CUTTER IT JOURNAL

The Journal of Information Technology Management

CUTTER CONSORTIUM
ACCESS TO THE EXPERTS

January 2002 Vol. 15, No. 1

The Great Methodologies Debate: Part 2

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

opinion

Opening Statement

by Jim Highsmith

Who wants to fly a wet beer mat? What is the opposite of a Bengal tiger? For answers to these and other intriguing questions, read on!

As I've read through the articles for the two methodology debate issues of *Cutter IT Journal*, I've wondered at times about the "debate." It seems that no one wants to be thought of as rigid, stiff, or inflexible, so the agile bandwagon has become very popular. Several of the articles in this issue — one by Ivar Jacobson, a principal author of the Rational Unified Process, and those by Alistair Cockburn and Steve Mellor, both authors of the Agile Manifesto — seemed to be more focused on explaining away the differences than explaining them. I don't mean to imply that this convergence is a bad thing, just an interesting one.

However, don't be lulled into the conclusion that the debate has abated. By converging on some issues, others are highlighted. If there were no differences, there would be no debate, and the response to these two issues of the *Journal* has shown that there are certainly debatable issues in this agile versus rigorous methodology arena.

I think there are three categories of issues related to the agile versus rigorous (or whatever label one chooses) methodologies — a "chaordic" perspective, "collaborative" values and principles, and "barely sufficient methodology." A chaordic perspective arises from the recognition and acceptance of increasing levels of unpredictability in our turbulent economic world and that traditional plan-then-execute methods based on stability and predictability will be unsuccessful in this environment. This perspective on turbulent change has a profound impact on how agilists approach project management. Two concrete ramifications of trying to manage in an unpredictable environment are that (1) while goals are achievable, projects themselves are often unpredictable, and (2) the foundation of many process-driven approaches, the goal of repeatable processes, will be unattainable. Jeff Sutherland directly addressed this "predictability" issue in last month's issue.

By project unpredictability, I mean that the traditional measure of project success — conforming to scope, schedule, and cost plans — is an unrealistic goal. In highly volatile environments, the best we can hope for is to focus on a single characteristic — scope, schedule, or cost (usually schedule) — and expect significant variations in the others. High-level CMM organizations tout their ability to achieve their planned goals most of the time (99.5% was reported in one article on a new Level 5 organization [1]). Agile developers would counter that this level of predictability is a sham — either the result of overpadding the plans or eliminating any risky (turbulent) projects from the portfolio. To agile developers, responding to change is more important than conforming to a plan. In a moderate- to high-change environment, responding to those changes and achieving 99.5% conformance to plan are incompatible goals. This "perspective" difference doesn't converge, nor should we want it to. The issues surrounding highly changing environments and how to respond to them will remain fruitful ones for debate.

Second, as Alistair Cockburn points out in his article, sometimes the differences are in emphasis. Agile developers, as described in the Agile Manifesto, have a laser focus on people issues, hence the second characteristic of agile development: collaborative values and principles. Cockburn's article contains words like "team," "community," "reflecting," and "amicability." If you were to ask

©2002 Cutter Information Corp.

him about tools, he would probably say, “Oh yes, you need good, ‘light’ tools also.” Cockburn believes strongly in automated testing tools, for example. However, his emphasis is probably 80% collaboration and 20% process and tools.

On the flip side, consider the emphasis areas in Ivar Jacobson’s article. In the first sentence of the second section, he states that software development is a highly creative undertaking and then recommends enhancing the creative environment by applying tools, processes, and automated knowledge bases to the 80% of the work that he labels as repetitive. Jacobson’s solution involves working on the non-creative piece in order to free up time for the more creative work. Cockburn emphasizes how to improve the creative piece. There is some combination of both, a convergence of ideas, that most organizations will employ. However, the tough question is how to balance, because the two areas are not independent. Too heavy an emphasis on tools and process — even if the 80% “repetitive” work is the target — can adversely impact the “collaborative” environment and thus creativity. Too light an emphasis on tools and process, and the team will spend all its time fighting fires and have little time to be creative.

The concept of a barely sufficient methodology attempts to answer the question of how much “structure” is enough. It derives from the agile chaordic perspective, one that suggests creativity and innovation occur in a slightly messy environment, not a mostly structured one. It comes from the complex adaptive systems concept of balancing “at the edge of chaos.” A barely sufficient methodology eliminates non-value-adding activities and minimizes documentation (and cost), but most importantly, the “barely sufficient” emphasis supports the concept of collaboration (and the idea that too much structure adversely impacts collaboration) and is driven by a chaordic perspective.

