM entered what professional athletes and martial artists call "the zone." No matter what happened or what problems arose, the response of the team always was far better than the response of any individual. It reminded me of the stories about the Celtics basketball team at their peak, when they could do no wrong. The impact of entering the zone was not just hyperproductivity. People's personal lives were changed. Team members said they would never forget working on such a project, and they would always be looking for another experience like it. It induced open, team-oriented, fun-loving behavior in unexpected persons. Those individuals who could not function well in an open, hyperproductive environment self-selected themselves out of the team by finding other jobs. This actually reinforced positive team behavior similar to biological systems, which select for fitness to the environment, resulting in improved performance of individual organisms.

VMARK: THE FIRST SENIOR MANAGEMENT SCRUM

When Easel Corporation was acquired by VMARK (now Informix), the original SCRUM team continued its work on the same product. The VMARK senior management team was intrigued by SCRUM and asked me to run a weekly senior management team SCRUM to drive all the company's products to the Internet. These meetings started in 1995, and within a few months, the team had caused the introduction of two new Internet products and repositioned current products as Internet applications. Some members of this team left VMARK to become innovators in emerging Internet companies, so SCRUM had an early impact on the Internet.

INDIVIDUAL: THE FIRST INTERNET SCRUM

In the spring of 1996, I returned to Individual, Inc., a company I cofounded as VP of engineering. Much of the SCRUM experience at Individual has been documented by Ken Schwaber [11]. The most impressive thing to me about SCRUM at Individual was not that the team delivered two new Internet products — and multiple releases of one of the products — in a single quarter. It was the fact that SCRUM eliminated about several hours a day of senior management meeting time starting the day the SCRUM began. Because the company had just gone public at the beginning of the Internet explosion, there were multiple competing priorities and constant revision of market strategy. As a result, the development team was constantly changing priorities and unable to deliver product. The management team was meeting daily to determine status of priorities that were viewed differently by every manager. These meetings were eliminated immediately, and the SCRUM served as the focus for decisionmaking.

It was incredibly productive to force all decisions to occur in the daily SCRUM meeting.

It was incredibly productive to force all decisions to occur in the daily SCRUM meeting. If anyone wanted to know the status of specific project deliverables or wanted to influence any priority, he or she could only do it in the SCRUM. I remember the senior VP of marketing sat in on every meeting for a couple of weeks sharing her desperate concern about meeting Internet deliverables and timetables. The effect on the team was not to immediately respond to her despair. Over a period of two weeks, the team self-organized around a plan to meet her priorities with achievable technical delivery dates. When she agreed to the plan, she no longer had to attend any SCRUM meetings. The SCRUM reported status on the Web with green lights, yellow lights, and red lights for all pieces of functionality. In this way, the entire company knew status in real time, all the time.

IDX SYSTEMS: THE FIRST SCRUM IN THE LARGE

During the summer of 1996, IDX Systems hired me to be its senior VP of engineering and product development. I replaced the technical founder of the company, who had led development for almost 30 years. IDX had over 4,000 customers and was one of the largest US healthcare software companies, with hundreds of developers working on dozens of products. Here was an opportunity to extend SCRUM to large-scale development.

The key learning at IDX was that SCRUM scales to any size.

The approach at IDX was to turn the entire development organization into an interlocking set of SCRUMs. Every part of the organization was team based, including the management team, which included two vice presidents, a senior architect, and several directors. Front-line SCRUMs met daily. A SCRUM of SCRUMs, which included the team leaders of each SCRUM in a product line, met weekly. The management SCRUM met monthly.

The key learning at IDX was that SCRUM scales to any size. With dozens of teams in operation, the most difficult problem was ensuring the quality of the SCRUM process in each team, particularly when the entire organization had to learn SCRUM all at once. IDX was large enough to bring in productivity experts to monitor throughput on every project. While most teams were only able to meet the industry average in function points per month delivered, several teams moved into the hyperproductive state, producing deliverable functionality at four to five times the industry average. These teams became shining stars in the organization and examples for the rest of the organization to follow.

PATIENTKEEPER SCRUM: INTEGRATION WITH EXTREME PROGRAMMING

In early 2000, I joined PatientKeeper, Inc. as chief technology officer and began introducing SCRUM into a startup company. I was the 21st employee, and we grew the development team from a dozen people to 45 people in six months. PatientKeeper deploys mobile devices in healthcare institutions to capture and process financial and clinical data. Server technology synchronizes the mobile devices and moves data to and from multiple back-end legacy systems. A robust technical architecture provides enterprise application integration to hospital and clinical systems. Data is forward-deployed from these systems in a PatientKeeper clinical repository. Server technologies migrate changes from our clinical repository to a cache and then to data storage on the mobile device. PatientKeeper proves that SCRUM works equally well across technology implementations.

The key learning at PatientKeeper has involved the introduction of Extreme Programming techniques as a way to implement code delivered by a SCRUM organization. While all teams seem to find it easy to implement a SCRUM organizational process, they do not always find it easy to introduce XP. We have been able to do some team programming and constant testing and refactoring, particularly as we have migrated all development to Java and XML. It has been more difficult to introduce these ideas when developers are working in C and C++. After a year of SCRUM meetings in all areas of development, our processes are maturing enough to capitalize on SCRUM project management techniques, which are now being automated.

CONCLUSIONS

After introducing SCRUM into five different companies of different sizes and with different technologies, I can confidently say that SCRUM works in any environment and can scale into programming in the large. In all cases, it will radically improve communication and delivery of working code. The next challenge for SCRUM, in my view, is to provide a tight integration of the SCRUM organization pattern and XP programming techniques. I believe this integration can generate a hyperproductive

©2001 Cutter Information Corp.

SCRUM on a predictable basis. The first SCRUM did this intuitively before XP was born, and that was its key to extreme performance and a life-changing experience. In addition, the participation of SCRUM leaders in the Agile Alliance [8], a group which has absorbed all leaders of well-known lightweight development processes, will facilitate wider use of SCRUM and its adoption as an enterprise standard development process.

REFERENCES

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

2. Beedle, M., M. Devos, et al. "SCRUM: A Pattern Language for Hyperproductive Software Development." In *Pattern Languages of Program Design 4*, edited by N. Harrison, B. Foote, and H. Rohnert. Addison-Wesley, 1999.

3. Boehm, B. "Project Termination Doesn't Mean Project Failure." *IEEE Computer*, Vol. 33, No. 9 (September 2000), pp. 94-96.

4. Coplien, J.O. "Borland Software Craftsmanship: A New Look at Process, Quality, and Productivity." In *Proceedings of the 5th Annual Borland International Conference*. Borland International, 1994.

5. DeGrace, P., and L.H. Stahl. *Wicked Problems, Righteous Solutions*. Prentice Hall, Yourdon Press, 1990.

6. Dennett, D.C. *Darwin's Dangerous Idea: Evolution and the Meanings of Life*. Simon & Schuster, 1995.

7. Fowler, M. "Is Design Dead?" *Software Development*, Vol. 9, No. 4 (April 2001).

8. Fowler, M., and J. Highsmith. "The Agile Manifesto." *Software Development*, Vol. 9, No. 8 (August 2001), pp. 28-32.

9. Humphrey, W.S. *Introduction to the Personal Software Process*. Addison-Wesley, 1996.

10. Levy, S. *Artificial Life: The Quest for a New Creation*. Pantheon Books, 1992.

11. Schwaber, K., and M. Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 2001.

12. Senge, P. M. *The Fifth Discipline: The Art and Practice of the Learning Organization*. Doubleday/Currency, 1990.

13. Takeuchi, H., and I. Nonaka. "The New New Product Development Game." *Harvard Business Review* (January-February 1986).

14. Wegner, P. "Why Interaction Is More Powerful Than Algorithms." *Communications of the ACM,* Vol. 40, No. 5 (May 1997), pp. 80-91.

15. Ziv, H., and D. Richardson. "The Uncertainty Principle in Software Engineering." In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*. IEEE, 1997.

Jeff Sutherland is CTO of PatientKeeper, where he directs PatientKeeper's (formerly Virtmed's) team of engineers in developing additional services and products. Prior to joining PatientKeeper, Dr. Sutherland served as CTO at IDX, the nation's third-largest hospital information systems company, where he was responsible for setting the architectural direction for products across all business units. While at IDX, Sutherland adapted the SCRUM process to large organizations, and SCRUM has become accepted as an industry standard for rapid application development. He also launched Outreach, the first Web-enabled patient information system, and IDXSite, the first Web-based physician's practice management system.

Dr. Sutherland is also the founder of two companies, Object Databases (now Matisse Software) and Individual, Inc. While at Object Databases, he developed and supported multimedia object database systems. As founder and vice president of engineering at Individual, Inc., he developed the technology for implementing the first automated personal newsletter.

Dr. Sutherland received his masters of science degree in statistics and mathematics from Stanford University and his doctorate in biometrics, medical imaging, and radiation physics from the University of Colorado School of Medicine.

Dr. Sutherland can be reached at PatientKeeper, Inc., 20 Guest Street, Suite 500, Brighton, MA 02135, USA. Tel: +1 617 987 0394; E-mail: jsutherland@patientkeeper.com.

the yin and yang

Agile Versus Traditional: Make Love, Not War!

by Robert L. Glass

Here we go again! With the advent of the "Manifesto for Agile Software Development,"[1] which some think of as a shot across the bow of traditional software engineering, people are choosing up sides and defining lines of fortification. It's the "methodology wars" all over again, except that instead of structured versus data, or object-oriented versus process- or data-oriented, or any of the other tiresome wars that we have fought over software engineering turf over the years, it's now agile versus traditional approaches.

Let me propose a different idea. Instead of fighting over who's right here, why don't we make the assumption that there's right on both sides and try to figure out what that right is? After all, both the traditionalists and the agilists are bright and dedicated people, people who want us to perform software engineering in better ways, resulting in better products. Why do we assume, over and over again, that one camp is right and one is wrong?

Granted, the traditional software engineering folk have grown fat and stodgy. They have institutionalized their various practices, made fun of those who opposed them, and built a massive following of dedicated believers. With all the software engineering textbooks espousing the traditionalist ways of doing business, it is a formidable task to imagine trying to get those traditions to change. The document-driven, waterfall-focused lifecycle; the emphasis on process and doing the right process at the right time; the methodology-driven collection of approaches — all of these are solidly entrenched in the traditional way of doing business.

But the notion of change is supposed to be built into the traditional processes. Even the Software Engineering Institute's Capability Maturity Model (CMM), seen by some as the epitome of traditionalist (perhaps even rigid) software engineering, encompasses in its fifth and highest level of maturity the notion of ongoing process change and improvement. The document-driven approach, then the waterfall lifecycle itself, have lately been rejected and pushed from the traditional mainstream. There is reason to believe, then, that traditional software engineering is open to accepting new ideas and/or rejecting old ones.

The agile newcomers, however, bring their own set of attitudes with them. The recent Agile Manifesto, for example, was constructed by people who referred to themselves as software "anarchists." (The Manifesto is the collected wisdom of those in the agile camp, presented at a meeting in February 2001 as the credo of their movement.) The formerly called "lightweight" processes — "agile" is a so much nicer term, although "lightweight" probably appropriately describes these fairly simple, small-project approaches — have developed a collection of followers who are sometimes zealots in their advocacy. One of the lightweight approaches, Extreme Programming, has become the focus of a series of books published by one of the leading publishing houses. Does a deliberately simple software development approach really need five or six books to define what it is about?!

What I would like to do in this article is propose a kind of truce — in fact, something better than a truce — a way of amalgamating the best of the agile and traditional approaches. I think it is important for both sides to admit, at the outset of what could all too soon become a senseless "war," that the other side really does have some important and worthwhile things to say.

How could we go about amalgamating the best of traditional

[1]See www.agile alliance.org.

 ©2001 Cutter Information Corp.

software engineering and the best of agile software development? I would propose the Agile Manifesto as the best place to start, for a couple of reasons. First of all, it clearly states the important beliefs of the agile camp and thus focuses fairly quickly on areas that may differ from the traditionalist view. Second, it is concise and a nice, clean, place to begin. Traditional software engineering has spread out over so many topics that it would be difficult to single out a collection of places to start describing it.

In the words of the Manifesto, "We are uncovering better ways of developing software by doing it and helping others do it." The Manifesto goes on to list the things its advocates have "come to value." I would like to take each of those valued topics and discuss them from the point view of amalgamating the best of agile and the best of traditional.

AGILE MANIFESTO VALUES

Individuals and Interactions over Processes and Tools

I personally believe that the agile folks are right in this regard. Traditional software engineering has gotten too caught up in its emphasis on process, even institutionalizing the CMM as the best way of building software. We have forgotten the lesson so clearly exhibited on the cover of Barry Boehm's book *Software Engineering Economics* [1], which shows that the quality of the programmers and the team is by far the most influential factor in successful software construction, with processes and tools falling considerably behind. I suspect that the traditionalists won't have too much trouble in accepting this revised emphasis — most practitioners already know that people matter more than process. Even the SEI has devised a People Maturity Model to supplement the CMM, and Watts Humphrey, Mr. Software Process himself, also wrote a couple of (admittedly lesser-known) books on the importance of managing for innovation in software development.

Working Software over Comprehensive Documentation

Once again, I side with the agile folks, although with some caveats. It is important for all of us to remember that the ultimate result of building software is *product*, and that the key component of product is the program/system itself. Documentation matters — we can't do without user manuals and requirements specs. And we desperately need to place more emphasis on maintenance documentation, in order to support the ongoing product evolution that is the dominant task of the software profession. But over the years, the traditionalists made a fetish of documentation. It became the prime goal of the document-driven lifecycle, and the interim documents produced in moving through that lifecycle became — in the minds of some — the primary focus of the software development process. Some academics even got caught up in teaching the document-driven approaches and thus produced a generation of software developers who seemed to believe that good documentation was the prime result of the software development process. No more!

Over the years, the traditionalists made a fetish of documentation.

Customer Collaboration over Contract Negotiation

Here, I think it is important to say that both sides are right. In that best of all possible worlds, contracts would never be needed, and good customer relationships would suffice. If you've ever experienced the result of one of those good customer relationships going south, though, you appreciate how important the contract is. Customer collaboration is a chaotic at best way of resolving disputes over what that collaboration was really supposed to be about. I deeply believe in customer collaboration, and I would agree with the agile folks who would say that, without it, nothing is going to go well. On the other hand, I also deeply believe in contracts, and I would not undertake any significant collaborative effort without one. Beware, of course, of those who make the contract supreme and destroy collaboration in the name of good contracting. A law degree sometimes makes people lose sight of what is important.

Responding to Change over Following a Plan

Once again, both sides are right. For too many years, the traditionalists have put so much emphasis on planning software development activities that they produced plans that were rigid and inflexible. Meanwhile, we clearly learned the lesson — over and over again — that customers and users do not always know what they want at the outset of a software project, and that we must be open to change during project execution. That was a difficult lesson to learn, because at the same time we were learning *that* lesson, we were learning another one — that requirements change was one of the two most common causes of software project failure. (The other was faulty or erroneous estimation.)

Those two lessons, unfortunately, didn't fit well together. It was some time before we realized that change in the software development process was inevitable and that planning must encompass an ability to accommodate that change. In the end, I tend to side more with the traditionalists than the agile folks on this issue. Planning is vital, and change shouldn't be allowed to run roughshod over it. That planning, however, must cover the notion of change.

It is important to point something out here. The agile folks tend to think in terms of small projects. Extreme Programming, for example, is explicitly for "small to medium projects." It is no accident that these were formerly called "lightweight" approaches. A lot of the stuff that the traditionalists believe in results from having worked on and with large projects. In my humble opinion, those traditionalist large-project beliefs represent hard-won knowledge that the agile folks would do well to pay attention to. Even if the projects they are involved with now are fairly small, there is no reason to think that those projects — or they as programmers — will not grow into larger endeavors. At the same time, I would admonish the traditionalists to quit trying to apply large-project processes in the small-project world. They tend to collapse of their own weight.

And one more important aside: small project software is relatively easy to build. It may be fun to define agile approaches to dealing with such projects, and the guidelines that result can indeed be beneficial. However, it is in the large projects of the software world where things all too often tend to go awry. We definitely need the accumulated wisdom derived from large project endeavors, and it would be worse than throwing out the baby with the bath water to concentrate exclusively on agile approaches without considering their applicability in the scaled-up, programming-in-the-large world. In the end, the compromise between agile and traditional approaches may need to be based on the nature of the project being undertaken.

AGILE MANIFESTO PRINCIPLES

The discussion of the Agile Manifesto above focuses on the values of the agile movement. The Manifesto also states some principles, and I think it is important, in seeking an amalgamation of agile and traditional, to consider each of these principles in turn.

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

This principle covers a lot of ground, including such things as the Extreme Programming notion of continuous customer involvement and the notion of frequent and regular product integration.

I personally side more with the traditional folks here. I think customer involvement at its best tends to peak at the beginning and the end of the project, and I think having customers aboard during the design and implementation and early checkout phases is a waste of their time and an unwarranted intrusion into the programmers' time. Regarding continuous delivery, I think that for a project of any size, there is little point in delivering early developmental work (e.g., unit test results) to the customer. The time will come, of course, when it is important to bring evidence to the customer that something valuable is indeed being produced, and the sooner that can be accomplished, the better. The "daily build" espoused by Microsoft and many other micro software firms is probably an excellent way of keeping all the

 ©2001 Cutter Information Corp.

project folk communicating with each other about the evolving product, and it would make an excellent addition to the traditional view of development. So there is goodness on both sides here, and some positive amalgamation could result.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
Well, sorry, agile folks, but you're flying in the face of human nature here! It is simply unnatural to *welcome* change late in the project. Perhaps you need to phrase that another way — "Be prepared to accept change," for example. What you are really saying, I think, is that projects belong to customers, and they should have the ultimate say in what they get. That is fine. Still, there need to be dampers on late-project change. Realistic cost and schedule impacts should be required for all changes during a project, and the later in the project, the more important they are. Project changes can often scuttle even the best of software projects, and that danger is more to be guarded against than the possibility that the customer won't get exactly what he or she wants. (Note this: on most significant projects, there are multiple customers, who may not agree on the need for during-project change. Be very careful that what a customer tells you is really what *all* the customers need.)

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter time scale.
I don't see much of a difference between this principle and the first one. I will only say this here — note the time frame, a couple of weeks or months. Clearly these are small projects we are talking about. Note the comments above about the needs of larger projects and the importance of bearing those needs in mind.

Business people and developers work together daily throughout the project.
This is pretty well covered by the discussion above about customer collaboration. Who can oppose good collaboration? But who can support that collaboration on a daily basis? Once again, this tends to be small-project thinking.

Note also the use of the expression "business people." This tends to imply a particular application domain. Why did the agile folks leave the term "customer" behind here? I am sure they want their ideas to apply beyond business applications.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
A wholehearted "Yes" to the agile folks on this one. Regarding motivated individuals, I can only add "skilled." Skilled and motivated individuals are by far the most important productivity and quality enhancers known to the software field. The differences between individuals have been shown, in many research studies, to be phenomenal. (Good programmers are anywhere from 5 to 30 *times* better than their alternatives.) Given that good people cost at most twice as much as their not-as-good brethren, they not only bring better productivity and quality to the table, but they also represent the biggest bargain in the software business!

It is simply unnatural to *welcome* change late in the project.

Regarding environment and support, Tom DeMarco and Tim Lister pointed out the importance of the programmer's support environment oh-so-many-years-ago in their book *Peopleware* [2], and nothing on that subject has changed since then. A good environment is vital.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
Once again, I think the agile folks got it very right here. It may seem almost archaic to emphasize face-to-face communication in an era where electronic communication is seen as sufficient. However, there is something about the informal instantaneousness of face-to-face that can't be duplicated any other way. If a programmer has to enter an e-mail system, or get up from his or her chair and walk 50 feet, it somehow isn't the same thing at all. When a programmer needs to know

the answer to a question, it is important for him or her to get the correct answer to that question immediately rather than to succumb to the temptation of making an assumption, which, as often as not, will turn out to be (a) wrong, and (b) troublesome.

By the way, there's a corollary to this principle. The development team must sit together, in contiguous office space. Only in that way can this informal, instantaneous communication be achieved. Since the Extreme Programming movement includes this in its own set of principles, it seems odd that the agile folks didn't do the same.

Working software is the primary measure of progress.

Right on, agile folks! To me, this is a reaction against that document-driven lifecycle the traditionalists were so enamored of a dozen or so years ago. Back in those days, progress was measured in documents completed, and there was a document invented for every milestone. Yet there was a serious problem with that way of measuring progress. If a development team wanted to hide its lack of progress from its management, it could simply concentrate on producing the required documents and avoid building code ("working software") altogether. I don't think I ever knew of a development team that made a practice of that, but I'm sure there was never a development team that didn't consider it when the going got tough.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Perhaps this principle is worded too subtly, because its true meaning may not leap out at you at first. This is an argument against our era's greatest plague on the house of software: unrealistic schedule pressure and the resulting death marches so familiar to everyone in our field. It is derived from, for example, the Extreme Programming belief that programmers should work a 40-hour week. I wholeheartedly support the agile movement's emphasis on this particular kind of humanistic sanity!

A great deal of the agile movement is about what I would call "programmer power."

Continuous attention to technical excellence and good design enhances agility.

This is another principle that may be too subtly worded. This principle is primarily about the notion of ongoing refactoring, the idea that software should undergo improvement whenever the need is identified. It is what many of us used to call "maintenance for maintenance's sake," and although it seems superficially something that everyone could buy into, there is some controversy here.

Most software maintenance, of course, is about enhancement and (to a lesser extent) error correction. Both of those activities involve tasks that are approved and supported by management. Maintenance for maintenance's sake, on the other hand, is a programmer-authorized activity, and one that is often done without management's knowledge. Is that a good thing or a bad thing? To my way of thinking (and, I expect, to the way of thinking of anyone who agrees with the previous principle about utilizing motivated and skilled programmers), this is a good thing, and I agree with the agile folks on this issue. But not everyone — especially managers who believe in tight control — will agree.

Note, by the way, that a great deal of the agile movement is about what I would call "programmer power." "Trust motivated and skilled programmers to get the job done." "Work at a sustainable pace." "Allow the programmers to make the needed refactoring decisions." There are a couple more "programmer power" principles still to come. Watch for them.

Simplicity — the art of maximizing the amount of work not done — is essential.

Who couldn't agree with this principle? There are all kinds of famous sayings about simplicity, including my favorite: "The solution to a problem should be as simple as possible, but no simpler." There is a trap hidden in this principle, however. The Extreme Programming folks, for example, believe in minimizing design time and moving quickly on

 ©2001 Cutter Information Corp.

to programming, and the notion of "simple design" includes that idea. That may be all well and good for tiny programs, but it quickly becomes infeasible as problem complexity increases. In my early programming days, I remember my goal was always to start programming as soon as I understood the problem I was to solve. Yet something funny frequently happened along the way. My coding would get slower and slower, until I realized that I really didn't know where I was going. What I had to do at that time was to stop coding, step back from the problem, and figure out how I was going to solve it. It was only later in my career that I realized that when I stepped back, what I was really doing was beginning the activity of design. I suspect that I'm not by any means the only programmer who got trapped in impossible coding activities because the design wasn't nailed down first.

There is something seductive about the principle of simplicity, but I think it leads us down many wrong paths. The first workable design solution is often not the most appropriate one. Research shows us that the best designers use a highly iterative process, reworking the design over and over until they can find a satisfying (or satisficing) solution to the problem at hand. Good design, for software with a significant life span, must include trying to anticipate the enhancements that customers will want in the future. That is not completely possible, of course, but significant simplification of ongoing maintenance can emerge from a more complicated design that facilitates changes needed later.

So — bottom line — I am going to support the traditionalists on this one. Everyone loves simplicity, of course, but beware of that "no simpler" caveat in the quotation above.

The best architectures, requirements, and design emerge from self-governing teams.

The wording of this principle is a bit strange. Recall that the Agile Manifesto was produced by a group of self-proclaimed "anarchists" from a variety of different "lightweight methodology" backgrounds. That kind of group effort sometimes leads to awkward results — as the saying goes, the camel is a racehorse designed by a committee — and this principle feels like a case in point. Upon reading it, you realize this principle is not about architecture, or requirements, or design. It is about an organizational approach, self-governing teams. Now, are such teams a good thing or a bad thing? Experience and folklore tell most of us that the best software is produced by the small, skilled team and that hordes of developers produce pretty messy software products. But does the small, skilled team equate to the self-governing team?