So as you read these articles and try to reach a convergence of ideas that might work for your organization, keep these three viewpoints in mind: a chaordic perspective on the turbulence and change that project teams face, the extent to which a collaborative set of values and principles resonates with your organization’s culture, and whether or not a barely sufficient methodology would be “enough” for your company.

“There is no opposite to agile software development,” writes Cockburn in our first article. “The word ‘agile’ depicts where people choose to focus their attention. Alternatives to agile software development arise as soon as they focus their attention elsewhere.” He argues that there is no opposite to agile, that non-agile is a nonthing. In so doing, he suggests altering the debate from the correctness or incorrectness of approaches to software

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.*

©2002 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.

development to a debate about where we place our emphasis. Do we emphasize predictability or maneuverability? Do we emphasize community or predictability? By altering the debate from either/or to where to place our emphasis — which may change from project to project based on the problem domain — Cockburn helps frame the debate in collegial, respectful ways. He then characterizes the four critical factors in "would-be-agile" development as (1) short iterative development subprojects, (2) frequent reflection and feedback on practices, (3) working together in close proximity, and (4) attending to the amicability and morale of the community.

In "The Bogus War," Steve Mellor contends that the war between agile and UML processes "is caused, or at least made to appear more serious than it is, as a result of mistranslation of a word. The mistranslated word is 'model.'" Mellor's article therefore falls into the category of the debate surrounding "barely sufficient methodology." He goes on to define three categories of "model" — sketch, blueprint, and executable — using the example of "models" in airplane development. A sketch might be the "back of the napkin" variety; a blueprint is more detailed, but still at odds with the finished system; and an executable is when the model and code are equivalent "concrete" rather than "abstract" forms. Mellor concludes that few would argue over the need to sketch out ideas, or over the benefits of executable models, so the prime area for debate is over blueprints. He goes on to discuss the progress in executable models, such that debate over blueprints may soon be a non-issue.

Ivar Jacobson, vice president of Rational Software and a primary author of the Rational Unified Process (RUP), recommends combining "rich" and "light" elements to form an agile process. He argues that "light" and "agile" are not equivalent and that rich tools and process help yield flexible, creative, and rapid development. The term "barely sufficient methodology" should actually contain a brief qualifier; it should be "a barely sufficient methodology for the job at hand." Alistair Cockburn's Crystal Methods — Clear, Orange, and so on — are examples of barely sufficient methodologies for different problem domains. Orange (for a 40-person team) is "heavier" than Clear (for a 6-person team). Cockburn begins with minimal process and tools and scales up, while Jacobson begins with an extensive knowledge base — the RUP — and customizes down. There are certainly advantages and disadvantages to each approach.

Jacobson writes that while creativity drives software development, 80% of the work remains repetitive, routine work that is amenable to automation through standard processes and automated tools. Furthermore, he views the RUP as an extensive knowledge base upon which teams can draw as needed. By systematizing and automating the 80% of the work that is routine, Jacobson contends that the remaining 20% that is creative will then be even more effective.

Brian Henderson-Sellers outlines an approach, the OPEN process, that he believes can accommodate both agile and rigorous components under one framework. He agrees with the camp that says one size doesn't fit all, but he cautions that the opposite approach of custom building processes for each project is too expensive. The OPEN Process Framework utilizes a metamodel, a repository of processes, and usage guidelines as components in constructing a process to fit a particular situation.

Finally, Matt Simons addresses one of the key issues in the debate surrounding agile development — project size. While a significant amount of the press surrounding agile approaches has focused on small, colocated teams, Simons reports on one of a growing number of successful agile projects that are in the 50- to 100-person category. The project, done at consulting firm ThoughtWorks,

©2002 Cutter Information Corp.

involved 75-80 people over an 18-month period and delivered approximately 750,000 lines of Java. Simons discusses how ThoughtWorks used the baseline practices of XP and then added additional collaboration practices to handle the larger team size.

Just for fun, I wondered how this project stacked up against industry averages. Since I didn't have access to Cutter's metrics guru at the time I was writing this, I pulled out my copy of *Software Assessments, Benchmarks, and Best Practices* from Capers Jones (Addison-Wesley, 2000) just to get a "ballpark" feel for the project. (Note: There are many unknowns and assumptions in this quick assessment. For example, I just assumed that 75 people worked full time on the ThoughtWorks project for the entire 18 months — probably an erroneous assumption.) The table below shows the statistics for a 10,000 function point (FP) outsourced software project.