My belief in the previously discussed principle of "programmer power" tempts me to say "yes," and my experience on self-governing teams strengthens that to "Oh, YES." To be honest, though, I know of no research study that either supports or refutes this concept. So I'm going to punt on this one. I'm unwilling to agree with the agile folks that self-governance is best, even though it's tempting to do so. On the other hand, I'm also unwilling to agree with the traditionalists who would probably opt for a strong management, as opposed to self-governing, approach. In the midst of this issue, it's important to remember that for massive system development — the kind where some kind of horde really is necessary — self-governance is nearly impossible, and a strong manager approach is probably required. So, this may well be a case of "it depends."

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Again, this is a principle that most everyone would superficially support. Even the CMM provides for ongoing process change and improvement in its fifth level of maturity. The key here may well be who is responsible for that reflecting and tuning and adjusting. This is another "programmer power" initiative — it's the team that is responsible for its own behavioral change. Is that a good thing? In my experience, the answer is an overwhelming "yes." I feel strongly about that because I have participated on teams where direction for behavioral change came from the outside — upper management or even hired-gun

gurus who inflicted severe harm on the team by demanding nonsensical changes that the team simply did not believe in and would not make.

The focus of methodological investigation should not be on invention followed by advocacy.

KEEPING SCORE

There you have it: a look at the agile programming movement based on a subjective examination of the Agile Manifesto values and principles, looking for amalgamation — ideas presented by the agile folks that could supplement or revise, and benefit, the traditional approaches. It is my opinion that the traditionalists have a lot to learn from the agile folks, and that traditional software engineering can be enriched by paying attention to new ideas springing up from the field. In fact, if you add up my subjective verdicts on when agile is best and when traditional is best, you'd find this:

Agile best: 8

Traditional best: 5

Tie: 3

There is one other point I want to make here. Much of the debate between agile and traditional comes down to "it depends" kinds of issues. I believe that the software engineering field has too long ignored the importance of project differences in determining how a project should be done. Project differences have a number of dimensions:

- **Size.** Small projects need a lot less formality than large ones.
- **Application domain.** The solution approach to a business problem will be very different from a scientific/engineering one.
- **Criticality.** Much more error-removal activity is necessary for critical projects.
- **Innovativeness.** Traditional approaches may have to be thrown out the window when a team is addressing a problem that has never been solved before.

This is my favorite set of project differences, but colleagues have suggested lots of others. A too-tight schedule, for example, calls for different approaches from those used when the schedule is reasonable. So does being unable to hire those motivated, skilled people we'd all like to have.

But the point here is this: the era of "one-size-fits-all" approaches to software development has long since passed. Each of the ideas built into any methodological approach — be it traditional or agile or other — probably has a time when it applies. The focus of methodological investigation should not be on invention followed by advocacy. It should be on determining, for each idea, what its area of best applicability is. Only when we reach that higher-level mindset will we eschew the methodology wars that have plagued our past and threaten to drive agilists and traditionalists apart. It's time for all of us to begin moving toward a "best of both" approach. Make love, not war!

REFERENCES

1. Boehm, Barry. *Software Engineering Economics*. Prentice Hall, 1981.

2. DeMarco, Tom, and Tim Lister. *Peopleware*. Dorset House, 1987.

Robert L. Glass is president of Computing Trends, publishers of The Software Practitioner. *He has been active in the field of computing and software for over 45 years, largely in industry (1954-1982 and 1988-present), but also as an academic (1982-1988). He is the author of over 20 books and 70 papers on computing subjects, editor of Elsevier's* Journal of Systems and Software, *and a columnist for several periodicals, including* Communications of the ACM *(the "Practical Programmer" column) and* IEEE Software *("The Loyal Opposition"). Mr. Glass was for 15 years a lecturer for the ACM and was named a fellow of the ACM in 1998. He received an honorary Ph.D. from Linkoping University in Sweden in 1995. He describes himself by saying, "My head is in the academic area of computing, but my heart is in its practice."*

Mr. Glass can be reached at 1416 Sare Road, Bloomington IN 47401, USA. Tel/Fax: +1 812 337 8047; E-mail: rlglass@acm.org.

 ©2001 Cutter Information Corp.

Business Intelligence Methodologies: Agile with Rigor?

by Larissa T. Moss

Several months ago, I was asked by one of the barely surviving dot-com companies to conduct a technical project management seminar. It took three months to negotiate a contract for a three-day seminar. When I arrived one hour prior to the seminar, the workbooks were not ready but "on their way" from Kinko's. Once the workbooks had arrived 15 minutes prior to class, I noticed that they were collated and bound incorrectly. Rather than causing panic or even mild concern, this situation raised the level of excitement among the staff, who jumped on their skateboards and rolled in to the "rescue." They manually recollated and rebound all 50 workbooks in a record time of 25 minutes. When they were done, there was cheering in the hallways, and "high fives" were exchanged among the "heroes" because they managed to "pull off the impossible" one more time. After all, class got started only 10 minutes late — with a slightly discombobulated instructor.

It was immediately apparent to me that my lesson plan was out the window. My prepared training material on how to plan, estimate, staff, coordinate, and control a project had to be adapted to their environment in a hurry if I was to make any impact on their work habits. Clearly these "kids" on skateboards thrived on chaos, and it would have been rather naïve of me to expect that the discipline (I dared not use the word "rigor") I had to offer would be accepted with open arms. Actually, I must admit, I delighted in their teamwork, in their ability to organize themselves (if you can call it that) and to "pull off" what seemed impossible. They gleamed with pride in their agility as well as their workmanship. However, it was clear to me, and evidently to the dot-com executives who engaged me, that their company could not survive like this.

I share this experience because it demonstrates the potential productivity that could come from the "thrills of chaos" (*agility* to them), if channeled effectively, and the devastating effects of killing enthusiasm and morale that could result from the "dregs of structure" (*rigor* to them). As a matter of fact, I started my class with precisely that exercise: "What are the thrills of chaos? What are the dregs of structure?"

THRILLS OF CHAOS

The first question we explored was the visible thrill and adrenaline rush of a chaotic situation. As with all brainstorming sessions, all contributions were equal and nothing was censored. Our brainstorming session produced a long list of thrills derived from chaos, including:

- It's like a "high"
- Unlimited creativity
- You are free, unrestrained
- Personal freedom
- Spontaneity
- Camaraderie
- Be a hero
- Stand out, be noticed
- Boost to one's self-esteem
- Excitement, fun
- Defiance
- Sense of accomplishment
- You can hide your incompetence

There is an indisputable payback for operating in chaos. Whether the payback is a "boost to one's self-esteem" or to "hide one's incompetence," there are distinct personal benefits. These personal benefits spur people on to accomplish "missions impossible." It would be counterproductive to try and squelch these paybacks. On the contrary, these paybacks must

be considered and carefully architected into any structure or discipline, such as a methodology.

DREGS OF STRUCTURE

The second question we explored was the flip side of "thrills of chaos," namely the fears and anxieties or "dregs" that any kind of structure or rigor would produce. This list spilled well over one flipchart page and included:

- Oppressive
- Big Brother
- Being under somebody's thumb
- Kills creativity
- Boring
- Never works anyway
- Wastes time
- Suffocating
- Military
- "The" establishment
- Being old and stale
- Dictatorship
- Losing one's individualism
- Choking, controlling
- Un-American

Fears and anxieties are even greater motivators to defend the "thrills of chaos" than are personal benefits. It is no wonder that any type of structure or discipline — or let's call it what it is: *rigor* — is rejected when it results in such strong personal feelings as "choking" or working under a "dictatorship."

RIGOR VERSUS AGILITY

Although it appeared that my pupils found bliss in their chaos, I suspected that many had not slept too much during the previous week, as they were finishing up a project. I heard from their manager that they wouldn't get too much sleep in the future either, because portions of their work had to be redone due to errors. I also picked up that some had family problems (yes, they were old enough to have families) because they were never home. It was time to turn the table and ask the other two questions: "What is bad about chaos?" and "What benefits can structure provide?" To my amazement, the resulting lists were almost as long as the first ones.

The answers to the question "What is bad about chaos" included:

- You don't have a life
- Not enough sleep
- It's frustrating to keep redoing work
- Lost concentration
- High error rate
- You run 200 mph but cover less than 100 (miles)
- Management is displeased
- Spouses threaten divorce
- Sets precedents for unreasonable expectations

The answers to the question "What benefits can structure provide" included:

- Less fury; calmness
- Being organized
- Sense of being in control
- Being able to plan activities outside of work
- Time to think
- Be creative
- Balance
- Rest and vacations
- Sense of accomplishment, job well done
- Recognition
- Learning
- Raised self-esteem

It was striking to me to see that some of their responses were the same as to the question about the thrills of chaos, such as "be creative" and "raised self-esteem." They did not reject structure per se, only the *rigor* that accompanied it and only when imposed by other people, such as their management. The stage was set for me to talk about how to plan, estimate, staff, coordinate, and control a project under the umbrella of an agile and adaptive methodology. Of course we didn't use the "M" word; we called it "organized dynamism" instead.

The most valuable lesson learned from my dot-com experience is that agility and rigor not only *can* complement each other — they *should*. As a matter of fact, agility without rigor is chaos. What is the worst thing about chaos? Extremely high risk and high cost. What is the purpose of a methodology? Reduce risk and cost. What projects have the highest potential

© 2001 by Larissa T. Moss, Method Focus, Inc.

for risk? The complex ones, regardless of their size (although larger projects are inherently more complex). One can therefore deduce that the more complex a project, the higher the risk; hence, the more rigor is required to manage the risk.

COMPLEXITY OF BI APPLICATIONS

Business intelligence (BI) initiatives require a complex cross-organizationally integrated (enterprise-wide) decision support environment. BI applications are pieces of that environment. They are developed and managed as an integrated and reconciled set of applications, not as standalone silo systems. Therefore, the cross-organizational interrelationships among BI applications make each BI project complex. Additional factors contributing to the complexity are large scopes, new technology, dirty data, lack of resources, untrained staff, nonparticipation of users, and few, if any, standards or common methods. The fact that most BI applications are affected by several of these factors unequivocally leads to one conclusion: BI applications cannot be developed without some degree of rigor. It would be impossible to remember all the activities and tasks that need to be performed on BI projects without it.

In addition to the above-mentioned factors that contribute to BI project complexity, each BI project is really a set of three simultaneous projects. The three parallel development tracks are:

- Back-end extract-transform-load (ETL)
- Front-end application (data access and analysis, usually online analytical processing [OLAP])
- Metadata repository

Back-End ETL Track

The ETL team is responsible for designing, building, and populating the BI target databases. The high-level activities needed to accomplish these objectives include:

- Understanding the data as well as access requirements
- Investigating the condition of the source data
- Purchasing and installing an ETL tool
- Designing the ETL process
- Designing the staging area
- Scrubbing (cleansing) the data
- Writing the programs or the technical metadata for the ETL tool
- Populating the BI target databases with current data, historical data, and periodic updates

Front-End Application Track

The application team is responsible for providing value-added data delivery mechanisms, which can take the form of reports, canned queries, formatted screens, and/or private data sets for downloading. The application team must also enable easy ad hoc (spontaneous) access to the business data housed in the BI target databases. The high-level activities needed to build the application include:

- Understanding the access requirements
- Understanding the reporting requirements
- Understanding the database designs
- Understanding the data distribution
- Purchasing and installing end-user tools (query tools, OLAP, data mining, etc.)
- Prototyping the application
- Designing and building the application
- Preparing the technical metadata for the end-user tools

Metadata Repository Track

Metadata is ancillary information about business data: for example, the definition of a data element, domain (allowable value) of a data element, business rules about a data element, file/table name where the data element is stored, definition of the file/table content, programs accessing the data element, and so on. Metadata is a deliverable for every BI project. It cannot be rejected as *documentation*, because it must serve the users as a tool for *navigation* through the BI environment. The high-level activities needed to enable this include:

- Understanding what metadata components are required
- Designing the metadata repository (metamodel)

- Purchasing and installing a metadata repository product
- Designing the metadata load process
- Designing the metadata access process
- Writing the programs or the technical metadata for the repository tool
- Populating the metadata repository with business metadata from the CASE tool and other analysis documents and with technical data from the database, programs, the ETL tool, end-user tools, and other technical documents

DEVELOPMENT STEPS

Each of the three tracks is an engineering project in its own right. Like any engineering project, be it building a system or an airplane, the project goes through six engineering stages, either explicitly or implicitly. When these stages are explicitly addressed, they take on the framework of a methodology. The six engineering stages are:

Justification. An assessment is made of a business problem or a business opportunity, which gives rise to the engineering project.

Planning. Strategic and tactical plans are developed, which lay out how the engineering project will be accomplished.

Business analysis. Detailed analysis of the business problem or business opportunity is performed, which provides a solid understanding of the business requirements for a solution.

Design. A product is conceived, which solves the business problem or enables the business opportunity.

Construction. The conceived product is built and is expected to provide a return on the development investment within a predefined time frame.

Deployment. The finished product is implemented (or sold) and its effectiveness is measured, which will determine whether the solution meets, exceeds, or fails the expected return on investment.

To get a better appreciation for the complexity of activities that potentially have to be performed and coordinated among the three development tracks, let us examine the 16 development steps within the six engineering stages [1].

Justification Stage

Step 1: Business Case Assessment

Each BI application should be cost-justified and should clearly define the benefits of either solving a business problem or taking advantage of a business opportunity. Major activities are:

- Determining the business problems
- Assessing the current decision support solutions
- Assessing operational sources and procedures
- Assessing competitors' BI initiatives
- Determining BI project objectives
- Proposing a BI solution
- Performing cost-benefit analysis
- Performing risk assessment

Planning Stage

Step 2: Enterprise Infrastructure

An enterprise infrastructure has two components:

Technical infrastructure components include hardware, software, middleware, database management systems, operating systems, network components, the metadata repository, and applications. The major activities are:

- Assessing the current platform
- Evaluating and selecting new products
- Expanding the current platform (installing products)

Nontechnical infrastructure components include metadata standards, data naming standards, enterprise data architecture (evolving), methodology, guidelines, testing procedures, change control process, issues management procedures, and dispute resolution procedures. The major activities are:

- Assessing the effectiveness of current infrastructure components
- Improving current nontechnical infrastructure components

© 2001 by Larissa T. Moss, Method Focus, Inc.

Step 3: Project Planning

BI projects are extremely dynamic. Therefore project planning must be detailed. Major activities include:

- Determining project delivery requirements
- Determining the condition of source files and databases
- Determining/revising the cost-benefit
- Determining/revising the risk assessment
- Identifying critical success factors
- Creating a project plan
- Preparing the project charter

Business Analysis Stage

Step 4: Project Delivery Requirements

Scoping is one of the most difficult tasks on BI projects. The desire to have everything instantly is difficult to curtail, but keeping the scope small is essential to minimizing risk. Major activities include:

- Determining requirements for the technical infrastructure
- Determining requirements for the nontechnical infrastructure
- Describing reporting requirements
- Defining requirements for source data
- Defining data quality requirements
- Reviewing the project scope
- Reviewing/refining the evolving enterprise data model
- Defining preliminary service level agreements

Step 5: Data Analysis

The biggest challenge to all BI projects is the quality of the source data. Data analysis in the past was confined to one line of business and was never reconciled with other views in the organization. Major activities include:

- Analyzing external data sources
- Analyzing internal data sources
- Creating/refining the logical data model
- Completing analysis of source data quality
- Preparing data cleansing specifications
- Removing/resolving data discrepancies and inconsistencies

Step 6: Application Prototyping

Analysis for the functional deliverable(s), which used to be called system analysis, is best done through prototyping. Major activities include:

- Analyzing access requirements
- Determining the scope of the prototype
- Selecting tools for prototyping
- Designing reports and queries
- Building and demonstrating the prototype

Step 7: Metadata Repository Analysis

Metadata repositories can be purchased or built. In either case, the requirements for what type of metadata to capture and store have to be documented in a metamodel. In addition, the requirements for delivering metadata to the users have to be analyzed. Major activities include:

- Analyzing metadata requirements
- Creating/enhancing the metamodel
- Creating/enhancing meta-metadata (descriptive information about metadata)
- Analyzing integration and interface requirements to other tools
- Analyzing metadata repository access and reporting requirements

Design Stage

Step 8: Metadata Repository Design

If a metadata repository is purchased, it will most likely have to be extended with features that are required by your BI environment. If a metadata repository is being built, the database has to be designed. Major activities include:

- Evaluating metadata repository products and vendors
- Selecting the metadata repository product
- Designing/enhancing the metadata repository database
- Designing/enhancing metadata load programs

- Designing/enhancing metadata delivery mechanisms (reports, online help, etc.)

Step 9: Database Design

One or more databases will be storing the business data in detailed or aggregated form, depending on the reporting requirements of the users. Major activities include:

- Analyzing access requirements
- Determining the level of aggregation (detail versus summary)
- Determining reporting dimensions
- Identifying similar reporting patterns
- Determining an appropriate design schema (relational versus multidimensional)
- Creating the physical data model (logical database design)
- Creating the DDL and DCL (data definition and data control language)
- Building and tuning the databases
- Developing database maintenance procedures

Step 10: ETL Design

ETL processing time frames (batch windows) are typically small. Therefore, designing the ETL process is a challenge for most organizations. Major activities include:

- Analyzing source data files and databases
- Evaluating ETL products and vendors
- Selecting and installing the ETL tool
- Mapping source data elements to target database columns
- Designing the ETL process flow
- Designing three sets of ETL programs: initial load, historical load, and periodic delta loads
- Setting up the ETL staging area

Construction Stage

Step 11: ETL Development

Depending on the data cleansing and data transformation requirements, an ETL tool may or may not be the best solution. In either case, pre-processing the data and writing extensions to the tool capabilities is frequently required. Major activities include:

- Building the ETL process (writing programs or metadata for the ETL tool)
- Testing the ETL process
- Preparing the ETL process for production

Step 12: Application Development

Once the prototyping effort has finalized the functional delivery requirements, true development can begin on either the same user access tool(s) or on different tools. Major activities include:

- Determining the final project delivery requirements
- Designing/refining reports and queries
- Building the application (writing programs or metadata for the user access tools)
- Testing the application
- Preparing the application for production
- Training the users

Step 13: Data Mining

The real payback for BI initiatives comes from the business knowledge hidden in the organization's data, which can only be discovered with data mining tools. Major activities include:

- Stating the business problem
- Collecting the data
- Consolidating and cleansing the data
- Selecting the data by variable type
- Preparing the data
- Constructing the analytical data model
- Running the data mining tool
- Interpreting the data mining results
- Performing validation of data mining results

Step 14: Metadata Repository Development

If the decision is made to build a metadata repository rather than to buy one, the metadata team is usually charged with the development process. Major activities include:

- Building or installing the metadata repository
- Loading the metadata repository

© 2001 by Larissa T. Moss, Method Focus, Inc.

- Building the delivery mechanisms (reports, queries, online help function, etc.)
- Testing the load and delivery programs
- Preparing the metadata repository for production
- Training the users and IT on the metadata repository

Deployment Stage

Step 15: Implementation

Once all components of the BI application are thoroughly tested, the databases and applications are rolled out. Major activities include:

- Planning the implementation rollout
- Establishing the production environment
- Installing BI application components in production
- Installing online help documentation/programs
- Initiating production processing
- Loading the production BI databases
- Preparing for ongoing support

Step 16: Release Evaluation

With a product release concept, it is very important to benefit from "lessons learned" on the previous project. Major activities include:

- Preparing for a post-implementation review (PIR)
- Organizing the PIR session
- Conducting the PIR session
- Following up on the PIR session

Naturally, these activities are not performed in sequence, not even within the parallel tracks. On some BI applications, some of the activities are performed implicitly or may not need to be performed at all. On other BI applications, activities from multiple steps can be rolled into one step and then be performed iteratively over multiple releases. It is up to the project manager to understand the project complexity and the associated risks before choosing the minimum number of steps and the minimum number of activities necessary to control those risks.

BI RELEASE CONCEPT

If project complexity with all its associated risks dictates the amount of rigor needed to control the risks, then it stands to reason that the complexity must be minimized in order to stay agile. Unfortunately, most BI project teams bite off much more than they can chew during the allotted time frame; that is, their scope is much too large for their deadline. These projects turn into disasters because the users demand too much functionality on too much data in too little time — and because IT scrambles to deliver by performing the unavoidable activities implicitly and either dropping or forgetting the remaining necessary tasks. Scoping will be a continuous struggle as long as we are married to the idea that the scope of a project is equal to the scope of the application.

We have been hearing for years that "you cannot build a data warehouse in one big bang." Nobody challenges this maxim anymore, seeing that a data warehouse is a very large piece of the BI environment with multiple applications rolled out over time. But that is not going far enough in the attempt to reduce scope. We should not even build a BI application in one big bang. Following the principles of Extreme Programming, Extreme Project Management, et al., we should also adopt Extreme *Scoping*.

Extreme Scoping

Extreme Scoping means reducing the complexity of each project in order to achieve a balance between agility and rigor, and to deliver *something* to the users in a very short period of time, without chaos or risk — or at least with minimum risk. The *something* being delivered would equate to a fraction of the typically requested application deliverable. Most users balk at this approach, not comprehending or not trusting that this is only the *first release*. They don't realize that BI is a journey, not a destination. They don't understand that more functionality and more data will follow with each subsequent release until the BI application (as originally requested) is fully functioning.

Maybe the best way to illustrate the potential effectiveness of Extreme Scoping is to recall the concepts of prototyping. In prototyping we focus on a small (partial and incomplete) scope that is not too complex, and we deliver a not yet fully functioning application. The same concepts apply to the

release of a BI application. We would start with a very small nucleus of the requested application, which is not too complex to build, and we would deliver some partially functioning piece of the overall BI application. The next release would include another small portion of the overall scope with a little more complexity, and the deliverable would be a more complete — but still not finished — application. This process would be repeated until the application is fully functioning. In other words, it would take several releases (projects) to deliver a fully functioning application, not just one.

Although Extreme Scoping is based on the same concepts as prototyping, it is not equal to prototyping. The difference is that traditional prototyping is pure ad hoc development, whereas Extreme Scoping demands that all necessary activities appropriate to the scope of each release are performed with the rigor of a methodology. But since the scope of each release is no longer the entire application, the complexity of each release is reduced, the number of activities and in many cases the number of steps to be performed are decreased, and the methods for controlling the project can be less rigorous. In other words, the smaller the scope, the higher the agility, with minimized risk.

CONCLUSION

The debate about rigor versus agility should not be about the choice between them, but rather about creating methods and guidelines for merging "just enough" of both. Project managers and projects teams are overwhelmed with the BI demands placed on them, which is why many react by either choking themselves with too much rigor just to stay in control or by thrashing like my dot-com pupils just to stay afloat. There has to be a better way — a creative way, a *fun* way — to work on BI projects, be in control, be productive, and deliver value-added application releases. However, it will require more than just extreme versions of known disciplines. It will require both cultural and infrastructure changes to how companies approach human resource development and how they reward (as opposed to punish) experimentation. It will depend on how much users take ownership of BI, how well standards are adopted, and how readily organizations accept a new release approach to application development. These are all topics that need to be explored in further debates and future articles. The message of this article is that methodologies are just checklists — it will take more than changing the nature of a methodology to change our traditional approach to system development.

REFERENCE

1. Atre, Shaku, and Larissa Terpeluk Moss. *Business Intelligence Roadmap: The Complete Lifecycle*. Addison-Wesley, forthcoming 2002.

Larissa T. Moss is a Senior Consultant with Cutter Consortium's Business Intelligence Practice and a contributor to the Business Intelligence Advisory Service. She is founder and president of Method Focus, Inc. and specializes in improving the quality of business information systems. Prior to founding Method Focus, Inc. in 1991, Ms. Moss was a partner at Tondolea Corporation, a data management consulting firm. She was the primary information resource management (IRM) consultant and relational database designer. In 1989, she coauthored the data-driven system development lifecycle methodology RSDM-2000. Prior to that, she was an assistant vice president at Security Pacific National Bank (now Bank of America), responsible for information engineering (IE) and quality assurance (QA).