While the ThoughtWorks project was staffed with more people, it was completed in nine fewer months than the industry best in class, while matching best-of-class numbers in total effort and productivity. Not bad!

In the second issue on the "Great Methodology Debate," you'll find another wealth of lively, thought-provoking, and entertaining articles. My thanks to all the authors for their wonderful contributions.

REFERENCE

1. Ferrarra, Linda, and Cathy Timko. "The Telcordia Technologies Road to Quality." *Cutter IT Journal*, Vol. 13, No. 2 (February 2000), pp. 28-34.

ThoughtWorks Project Versus Industry Norms

Metric	Industry Average	Best in Class	ThoughtWorks Project
Staff size	57	48	75
Schedule	36 months	27 months	18 months
Productivity in FP/staff month	4.82	7.62	7.5
Total effort in staff months	2,075	1,312	1,350

next issue

Is Risk Management Going the Way of Disco?

Guest Editor: Bob Charette

Risk management is a white-hot topic. It is hard to pick up a magazine or newspaper that doesn't have a story with the word "risk" spread from title to last paragraph. But as risk management becomes more popular, will it go the way of previous good ideas — from fashion, to fad, and then trash heap? Do the debates over the definition of risk signal risk management's good health or impending demise? Tune in next month as we debate whether risk management is doomed or here to stay.

what's your focus?

Agile Software Development Joins the "Would-Be" Crowd

by Alistair Cockburn

What is the opposite of a Bengal tiger — a Siberian tiger, an elephant, a gnat, or a puddle of water? None of them, of course. The phrase non-"Bengal tiger" is uninformative, just as is non-elephant, non-gnat, or non-water. Being a non-something doesn't depict anything.

Similarly, there is no opposite to agile software development. The word "agile" shows where people choose to focus their attention. Alternatives to agile software development arise as soon as they focus their attention elsewhere. They might focus instead on rigorous, predictable, repeatable, defect-free, traceable, or even fun software development. The important thing is to identify the focus of their attention, not to create a negation of a label.

That's still not right. There is really only *would-be*-agile development. Or, for that matter, *would-be*-rigorous, *would-be*-predictable, *would-be*-repeatable, *would-be*-traceable, or *would-be*-fun development.

Would-be indicates that the people involved in the project aspire to a certain focus. Before the project they can say that they intend to work from that focus, but it is only after the project that they can say that their way of working actually *was* agile, repeatable, predictable, defect-free, or fun.

Let's take a fresh look at would-be-agile and would-be-(something other than agile) software development.

WOULD-BE-AGILE SOFTWARE DEVELOPMENT

"Agile" refers to maneuverability, the ability to respond to changes in the environment. Would-be-agile software development means that the team decides to focus on being able to incorporate ongoing requirements changes without great trauma.

Core Elements of Would-Be-Agile

The standard mechanism for reducing the trauma of ongoing requirements changes is to break the project into subprojects, each ending with delivery of running, useful sections of the system. This early and regular delivery provides the team with feedback about both the development process and the system being developed.

Early and regular delivery helps build safety into the project. By delivering several useful releases within a short time span, the sponsors, customers, and developers gain confidence in their way of working. Pausing and reflecting after each subproject, they have opportunities to fix mistakes in their way of working and try out new ideas for becoming more effective.

Early and regular delivery also allows the people to get feedback about the system in operation and, particularly, which of their initial thoughts were mistaken. The deliveries also provide points at which they can uncover and react to new requirements, whether those come from users' reactions or from changes in the business environment.

A common recommendation among agile methodologies, therefore, is to run small (3- to 12-week), time-boxed subprojects and to reprioritize the requirements after each time-box. This gives the projects half of the agility they need.

The other half of their agility comes from replacing some of their written documents with enhanced informal communication among team members, shifting the group's organizational memory from external to tacit knowledge.

External knowledge is the kind we can look at in paper or online documents. Examples are the project plan, the requirements document, interface definition documents, design documents, meeting minutes, design review results, test plans, test scripts, defect reports, and many others. Tacit knowledge is the kind people retain in their persons. It includes not only what each person knows about the project plan, requirements, design, and so on, but also who they know to talk to, when various people are available, the social and technical conventions in place.