Ms. Moss started her IT career in 1980 as a systems analyst and mainframe developer. Her articles on data warehousing, project management, information asset management, and data quality are regularly published in The Navigator, DM Review, Journal of Data Warehousing, *and* Analytic Edge. *She is the coauthor, with Sid Adelman, of* Data Warehouse Project Management, *and the coauthor with Shaku Atre, of* BI Roadmap: The Complete Lifecycle. *Ms. Moss is a frequent speaker at conferences in the US, Europe, and Asia on data warehousing, CRM, and other information asset management topics, such as data quality, data integration, and cross-organizational development.*

Ms. Moss can be reached at Method Focus, Inc., P.O. Box 67, Sierra Madre, CA 91024-0067, USA. Tel: +1 626 355 8167; Fax: +1 626 355 4177; E-mail: lmoss@cutter.com.

© 2001 by Larissa T. Moss, Method Focus, Inc.

Agility with the RUP

by Philippe Kruchten

WHAT IS AGILITY?

What characterizes an *agile* software development process? Is agility about being fast to deliver? "Faster, better, cheaper" is a laudable goal, but faster in itself is not necessarily the right thing to achieve if it is detrimental to quality. Is agility about minimizing the size of the process used to develop software, the size of its description? Not really, or by that measure the null process would be the best.

Agility, for a software development organization, is the ability to adapt and react expeditiously and appropriately to changes in its environment and to demands imposed by this environment.

An *agile process* is one that readily embraces and supports this degree of adaptability. So, it is not simply about the size of the process or the speed of delivery; it is mainly about flexibility. In this article, I will explain how the Rational Unified Process® (RUP®) is a process framework that allows this flexibility [3, 6].

WHAT IS THE RATIONAL UNIFIED PROCESS?

The RUP has several facets:

- It is a software development process.
- It is a catalog of excellent and field-proven software development practices.
- It is developed and commercialized like a software product.
- But it is first and foremost a software process *framework*.

The RUP was intentionally designed with a wide scope of applicability to accommodate variations in:

- Project size
- Application domain (business system, technical system)
- Technology used (language, platforms)
- Business context (inhouse development, product development, independent software vendor [ISV], contractual development)

As a process framework, the RUP provides a systematic way to capture, organize, and deliver software engineering know-how. It relies on a simple and rigorous underlying process model to organize the know-how: it is more than just a collection of texts or books.

But this process framework is not just an empty shell. It comes prepopulated with a large amount of process know-how already captured over the last 15 years by Rational folks (or people working for companies later acquired by Rational Software, or some of our partners, such as IBM, HP, and BEA). The RUP is not a closed, finite process, published once and for all, and it does not answer all the questions or solve all the problems of software development. The RUP framework is an *open* framework, which continues to evolve and which is updated twice a year: its structure is refined, the associated tool support is developed, and its content is expanded.

On one hand, the Rational process development group continues to add to and update the RUP's content as technology evolves and based on feedback from users of its installed base. On the other hand, many partner companies and individuals have adopted the framework to capture and organize their own process know-how, which they use for their own

purpose, or resell, or which becomes integrated with the framework delivered by Rational.

THE RUP FRAMEWORK

The RUP framework is organized around several simple interconnected concepts [4]:

- It defines process *roles*, which capture the competence, skills, and responsibilities that individuals taking part in the development may be called to play.
- It describes the work that each role performs in *activities*, decomposed into *steps*.
- These activities operate on concrete *artifacts*: models, code, documents, and reports.
- There is a wealth of associated *guidance* for the roles, activities, and artifacts, in the form of guidelines, templates, examples, and tool mentors, which explain how to perform a certain activity with a given tool.
- All these process definition elements are organized into *disciplines*.

Some of its detractors call RUP a "heavyweight" process and depict it as a behemoth that forces you to do zillions of useless and unnatural things. We see it more as a rich palette of knowledge from which to choose what you need.

Presently the basic RUP defines 9 disciplines, some 40+ roles, and 100+ artifacts and has more than a thousand guidance pages. And you can find process "add-ins" to add even more functionality and content. Some of its detractors call RUP a "heavyweight" process and depict it as a behemoth that forces you to do zillions of useless and unnatural things. We see it more as a rich palette of knowledge from which to choose what you need.

Let me give an analogy. With a well-chosen vocabulary of, say, 800 words, you can get along in many circumstances in many parts of the world. If you are given a full-blown dictionary (like *Webster's*), first, nobody forces you to use each and every word it contains; second, you can adjust your level of speech to adapt to many other situations beyond the basic ones, and, third, you can gradually enhance your vocabulary. Hopefully, the 800 words are a subset of the dictionary.

ADAPTING THE RUP

The RUP is neither a dogma nor a religion. The RUP does not define a rigid, "one-size-fits-all" recipe for software development. No, you do not need a minimum of 40 people to fulfill the 40 roles, and you do not need to develop over a hundred different artifacts. You could quickly get into trouble if you were to try this. These process elements are defined in the RUP and provided in *electronic form* to provide the flexibility you need to adapt the process you want to use to the demands of your specific development environment.

The RUP embodies many proven software development practices. Six have been made more visible:

- Develop iteratively.
- Model visually.
- Manage requirements.
- Control changes.
- Continuously verify quality.
- Use component-based architectures.

Additionally, the RUP is based on other key principles that are less visible and more easily forgotten. To mention only a few:

- Develop only what is necessary.
- Focus on valuable results, not on how the results are achieved.
- Minimize the production of paperwork.
- Be flexible.
- Learn from your mistakes.
- Revisit your risks regularly.
- Establish objective, measurable criteria for progress.
- Automate what is human intensive, tedious, and error prone.
- Use small, empowered teams.
- Have a plan.

©2001 Rational Software Corporation. All rights reserved.

A key concept in adopting and adapting the RUP is the *development case*. A development case is a tailored project-specific instance of the RUP. This process description defines and identifies, by reference to the RUP framework:

- What will be developed
- Which artifacts are really needed
- Which templates should be used
- Which artifacts already exist
- Which roles will be needed
- Which activities will be performed
- Which guidelines, project standards, and tools will be used

PRODUCING A DEVELOPMENT CASE

The development case is usually produced by:

- Selecting relevant elements in the RUP framework
- Eliminating unnecessary process elements
- Adding any missing company-, domain-, or technology-specific elements
- Tailoring some elements to the precise project context

As the project unfolds, the development case captures some of the lessons learned by evolving itself in parallel with the software product being developed. Often guidelines or checkpoints for reviews are added to avoid repeating mistakes. The project evolves not only the software it produces, but also its own ability to develop that software.

In many cases, the development case is produced by referring to the generic RUP, but you can also create and deploy an instance of the RUP; that is, a configuration of the RUP that is tailored to your own needs. A development case is not necessarily a big document.

If starting from a large process framework is an intimidating task, you might want to use a bottom-up approach and start with a minimal set of process elements — the RUP essentials, which are the elements generally found in 95% of software projects [5]:

- A vision for what you want to achieve
- A plan
- Some objective success criteria, such as a business case
- A design
- A list of risks
- A list of defects or other kinds of change requests, and so on

ITERATIVE DEVELOPMENT

Of the six best practices embraced by the RUP, the one that has the greatest impact on process agility is *iterative development*.

The RUP describes a versatile development lifecycle with four phases (inception, elaboration, construction, and transition), each one with specific objectives. Inside these phases, development proceeds in small iterations: design a little, build a little, and test a little. It is a spiral process as described by Barry Boehm [1].

The project evolves not only the software it produces, but also its own ability to develop that software.

Building the software product incrementally not only allows for early risk mitigation, making tactical changes, and managing scope, but also fine-tuning the process itself: adjusting the development process, guidance, activities, and artifacts as you learn from previous iterations. So an initial development case is likely to evolve during the development cycle. This is done after each iteration by reflecting on what has worked, what has not worked, and how the organization and the process can be improved. Iterative development provides an ideal setting for a software process improvement process. Jim Highsmith describes an iteration as: Speculate, Collaborate, *Learn* [2].

Iterative development requires some careful planning. The development case is only one facet of that planning, which must be done not only for the whole cycle, but also for each iteration. The project plan, revisited at each iteration, must reflect the instantiated process and explicitly define everyone's roles and activities. The

mapping of roles to individuals embodied in the plan means that not everyone would have to know everything about the RUP.

PROCESS ENGINEERING

The RUP framework achieves this process agility, this capacity to adapt the development process itself, through one of its disciplines that covers the *process engineering* aspects. In other words, the RUP framework contains the role, the activities, the artifacts, the guidance, and the tools to evolve the RUP framework, to build a development case, and to evolve it throughout the project, and if you wish, to create your own configuration of the RUP.

The role is the *process engineer*. The *process* itself and the *development case* are the artifacts being produced or modified.

The actual process you will use is subordinated to the needs of the project, not the other way around.

Having an explicitly defined process rather than a vague reference to the literature and a handful of valuable principles is not an attempt to minimize the importance of people. The process does not make people replaceable pawns and does not justify employing less-qualified or less-experienced developers. A defined process also does not, by itself, make the products better in any dimension of quality.

Having a defined process:

- Allows people to communicate better, especially in larger, more distributed organizations, or across multiple organizations.
- Avoids reinventing process on the fly, arguing about the definition of what needs to be done and how, and debating about the criteria used to evaluate the quality of the result.
- Helps achieve greater consistency in the artifacts produced.
- And finally, allows the process to be used as an objective, concrete object of study and reflection in order to take the explicit step of improving it. When things are not working, having a defined process allows objective discussion about the failures and weaknesses of the team, rather than simply finger pointing.

In this process engineering effort, which is easier?

1. Starting from a blank slate, with a few key principles, and then building the process from the bottom up?
2. Or starting from a rich knowledge base and choosing, shrinking, modifying, and evolving existing recipes to fit the problem at hand?

With proper guidance (in the process itself) and some tool support, many organizations have found the second approach more efficient and scalable, as it allows them to benefit from the experience of many others, without excluding the integration of their own.

Agility does not always equate to "small." If your environment imposes a high-ceremony, over-the-fence approach, or requires that your organization be certified CMM Level 3, or demands that you deliver design documentation, agility is being able to rapidly accommodate these constraints, which is hard to do when starting from a very small process base. Waving your arms and saying that these are stupid or unfair constraints may not win you the contract.

PROJECT, PROCESS, ORGANIZATION

The project comes first, with its constraints and its environment driving your choice in the most adequate process configuration. The actual process you will use is subordinated to the needs of the project, not the other way around. Development cases can be reused from one project to another, allowing a faster project start.

In larger organizations that have a method and tools department, an SEPG (software engineering process group), or a QA department, some more ambitious process engineering can take place. The harvesting of process know-how can be done across multiple projects and departments and then be consolidated. To speed up project inception, a

©2001 Rational Software Corporation. All rights reserved.

predefined, partly instantiated development case can be defined, containing company-specific templates and guidelines. This predefined development case is then fully instantiated by each project (or division, or department) to further tailor it to suit their needs.

The proper order is (1) project needs, which drive (2) process definition, which can be (3) supported and amplified by the organization. It is not, as we often see, first an organization-level process pushed down to the project level, which the team is then left to try to make the best of it on a given development cycle, usually only paying lip service to this process that has fallen from the sky.

TOWARD GREATER PROCESS AGILITY

Creating the initial, very first development case is often a hurdle, especially when an organization is small, or newly created, and has not been exposed to much of the RUP philosophy. There are many choices to make, and as the scope of the RUP framework widens, there will be even more. Also, the process description elements in the RUP are not just standalone pieces; they are tightly integrated with many hyperlinks. So eliminating an arbitrary element often looks like pulling one spaghetti noodle out of your plate when you have added too much Emmental cheese: it takes a lot of other noodles with it.

This is where process agility can be further improved by helping the project manager or process engineer to make the right set of choices for the initial development case. There are several ways to attack this issue:

Predefined configurations. Provide predefined instances of the RUP framework, where some of the choices have already been made for a given type of software development environment.

Componentized RUP. Organize the RUP framework to manipulate smaller chunks of process know-how: process components, where the amount of coupling is reduced and well defined.

Tools for RUP configuration. Provide tool support for the construction of a RUP configuration.

Tools for process authoring. Provide tool support for process authoring, to allow the users to expand the RUP framework and create their own process components.

Predefined Configurations

An example of the predefined configuration is a variant *RUP for Small Projects*, where many choices have been made to eliminate elements (artifacts and the corresponding activities and roles) that are not likely to be found in small five-people-for-five-weeks projects, or where artifacts have been merged or reorganized. Note that a RUP for Small Projects is indeed smaller than the generic RUP framework, but it is not an extremely small RUP process reduced to a few high-level recommendations for beginners. This process still aims at providing full guidance on how to achieve quality results, and therefore still contains a lot of guidance. Remember also that software development is not just about programming; there are other important aspects, such as deployment, requirements management, and technical project management. Because of the wide range of factors that affect the project choices, there are only a few such configurations that can be ready-made out of the box.