All project teams rely on tacit knowledge, usually to a far greater extent than they suspect. Agile methodologies, though, place a deliberate reliance on tacit knowledge. Through the use of informal communication channels, such as colocated teams, pair programming, or daily stand-up meetings, they demand that the people on the team keep each other abreast of ongoing changes to the plan, the requirements, the design, the code.

When it works, it makes agile teams far more maneuverable. A short discussion at the stand-up meeting can suffice to indicate a change in corporate policy or system goal. A discussion between a few developers informs them of changed requirements or a change in the design. Not having to update the external knowledge base means they can make many changes in a short time period.

When it works.

To make it work, the team members build not only on sitting close together, but also on maintaining open communication among themselves. The tacit knowledge base doesn't track very well if people are being secretive with each other.

In other words, group amicability, morale, and community become first-order concerns for the group. To the extent these are in place, the informal communications channels may suffice. To the extent they break down, the communications suffer, and so does the team's maneuverability.

This, then, is the biggest difference between agile and other sorts of development processes. Agile processes put the topics of people, community, amicability, and morale front and center. Standard process descriptions don't have a place for discussing these topics.

It is this line of thinking that brought developer Andrea Branca to write, "Other processes may *look* agile, but they won't *feel* agile."

Variations in Agility

As we have seen, four characteristics of agile processes are:

- Using short subprojects
- Reflecting on their practices
- Working in close location
- Attending to community

With these four characteristics, we can look at ways to vary, strengthen, or weaken the agility

All project teams rely on tacit knowledge, usually to a far greater extent than they suspect.

of the team, depending on where the organization chooses to focus its attention. (Yes, Virginia, the organization might actually choose *not* to focus its attention exclusively on the agility theme.)

Short Subprojects

Ward Cunningham tells of using ultra-short subproject cycles while writing code for Wall Street companies. Given the fickleness of the stock market, there was no telling on one Monday what the critical business need might be the next Monday. Therefore, the team ran each project to be just one week long!

On Monday, team members would meet with their sponsors to select the top issues for the week. The programmers bid what they could accomplish in the week (often not a complete system, of course, but enough to get some initiative to a functioning stopping place). At the end of the week, they completed their initiative, delivered it, and paused. The following Monday, they would find out whether the market still craved that initiative, as they repeated the exercise. If the pressing demand was for some other piece of software, they worked on that instead. Since a piece of software might not get touched again after Friday, it was critical that each subproject be sized to complete and reach a stable, final state on Friday.

That was an extreme situation, of course. Still, the longest I have ever seen subprojects work well is four months. It seems that if they are longer than that, the people aren't able to focus on their work properly. Even in four months, a lot can change about the business. Longer subproject periods make the group less agile.

Setting up and tearing down subprojects is not free. There is planning and setup work to be done at the beginning, and integration, test, and delivery (and possibly training) to be done at the end. Therefore, choose a duration of time that balances the need to respond quickly with the overhead cost of setting up subprojects.

SCRUM recommends one-month subproject lengths. Extreme Programming (XP) recommends three weeks. Crystal Orange used three months. You can see that the exact duration varies by circumstance and preference, with 16 weeks as the outermost boundary.

Reflecting on Practices

The Crystal methodology family is the most adamant about having the members of the team get together and discuss what they are doing well and what they should change. Jim Highsmith (*Adaptive Software Development*) recommends a *product* feedback meeting, so the product requirements can be refined on an ongoing basis. Both of these meetings happen at the end of every subproject. Although reflection workshops were not cast into the 12 key XP practices, in point of practice most XP groups do insist on holding such workshops at the end of each iteration.

The longest I have ever seen subprojects work well is four months.

SCRUM and XP call for short, daily stand-up meetings, in which fast-breaking planning, requirements, design, and also team communication issues can be brought to light. (They are held standing up in order to keep them short!)

Such reflection is core to having the group's process track its needs. To vary, weaken, or strengthen this characteristic, the team can make the reflection workshops longer or shorter, more or less frequent.

An eight-person German project team using Crystal Clear with three-month subprojects chose to hold reflection workshops after each quarter's delivery. To have extended peace and quiet for their workshop, members of the team went to a village well outside of town for two days. They spent the first day team building and reflecting on the previous quarter's work, and the second day jointly creating a plan for the next quarter's work. During the second year of the project, they shortened the workshop to a single day.

In contrast, on my first Crystal Orange project, which also used three-month deliveries but had 45 people (24 programmers), we spent one to two hours in the cafeteria after each delivery. (This is perhaps a commentary both on German workers versus American workers and the insurance industry versus the retail industry.) We only gathered a subset of the team — just the analysts, say, or just the team leads and selected programmers. However, we did it twice per subproject, so that we could change our process in the middle of a subproject.

XP iterations are so short (three weeks), that once after each iteration for an hour may be sufficient time for reflection. A dot-com company using Crystal Orange Web calls for delivery every second Thursday and holds a one-hour reflection workshop with all 50 employees the (Friday) morning after.

Working in Close Location

Crystal Clear and XP simply require that people work in the same or adjacent rooms. This obviously only works for teams up to 8 or (with a stretch) 14 people. None of the other agile methodologies is so adamant on this point.

All agile methodologies are, however, sensitive to colocation, since they rely so heavily on rich, fast communication channels. Using those communications channels allows them to reduce the external documents they have to construct and maintain.

To vary the agility, vary the quality of those communication channels and vary the balance between external and tacit knowledge. It is not the case that to be agile, the team should produce *no* paperwork (external knowledge). External knowledge storage has various advantages, the least being any contractual obligation for the team. Information on whiteboards and in documents has a sort of "stickiness" to it that allows people to refer back to what was decided.

The would-be-agile team, therefore, needs to adjust the amount of knowledge that should reside in conversation and personal memory versus sticky or archivable form. It is extremely rare that the answer to the communication question might be "all oral." In the other direction, due to the very real requirement for tacit knowledge and the overall weakness of our current forms of documentation, I have never yet seen a situation where the answer might be "all external." This means that the organization and the team must decide on how much of each is appropriate. Usually, a little external knowledge goes a long way, and so less is needed than is usually believed.

For communication channels, some people are experimenting with having team members sit in different areas of the same building, connected with Webcams and chat software over a high-speed intranet. While this is obviously not quite as rich as sitting in the same room, they claim fair success with it [1]. Others have experimented with pair programming over high-speed Internet lines, using NetMeeting and a telephone between them. They report that while it is not as good as being in the same room, it is much better than programming alone.

You can see that running a project across countries and time zones creates a real obstacle for the would-be-agile team. Fiddling with the issue of spontaneous, rich communication is a key part of deciding how agile to be.

Running a project across countries and time zones creates a real obstacle for the would-be-agile team.

Attending to Community

The daily stand-up meetings and reflection workshops give people a chance to voice their concerns or suggestions regarding the group's amicability. I recall visiting one XP project on a Tuesday morning and listening while a programmer described a concern and a wish. He said that when he came in on Monday, he found the code vastly changed from Friday. Could people please let others know if they planned to work on the weekend, so everyone would be alert to changes on Monday and would know whom to talk to? This is the sort of attention to community that slips through the cracks in standard process descriptions.

One South African executive likes to take new teams off-site for a two-day team-building course. He's not sure to what extent it really *builds teams*, but it obviously helps. At the very least, he says, the people get to meet each other socially and come to understand that the company considers team quality an issue important enough to spend money on.

Of course, the best team building comes from succeeding, which gets us back to the theme of short subprojects. Along the way, though, the members of the team have to find ways to express their fears and wishes for the community.

Varying this characteristic of the agile process is done by the project sponsor, project manager, or someone on the project with a dominant personality, who can either initiate or kill such discussions. The amount of community present, or even the amount of attention paid to community, is so dependent on the personalities of the people involved that it often is more accidental than orchestrated.

WOULD-BE-(SOMETHING OTHER THAN AGILE) DEVELOPMENT

Not all project teams aspire to being agile. If they do not, they must announce where they are actually focusing their attention. They might strive to achieve predictable, repeatable, cost-optimized, rigorous, traceable, defect-free, fun, or laid-back development. Let's look at some of these in turn.

Predictable Development

In would-be-predictable development, the team focuses on hitting a cost, time, or defect window. An example would be a fixed-time, fixed-cost project. People bidding on such projects may abandon the advantages of the agile mechanisms in exchange for predictability.

Unfortunately, the world is not kind with respect to predictability these days. Unless the organization has already done the same kind of project with the same people in the same technology, the team will be hit by unpleasant surprises somewhere along the way.