Componentized RUP

We are gradually reorganizing the RUP framework to fully support the notion of process components, implemented in the form of process plug-ins (see Figure 1). The framework contains a base RUP, or the very elementary process guidance that will figure in most configurations you can imagine: general definitions, concepts, and principles, as well as the definition of key best practices; in particular, the definition of the iterative lifecycle or the use of key technology, such as the UML.

Process components can be added to this base RUP. A process component contains new process definition elements that are added to the base, but it may also redefine and specialize elements already existing in the base. For example, the base may contain a simple project management discipline, and you might replace it with a

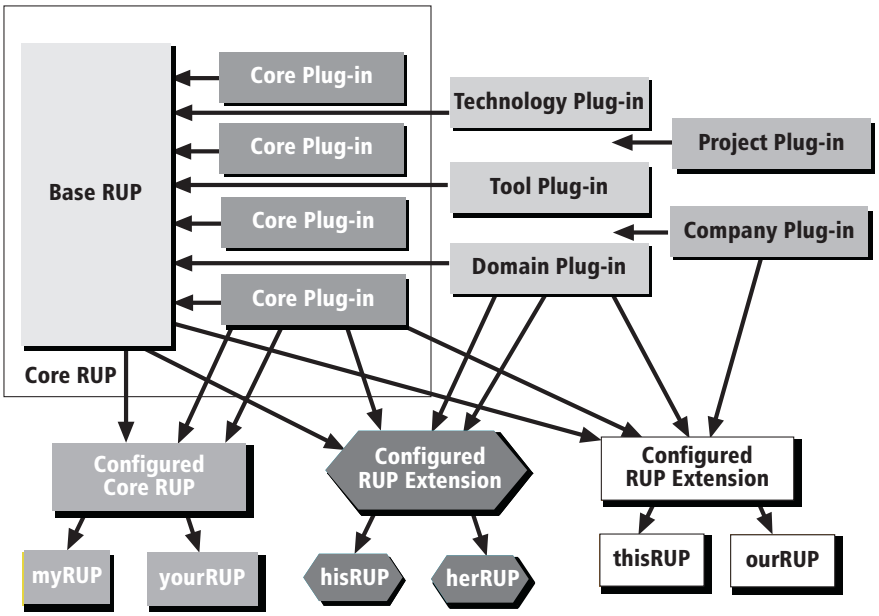

Figure 1 — Componentized RUP: base RUP with process plug-ins.

more sophisticated project management discipline geared toward the management of a large project under government contract. Extensions are done by "dropping" a plug-in into a compatible base.

Beyond plug-ins that provide the core software engineering guidance, very specialized plug-ins bring know-how targeted at a given technology (e.g., J2EE), domain (e.g., real-time embedded systems, data warehousing), or tool (e.g., code generator).

The result is still a fully integrated, hyperlinked process, since each plug-in "knows" how to relate its own elements to the base. This approach is more like adding spaghetti noodles and cheese and ground pepper to a small initial serving.

Tools for RUP Configuration and Process Authoring

Plug-ins are developed by the process engineers using the Rational Process Workbench,® a process modeling tool built on top of Rational Rose,™ using UML to model the process. The practitioner — a project manager, for example — can configure an instance of the RUP using the RUP builder, which allows the selection of appropriate and compatible plug-ins for a base RUP (see Figure 2).

TOWARD A PROCESS MARKETPLACE

The concept of process components and plug-ins opens the possibility of a process marketplace. Not all plug-ins have to be developed and marketed by Rational.

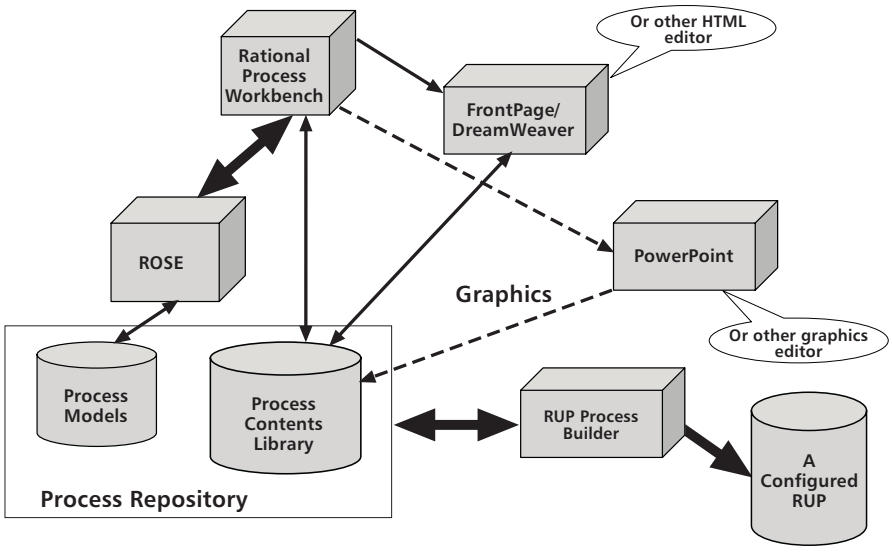

Figure 2 — Tools for RUP configuration and process authoring.

©2001 Rational Software Corporation. All rights reserved.

By making available the very tools with which Rational developed the RUP, and by pushing toward a standardization of the underlying process metamodel [4], Rational is opening up this possibility. The OO technology it relies on is powerful enough to allow a third party to replace parts of the RUP guidance with its own idea, further opening this marketplace.

CONCLUSION

Agility is the ability for an organization to adapt and react expeditiously and appropriately to changes in its environment and to the demands imposed by the environment. An agile process is one that readily embraces and supports this kind of adaptability.

The Rational Unified Process contains the guidance necessary to adapt its framework to the initial project environment and to evolve it as a project unfolds. The RUP framework provides this agility by offering a wealth of process guidance to choose from, which prevents having to reinvent process and accommodates a larger scope of situations. A great part of the RUP's agility is derived from its iterative approach, which provides many feedback loops and opportunities to make ongoing tactical changes to evolve the process.

By introducing the concept of a base RUP that can be augmented and enhanced with process components and plug-ins and making available the tools to develop them, Rational provides greater flexibility in tailoring the RUP and opens a potential process marketplace.

REFERENCES

1. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement." *IEEE Computer*, Vol. 21, No. 5 (May 1988), pp. 61-72.

2. Highsmith, James. *Adaptive Software Development.* Dorset House, 2000.

3. Kruchten, Philippe. *The Rational Unified Process — An Introduction.* 2nd ed. Addison-Wesley, 2000.

4. Object Management Group (OMG). *Software Process Engineering Metamodel (SPEM),* doc ad/01-03-08, 2 April 2001 (http://cgi.omg.org/cgi-bin/doc?ad/01-03-08).

5. Probasco, Leslee. "Ten Essentials of RUP." *The Rational Edge*, December 2000 (http://www.therationaledge.com/content/dec_00/f_rup.html).

6. *Rational Unified Process*, version 2001. Rational Software, 2001.

FURTHER READING

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

Royce, Walker. *Software Project Management — A Unified Framework*. Addison-Wesley, 1999.

Philippe Kruchten is the director of process development at Rational Software Canada. He has led the development of the Rational Unified Process since 1996. Mr. Kruchten has been with Rational Software for 14 years, in various positions and in different countries, mostly as a technical consultant on issues such as software architecture and process, in large projects in the domains of telecommunication, transportation, and defense. Prior to joining Rational, he was at Alcatel for eight years designing telephone switches. He was an assistant professor of computer science at the French Institute of Telecommunication.

Mr. Kruchten is a Rational Fellow, a member of IEEE and ACM, and a professional engineer in British Columbia. He has a degree in mechanical engineering and a Ph.D. in computer science. He is the author of the book The Rational Unified Process — An Introduction *(Addison-Wesley, 2000).*

Mr. Kruchten can be reached at Rational Software Canada, 638-650 West 41st Avenue, Vancouver, BC V5Z 2M9, Canada. Tel: +1 604 261 1653; E-mail: pbk@rational.com.

Extreme Requirements Engineering

by Larry Wagner

Requirements engineering is the most critical activity for a software project to ensure delivery of a product that meets the needs of the customer. Poorly done requirements can lead to myriad problems in the development cycle, such as software that is over budget, delivered late, or of poor quality. Incomplete requirements and changing requirements are often major contributors to these problems [3]. Debates rage on today over agile software development versus rigorous software development. Some would accuse the agile side of a "full speed ahead, no matter the consequences" attitude — expediency at whatever the cost. Others would say of the rigorous side, "You will complete this 700-pound requirements specification, no matter if the client no longer needs the software by the time it is ready" — process at whatever the price.

We'll examine the principles of agility versus the principles of rigor in the area of requirements engineering to see if either side of the debate is right, if there are any areas of agreement between the two sides, and, more importantly, what you can leverage from the controversy.

THE AGILE APPROACH

Let's start with a definition of Extreme Programming (XP). XP is an approach to software development based upon speed, customer satisfaction, simplicity, communication, iteration, and small group dynamics. These certainly sound like fine principles. Who could argue with any of them? XP guru Kent Beck points out that:

> XP makes two sets of promises: First to programmers, XP promises that they will be able to work on things that really matter, every day. They won't have to face scary situations alone. They will be able to do everything in their power to make their system successful. They will make decisions that they can make best, and they won't make decisions they aren't best qualified to make. Second to customers and managers, XP promises that they will get the most possible value out of every programming week. Every few weeks they will be able to see concrete progress on goals they care about. They will be able to change the direction of the project in the middle of development without incurring exorbitant costs. In short, XP promises to reduce project risk, improve responsiveness to business changes, improve productivity throughout the life of a system, and add fun to building software in teams — all at the same time [1].

These are promises anyone would like to achieve.

Now let's examine the practices and rules of XP that relate specifically to requirements engineering. The practices and rules reviewed here come from the work of J. Donovan (Don) Wells [5], Kent Beck [1], and Ron Jeffries [2].

- **User stories.** These are in sentence format written by the user. They are usually about three sentences long and are often written on an index card. The main purpose they serve is to provide enough detail about a feature of the system to enable the developers to estimate how long it will take to implement the story.
- **The customer is always available.** This allows the builder of the system to sit down with the customer to receive a detailed description of the requirement/feature first identified in a user story in order to complete the building/coding task.
- **Code the unit test first.** By creating unit tests for the story first, this forces the developer to understand the

 ©2001 Cutter Information Corp.

requirement before he or she starts coding. The concept is to create one test and then code to pass that test, but no more. Then create a second test and add code to pass this test and the original test, and so on until the story and the tests are done. The requirements are documented by the three-sentence story, the tests, and the code. There is no other specific written requirements document.

- **Never add functionality early.** All functionality is based upon the user story and the unit tests that prove the code works. No extra or anticipated functionality to support the next user story is added. Functionality is added only after the user story has been received and the unit test has been written. Additional functionality comes about through adding the next user story to be implemented in the code.
- **Collective ownership.** All code belongs to all programmers. Everyone on the team knows what user story is being satisfied by this code and how the code works. The code is written by a pair of programmers who can be moved around to help other teams or to get help for this program, including splitting the pair up. This movement helps ensure the whole team understands all the code, the tests, and the requirement being fulfilled.
- **Release planning.** This is a meeting used to plan releases for the project. Both the customer and the development team agree to the specific content for the next iteration as well as a plan for future iterations.

When these practices are taken together, requirements are documented by stories, by tests, by code, and in the collective mind of the team. The requirements are added to and changed until the customer is satisfied with the software. No explicit review of documented requirements is done with the customer or the rest of the team (i.e., peer reviews.) The product is the user's demonstration of the requirements.

A RIGOROUS APPROACH

In the area of rigorous approaches, we will look at the Capability Maturity Model (CMM) [4]. The CMM, like the agile approach, purports to have customer satisfaction as its aim. The CMM helps organizations achieve the goal of customer satisfaction by gaining control of their processes for developing and maintaining software. Its purpose is to guide organizations in the improvement of their software processes.

Let's examine what the CMM has to say about requirements engineering. The purpose of managing requirements is to establish a common understanding between the customer and the developer of the customer's requirements. This is accomplished through the following principles.

The CMM helps organizations achieve the goal of customer satisfaction by gaining control of their processes for developing and maintaining software.

- **Requirements are developed, maintained, documented, and verified according to the project's process.** The project has an established method for dealing with the customer's requirements.
- **Requirements are controlled to establish a baseline.** A requirements baseline represents an agreement at a point in time between the customer and the development team of what features and functions will be in the software.
- **Plans, products, and activities are kept consistent with the requirements.** Once the baseline between the customer and development is achieved, it is used to plan and run the project.
- **These principles are institutionalized in the organization.** This suggests that these principles will endure beyond the current project into the future. They will become the ongoing way of doing business. There are nine attributes of an institutionalized process: defined, documented, planned, supported, trained, practiced,

enforced, measured, and improvable. Institutionalization is established by supporting these principles through the organization's policies, training, leadership involvement, quality reviews, and the provision of the resources needed to sustain the principles.

There isn't a conflict between agility and rigor if the XP practices satisfy the CMM conditions.

When these principles are applied together, typically requirements are documented in a requirements specification. The specification is signed off by the customer and becomes the baseline. The requirements become the input that drives the rest of the project. The requirements are placed under configuration management and can be changed through a change process that ensures proper review, analysis, and integration of any change into the entire project.

DO AGILITY AND RIGOR HAVE ANYTHING IN COMMON?

Reviewing the principles and aims of both agility and rigor, there are certainly some things that are common between the two approaches. First of all, both approaches espouse customer satisfaction as the aim of the method. Second, since the CMM doesn't actually specify how a goal has to be accomplished, it is possible for both to coexist. (The CMM does, however, have a set of typical practices based on observed industry best practices.) So again, there isn't a conflict between the two if the XP practices satisfy the CMM conditions. Reviewing each of the CMM requirements principles against what XP calls for, we find the following:

- *Requirements are developed, maintained, documented, and verified according to the project's process*. XP uses index cards to document user stories. Remember that these are not the complete requirement but are the basis for the developers to meet with the customer to talk about the rest of the story. The rest of the story is documented through the test cases and through the code.

  Conclusion: This technique only begins to meet the "spirit" of the CMM principle. It is difficult to find requirements by reading code. A requirement may be spread out through several different sections of code within the module, or even across many modules. Lack of written, traceable requirements can make it difficult to maintain the software over time as developers and customers come and go and the original requirements have changed.

- *Requirements are baselined*. There is a rigorous concept of a baseline in XP. Each iteration is a baseline with a defined set of content agreed to by the customer and the developers.

  Conclusion: This principle seems to be satisfied.

- *Plans are driven by requirements*. The plan for an iteration is directly driven by the content (user stories) of the iteration. The overall project is driven by the release planning meeting in which the overall release plan is created.

  Conclusion: The plans of an XP project match up to the requirements.

- *These principles are institutionalized in the organization*. The CMM principle of institutionalization does not seem to be specifically addressed by XP. However, there is nothing about XP that is contrary to this principle.

  Conclusion: Some areas of institutionalization are indirectly addressed in XP, such as measurement, definition of the process, and enforcement of the process. Good XP practitioners use a defined and documented method. They are given support in the way of workspace needed to effectively use XP and are provided training. The XP methods are enforced (often with peer pressure) and improved through experience.

 ©2001 Cutter Information Corp.

WHAT ARE THE LIMITS FOR XP REQUIREMENTS?

My experience working with customers leads me to conclude that they are usually not ready to start iterating user stories. Customers typically start at a high-level need and work their way down to the details. The customer has a business goal or a process change that he or she is trying to implement. I think of this as the 50,000-foot view of the project. XP doesn't address the high-level initial requirement. This level of requirement is typically documented in a mission statement or in a business needs statement or in a set of objectives for the project. Next, the customer works his or her way down to the 5,000-foot view. Here the actors or users of the system can be identified, along with their goal for using the system and perhaps an initial list of the user stories. The final 5-foot level is the level XP seems to start with — the user story and the test for the story. If the customer is not ready for this level of detail yet, then an approach to document higher-level requirements needs to be undertaken first.

Some software doesn't seem to lend itself as well to XP requirements strategies as others. I offer an example of one system that would work well and one not so well.

Good Fit for XP

I have been working with a client who is the midst of developing a Web-based human resources self-service application. Employees will be able to go into the system and view, change, and add items such as address, dependents, deductions, and so on. Each of these instances makes a good user story that can be elaborated when it is time to code a particular instance.

XP doesn't address the high-level initial requirement. This level of requirement is typically documented in a mission statement or in a business needs statement or in a set of objectives for the project.

Poor Fit for XP

I have also worked with a client who developed a new service parts order entry system. The user story is that an "external" customer wants to order parts by noon and receive them the next day. The issues here are the simplicity of the story and complexity of the system. The first complexity is that there are at least five separate "internal" customers (order department, material control department, warehouse, shipping department, and accounts receivable). The second complexity is that the inventory allocation system used to determine if the inventory is available to ship has to check other commitments: to build assemblies that can also be purchased, other customer orders promised material, material on hold, material being returned to supplier, material committed to manufacturing, material minimum reserve, and so on. The third complexity is pricing and discounts for which the customer might be eligible based on each part, on the total order, and on the total orders for the year. The last complexity is the warehouse requirement to combine this trip through the warehouse with nine other orders, to sequence the picks properly, to not let the trip become bigger than one employee can handle, to not let the trip last longer than an hour. You get the idea: some systems are too complicated to be developed without thoroughly documented, traceable requirements.

WHAT CAN YOU LEVERAGE FROM THIS?

Think about where you are in your project, and think about what your customer is able to contribute at this point. Use XP techniques where they fit well.

Times that the XP requirements techniques will work for you:

- You're working with an already developed system and extending it.
- You're building a new system, you've gotten to the 5-foot level, and the customer is starting to tell user stories.

In these cases, document the conversation with the customer — on an index card, or in a use case, or some other way.

Times the XP requirements techniques will not work for you:

- The system is ill-defined, and the objectives need to be understood at this point.
- The system is very large.
- The system is very complex, and any "apparently simple" change can cause many other segments of the system to change.
- The system is life-critical.

Even if your project falls into the category where a complete XP approach will not work, apply some of the XP principles to help speed up the requirements process. Use facilitated requirements gathering sessions, which can offer some of the benefits of XP. Facilitated sessions speed through the requirements gathering process and get the project quickly to the design and code phases. A facilitated requirements session is a focused meeting led by a trained IT facilitator, which produces specific project deliverables. There can be three or more types of sessions in which requirements are documented for the 50,000-foot, 5,000-foot, and 5-foot level of detail. At the 50,000-foot level, typical deliverables would be: business needs statements, mission statements, objectives, and/or stakeholder lists. At the 5,000-foot level, typical deliverables would be: lists of user stories or lists of use cases, the primary user stories or use case briefs, and any assumptions or constraints related to requirements. At the 5-foot level, typical deliverables are: test cases, use cases, detailed user stories, and any technical system requirements.

As you consider the agile and rigorous approaches available, you need to consider which options to use for which portions of your project, to speed up the development of the software while providing a system that meets your customer's expectations.

REFERENCES

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

2. Jeffries, Ron. "Welcome to XProgramming.com," 2001 (www.xprogramming.com).

3. Johnson, Jim. "Chaos: The Dollar Drain of IT Project Failures." *Application Development Trends,* Vol. 2, No. 1 (January 1995), pp. 41-47.

4. Paulk, Mark C., Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, 1994.

5. Well, J. Donovan. "Extreme Programming: A Gentle Introduction," 2001 (www.extremeprogramming.org).

Larry Wagner is process improvement director for TeraQuest Metrics, Inc., a company that helps IT organizations improve their performance by focusing on their capability to deliver. Mr. Wagner has 30 years' experience in software development and management in the automotive and retail industries. A long-time process advocate, Mr. Wagner recently has been leading requirements engineering workshops that develop requirements management skills and leading requirements facilitation sessions that quickly and accurately gather requirements for projects. His expertise also includes project management, software quality assurance (SQA), process improvement, training, business planning, business process reengineering, and development on mainframe, mid-range, client-server, and Web architectures.

Mr. Wagner can be reached at 3610 Vineyard Springs Court, Rochester, MI 48306-2253, USA. Tel: +1 248 340 7036; E-mail: larry.wagner@teraquest.com.

 ©2001 Cutter Information Corp.

embrace diversity

Exclusion, Assumptions, and Misinterpretation: Foes of Collaboration

by Lou Russell

SURPRISE AND DISTRESS

It all started when I was reviewing the comments on an e-question posed by Martha Heller, executive Web editor of *CIO* magazine: "Would agile software development work in your shop?" As someone who is very interested in the practices and, more importantly, the philosophies of the agile approaches, I wanted to see what the consensus was. I personally have witnessed some surprisingly emotional responses from CIOs about the agile approaches: it would never work, it was just RAD, prototyping was a bad idea, pair programming was too expensive, and so on. I expected to see more of that. To my surprise, I read the following:

> **Boys Club?**
> After reviewing the manifesto site of the Agile Alliance, I have to ask: where are the women? There are no women in the group, no women identified on the site. It appears that an idea that blossomed from an all-guys ski weekend is getting national attention.
>
> As a woman who owns a consulting company dealing with all of the identified corporate issues that stall out projects, I decided in 1999 to make a difference — and I set about creating a practical development model to tie business, design, and technical implementation together, dealing with all of the same identified points as the alliance has mentioned. Ironically, I even called it the "Agile Development Process." When I tried to sell the idea to publishers or video production studios, no one was interested. I received lots of comments like, "Well, this is good, but you really can't change the industry's ways through suggesting better methods."
>
> It's more discouraging than ever to see the attention a group of men can get when they simply identify a problem publicly, but don't provide a well-defined solution. I truly believe that if we had more women involved in technology, we'd have a much more practical and economic focus to our projects, but it continues to look like we still have miles to go... I hope there are enough of us who hang in there to implement good process ideas in the future.

This comment was the only comment on the list without a name and is still only one of two written by women (I wrote the other after reading this). I know quite a few of the men involved in the Agile Manifesto, and a few are friends. I know it was never their intention to be exclusionary, and I wrote as much.

Certainly, the rules are not always fair, and men and women often face different challenges and receive different levels of support. This reality, of course, would be more obvious to the objects of the prejudice and so may not be as obvious to men. I can honestly say, though, that as a woman in IT for over 25 years, I cannot name one time when I was passed over or excluded because I am a woman. There probably were times, but I was too pig-headed or naïve to notice.

I, too, own an IT consulting company and have for over 15 years. I believe that I have grown my company, or chosen not to grow it at times, based on certain beliefs, preferences, values, and behavior that are important to me. It could be these are similar to those of other women. In any case, the question of whether I have been blocked because I am a woman is muddied by the fact that I have sometimes chosen to be blocked because I am Lou.

Thinking all these thoughts, I tucked the experience away in the back of my mind, only to have it reappear more personally days later. I was invited to give a

The woman explained to me that the Agile Manifesto was written by an exclusive group of white men who had excluded both women and non-Americans.

keynote speech to a group of project management professionals in Newport Beach, California, and I talked to them about treating project management improvement systemically, rather than with quick fixes. In talking about systemic intervention, I briefly mentioned the Agile Manifesto as a collection of new thoughts concerning development approaches and suggested that people check it out.

Later, a woman approached me in the hall very calmly and asked if she could talk to me. She explained to me that the Agile Manifesto was written by an exclusive group of white men who had excluded both women and non-Americans.[1] I suggested that was not the intent and that the Agile Manifesto was published to generate dialog, not force a new religion. She said that the fact that these men excluded everyone else might not have been conscious, but that was even worse — they didn't even know how prejudiced they really are. Although we didn't share the same view, I acknowledged hers and expressed the hope that the dialog could continue about these powerful new thoughts, regardless of the source. She claimed that no woman or non-American would have anything to do with the agile movement because of the nature of its creation. She was visibly annoyed that I had "defected" to the white male side.

Stunned, I wrote up the conversation and sent it to my friend Jim Highsmith, thinking he might want a heads-up that this was brewing. He was aware of the "religious" debates going on between traditional and agile methodologies, but the sexism/nationalism angle was new to him. Thus, this article was born.