Therefore, there is an advantage to mixing some would-be-agile in with would-be-predictable development. That way, when the inevitable surprises do hit, a team that has been attending to short subprojects, reflection, communication, and community can recover more quickly and with less cost. That is a contribution of an agile process to the other would-be processes.

Repeatable Development

Would-be-repeatable development is usually done with the intent of improving predictability, and usually with an added intent of shrinking the time, cost, and/or defect window.

Unfortunately, the world is not kind with respect to predictability these days.

Having the goal of improving predictability presupposes two things: first, that the project assignments, technology, and project teams are similar enough across projects for measurements to transfer, and second, that the metrics are measuring the correct things on the project.

To most project teams engaged in a series of projects where the aim is repeatability, it does not matter if either of those presuppositions is not met. Their asserted goal is to achieve repeatability, and so they must evolve metrics that will transfer across people, technology, and assignments.

Personally, I don't think there are many project series that fit the presuppositions, and I don't think the metrics are yet measuring the correct things. (For example, I have not yet seen such projects measure and report communication and amicability within the team.) One very senior developer hypothesized for me that where repeatability is being achieved, it is being achieved by making the process so slow and heavy that the time spent inventing and revising ideas is negligible in the cost of the entire project.

Far from being a glib attack on would-be-repeatable projects, this idea may be correct: to generate repeatability within a project series, create an overall process large and costly enough to dwarf the inevitable mistakes the developers will make. Then, the risk is greatest on the first project and diminishes after that, as long as the projects *and staff* are similar enough.

Would-be-agile development can still inform would-be-repeatable development. Rarely is any project as similar to a previous project as a subproject is to a previous subproject on the same overall project. The assignment is very much the same, the technology is the same, the people are the same. Repeatability improves over the subprojects, which help for the next overall project. The ideas of agile development also predict new project metrics to be tried: communication distance, frequency, and richness, as well as measures of community and amicability.

Cost-Optimized Development

Would-be-cost-optimized development calls for strategies almost opposite to would-be-agile development. Correctly executed would-be-agile processes cost more than correctly executed would-be-cost-optimized processes. That is, agile teams expend more *work-days* in fewer *elapsed days,* while cost-optimizing teams expend fewer *work-days* in more *elapsed days.*

The cost-optimizing project coordinator arranges for workers to show up only when their particular skills are needed, and to leave again as soon as their work is done. To the extent that the project coordinator can avoid being hit by unpleasant surprises during the project, this careful scheduling allows him or her to optimize salary costs. Since some number of mistakes will

©2002 Cutter Information Corp.

certainly be made, the project coordinator may arrange for people to show up perhaps two or three times, to reduce the defects to acceptable amounts.

You may recognize in this description several other fields besides software development where this strategy is used. I immediately think of building houses and publishing books. In both cases, there is similarity across projects and great incentive for minimizing costs. In house construction, the goal is to get things right on the first pass; publishing houses use a fixed two- or three-pass schedule.

You should see that this form of serialized development will be lower in cost than having the people around for longer periods of time, starting work early, and doing rework as they learn their assumptions were incorrect, as is done in would-be-agile development. A bit of further thinking shows that the serialized development also takes *longer* than the concurrent development, since no team is allowed to start work until the team feeding it is done. In fact, correctly executed serialized development takes the longest time of any correctly executed strategy, at the least cost.

The reason that would-be-cost-optimized projects so rarely succeed in their goal is that surprises pop up at all stages in software development. This invalidates the presupposition for cost-optimizing (recall: "to the extent that the project coordinator can avoid being hit by unpleasant surprises...") and is the very reason people are responding so favorably to would-be-agile development projects.

There probably are circumstances in which surprises can be minimized and plans made to optimize costs. They are likely to be the same circumstances that permit predictability and repeatability.

Rigorous Development

Would-be-rigorous development is a false lead. Few people try to be rigorous for the sake of being rigorous. Rigor is hard. Usually, a group works at being rigorous because they think it will help in some other way, such as predictability, repeatability, defect reduction, or fun. (XP done properly is rigorous. In an odd twist, people practicing XP tell me that the extra rigor makes programming both more defect-free *and* more fun!)

On occasion, a manager or sponsor will advocate more rigor in development as a catch-all prescription meaning to "get better" at something. Therefore, you should probably choose to be would-be-something-else, not merely would-be-rigorous.