TWO PERSPECTIVES

In any battle there are two sides, with the one we do not embrace unclear to us by choice. I have a quote hanging above my desk that reminds me, "Everyone's behavior is sane from their perspective." This is often very difficult for me to remember. It is much easier to interpret someone's behavior through the filters of my mind, much easier for me to jump to conclusions about what someone meant to say, what they *should* have done, what *could* have happened. These are behaviors that are a huge part of my dark side, and I work daily to contain them, often without success. I suspect I am not the only person with this problem!

But there *is* truth in other perspectives if we take the time to fully listen and understand, and I know there is truth in the points the two women made. As I look at the Web site where the Agile Manifesto resides, the picture of the meeting used as a background gives me a strong sense of a "men's club" — a place I cannot go, although I know rationally that was not the intent. I can't put my finger on it, but there is a sense of exclusivity.

WOMEN IN TECHNOLOGY

There are still men who believe that women will always be less technically talented. Society may not allow them to *say* it anymore, but the choices they make confirm the belief. Most, I am sure, are not conscious of this belief. Some of the articles written by IT professionals, including some in the Cutter journals, sound pompous, judgmental, and egotistical, not just toward women, but toward anyone who does not share their particular view. Dialog is replaced with a good, tough competition, with someone's winning requiring that others lose.

I think this need to win by defeating others is more prevalent with men (I am biased here) and, furthermore, very prevalent in technology fields where there are more men than women. I certainly know women who have learned to "act male" to succeed, and they exhibit this behavior as strongly as anyone. I believe this competitive nature prohibits us from meeting the needs of our business customers with technology solutions. I believe many women have the skills (missing in

[1]Note: This occurred before 9/11.

 ©2001 Cutter Information Corp.

IT organizations) that are required to deal with these very issues. However, the status quo prevents them from using these critical collaborative skills.

It's a problem — fewer and fewer women are entering technical fields. More and more women are leaving the technical fields. As I left a software development conference in San Francisco with one of our consultants, also a woman, we were asked if we were flight attendants. Women are still few and far between in the IT world. If you are a man, this is a problem for you, too, as your wife and daughters do not have the same options for success that you have.

Politically, we know that it is improper to talk of men being something and women not being something or vice versa. Our cleaned-up language requires that we speak as if everyone is the same or that the differences do not run along race, sex, or nationality lines. But maybe, just maybe, there are inherent strengths that each sex brings to the technical community. Maybe it's time to strengthen the hands of the strong, rather than try to get everyone to look like clones.

A young female system architect spoke to me after another software development conference last week. She was frustrated by the lack of planning, collaboration, and team communication in her work group. She loved technical work, but she also loved working with others in the process. She wanted her team to build a plan collaboratively, to talk to each other when there were challenges, to work and be rewarded as a team. Unfortunately, she was in a work culture with men who preferred to work as individuals, to the extent of overt competition. It was a cowboy world. She was trying to figure out whether it was possible to make it work, or whether she should just go find another job. Her big fear was that a technology job that valued her technical and team talents equally just might not exist.

I suspect that this woman was hired for her team and collaboration competencies. I suspect that the managers at that company saw the cost of cowboys operating unfettered and hoped that through her skills and some corporate osmosis, they would all be cured. It wasn't happening, and it never does. This type of cultural change requires a systemic, open, out-in-front approach. The managers were part of a culture that constrained them from being open and out in front.

BACK TO THE SCENE OF THE CRIME

To my mind, the most ironic thing about the view that the Agile Manifesto is a white male supremacy initiative is that the agile movement is helping emphasize people and collaboration issues, whereas traditional approaches have focused on process and tools. The manifesto, in fact, creates an infrastructure of beliefs encouraging collaboration between customer and developer. The traditional methods seemed to contain an underlying belief that customers were not to be trusted and had to be corralled to prevent them from changing the requirements and scope constantly. The Agile Manifesto may contain the very principles many women have always wished were part of their work. To write it off because men have defined it is, in my opinion, shortsighted. To leverage it, challenge it, and redefine it so women are more accepted in the IT world seems a better strategy.

As I left a software development conference in San Francisco with one of our consultants, also a woman, we were asked if we were flight attendants.

MENDING FENCES

I appreciate the women I spoke about at the start of this article for their bravery in stating their beliefs. Having said that, however, I also believe that if we, as learning consultant Sue Miller Hurst once said, "think someone is a jerk, [we] look for ways their jerkness shows up." If we as people go forward looking for people to mistreat us, we will see it. If we look for ways we are oppressed, we only see the ways we are oppressed. We attract what we fear. When we trust, we earn trust. When we judge and distrust, the behavior reinforces itself rapidly.

PROGRAMMING WITH A PURPOSE

A recent university panel discussed the issue of women, mathematics, and computer science. Here are some of the opinions:

Paul De Palma, Gonzaga University (depalma@cps.gonzaga.edu)

Women are drawn to mathematics, according to the research. They receive nearly half of the undergraduate degrees in the field. For women to be engaged in technology fields requires emphasis on logic and problem solving while deemphasizing bits, bytes, and programming languages independent of a business context.

Peter Henderson, Butler University (phenders@butler.edu)

Ten years of student and alumni surveys at SUNY Stony Brook show that a mathematically oriented first course for computer science majors, Foundations of Computer Science, had women performing better than men, devoting more time, and feeling technology was a positive experience.

Kim Potter Kihlstrom, Westmont College (kimkihls@westmont.edu)

When male students were asked how they became interested in computer science, they replied, "Through playing computer games." Women were more likely to respond, "Because I like math."

It appears the women have a need to know the "Why" behind an activity like programming. Where the program may be a sufficient end unto itself for a man, the program without purpose is not sufficient for many women. Women appear to be drawn to the logic, the strategy, the problem solving, and the puzzle. Combining a woman like this with a man who enjoys translating the solution into code produces a powerful force. One plus one can equal three.

The cycle of respect reinforces itself, but one quick judgment creates an equally powerful cycle of distrust. Hate creates hate, and the only cure is tolerance. We all make poor choices and mistakes. None of us is perfect, and none of us is as good as all of us. For this reason, I avoid with great energy the belief that I am being passed over in the technical world because I am a woman. If I trust that I am included, I am.

So what can we do? We can all stop being silent about times when we feel we have been unfairly excluded or judged. But I also believe we must be more conscious of our approach to providing feedback, making it respectful. Simply exchanging words like "they" and "you" for "I" can create the beginnings of dialog rather than the beginnings of war. In a business culture already dysfunctionally competitive, language is a critical consideration. Consider:

> "They have excluded all women. They think they know everything."
>
> versus
>
> "I feel excluded because I am a woman. I believe I have some ideas to contribute that have not been heard."

We can also apologize when we have hurt someone else, whether consciously or unconsciously. We can ask him or her to help us rebuild mutual trust. When we get angry, we can pause for a second and review the facts of the incident versus our interpretation of the facts. Is it possible we have misinterpreted (yes!)? We can agree that we are all here to take care of each other, not knock each other down. We can agree that we can only survive together, not apart. I write this list for myself as much as for you.

In other articles in this journal, you will read about strong positions, many of them different than your own. You may get angry. Try to postpone your anger, and look for nuggets of truth in everything you read. Allow difference to grow your own thought. Look for points of agreement rather than "jerkness."

In terms of the agile approaches themselves, open your mind. Collaboration, so completely missing from IT work, holds the key to inclusion and business success. Only through collaboration — men and women, technologists and customers, older and younger — can value be delivered to a business and quality of work life delivered to the team. The agile approaches are rooted in the concept of collaboration. Share your thoughts, begin the discussion. Do not allow yourself to be ignored. Honor others' views, but make sure your views are part of the debate.

YOUR DAUGHTERS NEED YOUR HELP

Last Friday night, I received one of the Woman of the Year awards, a

 ©2001 Cutter Information Corp.

huge compliment from the Indiana organization Women and Hi Tech (www.womenandhitech.com). The overriding purpose of this highly productive group of about 200 women is to help other women, young and old, thrive in technology fields. My own three daughters were sitting at the banquet, surrounded by brilliant, caring, busy technical women. I asked my daughters about the evening, and that didn't strike them as unusual. They expect to be heard, expect to be allowed to thrive in whatever field they are best at. I do not want them to go forward thinking that they will be blocked. I want them to go forward excelling at everything they do partly because they fully expect to. I want them to anticipate strength and success. And I want them to treat other human beings like the unique, multidimensional, flawed people we all are, without judgment and misinterpretation, looking for the good and deemphasizing, when possible, what is bad.

As I write this, it is 9/13, just two days after the horrendous tragedy of 9/11. It strikes me how important it is for all of us, regardless of sex, race, or nationality (to say nothing of methodology preference!) to set aside the petty focus on differences that keep us from connecting. We must all work hard to reach out to others, to respect their opinions and skills, even when they are different from our own. We must invest time in all relationships around us, as we have learned how truly fragile they all are. We must be the glue to heal our world. IT has an amazing amount of power to do just that, a power as yet untapped.

SOURCES

Camp, T. "The Incredible Shrinking Pipeline." *Communications of the ACM*, Vol. 40, No. 10 (October 1997), pp. 103-110 (also available at www.mines.edu/fs_home/tcamp/cacm/paper.html).

De Palma, P. "Why Women Avoid Computer Science." *Communications of the ACM*, Vol. 44, No. 6 (June 2001), pp. 27-29.

MIT Department of Electrical Engineering & Computer Science. "Women Undergraduate Enrollment in Electrical Engineering and Computer Science at MIT" (www.swiss.ai.mit.edu/~hal/women-enrollment-comm/final-report.html), 1995.

WICS: An organization for Women in Computer Science at Stanford University (www-cs-students.stanford.edu/~womencs/).

WICSE: An organization for Women in Computer Science and Electrical Engineering at the University of California in Berkeley (www-inst.eecs.berkeley.edu/~wicse/).

Lou Russell founded Russell Martin & Associates in 1987 as a consulting and training company. Ms. Russell's Fortune *500 clients rely on her knowledge of information services infrastructures to help manage major efforts in retention, customer service, project management, systems development, and retooling. Her career has included positions at Ameritech and McDonnell Douglas in training and in course development and delivery. Ms. Russell is the author of* The Accelerated Learning Guidebook: Making the Instructional Process Fast, Flexible, and Fun*; her second book,* Project Management for Course Developers*, will be published soon. Ms. Russell has written articles for* Cutter IT Journal*,* Training*,* Computerworld*, Auerbach journals, and others, and currently serves as president of the Society of Information Management (SIM) Indianapolis chapter. Ms. Russell is a frequent speaker at national industry events.*

Ms. Russell graduated from the computer science program at Purdue University and received a master's of instructional technology in education from Indiana University.

Ms. Russell can be reached at Russell Martin & Associates, 6326 Rucker Road, Suite E, Indianapolis, IN 46220, USA. Tel: +1 317 475 9311; Fax: +1 317 475 0028; E-mail: lou@russellmartin.com.

Cutter IT Journal

CUTTER IT JOURNAL

Topic Index

December 2001	The Great Methodologies Debate: Part I
November 2001	BI and CRM: Critical Success Factors for Achieving Customer Intimacy
October 2001	The Future of SPI
September 2001	Testing E-Business Applications
August 2001	Enterprise Application Integration
July 2001	Web Engineering: An Adult's Guide to Developing Internet-Based Applications
June 2001	The War for IT Talent
May 2001	Implementing an E-Business Strategy
April 2001	Multicultural and International Project Management
March 2001	Developing Wireless Distributed Applications
February 2001	Security
January 2001	Reorganizing IT for E-Business
December 2000	Intellectual Property
November 2000	Light Methodologies

Upcoming

Issue Themes

The Great Methodologies Debate

Legacy Architecture Migration

The Technology Myth in Knowledge Management and Business Intelligence

Risk Management

Web Services

Security

Design for Globalization

Open Source

Testing

XP and Culture Change in an Organization

Mobile Wireless

Preventing IT Burnout

B2B Collaboration

Events

Extreme Programming with Kent Beck
28 April 2002, 9:00-4:00
University Park Hotel@MIT
Cambridge, MA 02139, USA
Early bird special: register now at
www.cutter.com/workshops/extreme.html

Summit 2002
"Business Technology in Uncertain Times"
29 April-1 May 2002
University Park Hotel@MIT
Cambridge, MA 02139, USA
www.cutter.com/summit/

http://www.cutter.com/ or +1 800 964 5118