Laid-Back or Fun Development

A university organization can't afford to pay its programmers much. To attract and keep anyone of caliber, it has to offer other attractions. The group I am thinking of allows its programmers to work at any time of day or night that suits them, lets them choose their own projects and

The reason that would-be-cost-optimized projects so rarely succeed in their goal is that surprises pop up at all stages in software development.

technologies within a very liberal set of guidelines, and generally works to keep them from feeling pressured. This is would-be-laid-back development at work.

Other groups feel that the only way they can keep their developers motivated and producing excellent software is to make the development environment fun. Programmers in these organizations, who are likely to be working on aggressively state-of-the-art projects, may tease and compete with each other. Rewards include playing competitive computer games like *Doom*, even within standard working hours.

I think it should be obvious that would-be-laid-back and would-be-fun processes are fully compatible with agile development. The latter are even likely to raise morale and communication, although the intense competition in some places may affect amicability to some extent.

SUMMARY

The values and strategies of the "Manifesto for Agile Software Development" are only *supposed* to confer agility on the team. Agility shows up in the execution — or it doesn't. The same is true of the values and strategies of the other sorts of processes. Conversations about these different approaches would be a lot less heated if everyone was clear on the fact that they are *would-be* processes: would-be-agile, would-be-predictable, would-be-fun, and so on.

Calling them *would-be* this or that makes it clear to all where the project sponsors are focusing their attention — responding to late-breaking surprises, hitting a cost window, retaining staff, or whatever they may choose — and thereby diffuses much of the hype currently surrounding would-be-agile development. It allows us to talk about why the sponsors aim for that quality, what strategies might work, what might get in the way, and how to blend the priorities.

Would-be-agile development centers around handling late-breaking surprises. That leads to the strategy of building the project from subprojects (incremental development), enriching informal communications between people, and emphasizing the tacit rather than the external knowledge base.

Would-be-agile strategies can be blended into would-be-predictable development to reduce the cost of handling the surprises that inevitably crop up. They are antithetical to would-be-cost-optimized development, since they make the opposite tradeoffs between elapsed project time and the work-hours used. Would-be-agile strategies should work well with would-be-fun strategies.

I hope that by considering all of these approaches as *wanting* to produce an effect rather than *promising* to produce an effect, we can make progress in evolving and blending strategies.

I hope that by considering all of these approaches as *wanting* to produce an effect rather than *promising* to produce an effect, we can make progress in evolving and blending strategies.

REFERENCE

1. Herring, C., and M. Rees. "Internet-Based Collaborative Software Development Using Microsoft Tools." In *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2001)*. International Institute of Informatics and Systemics, 2001 (http://erwin.dstc.edu.au/Herring/SoftwareEngineering0verInternet-SCI2001.pdf).

Alistair Cockburn is a Senior Consultant with Cutter Consortium's Agile Project Management Practice and a contributor to the Agile Project Management Advisory Service. He is Consulting Fellow at Humans and Technology, where he helps clients succeed with object-oriented projects, including corporate strategy, project setup, staff mentoring, process development, technical design, and design quality. He is the originator of the Crystal family of methodologies, and he has over 20 years' experience leading projects for companies in the US and around the world. He is the author of Surviving Object-Oriented Projects *and* Agile Software Development *and has written papers and articles for a variety of trade journals.*

Mr. Cockburn has developed courses in project survival, methodology development, requirements writing, OO design, and project management strategies. He has been invited to teach and talk at the Software Technology Conference, Object World, Software Development, *and* OOPSLA. *He has taught OO concepts and design to hundreds of executives, managers, and programmers worldwide.*

Mr. Cockburn can be reached at Humans and Technology, 7691 Dell Road, Salt Lake City, UT 84121. Tel: +1 801 947 9275; Fax: +1 775 416 6457; E-mail: acockburn@cutter.com.

©2002 Cutter Information Corp.

The Bogus War

by Stephen J. Mellor

Wars have started as a consequence of misunderstandings or mistranslations during diplomatic negotiations between parties. The use of a word may appear quite innocent to one party, but when translated into the vocabulary of the other, the word may have connotations quite at odds with what was intended. Today, we are told, there is a “war” between the proponents of UML and those of the several agile processes. I contend that this war is caused, or at least made to appear more serious than it is, as a result of the mistranslation of a word.

The mistranslated word is “model.”

A model abstracts away certain information the more clearly to see the issue at hand. For example, we may model an airplane to understand how well it will fly. The model must be aerodynamically accurate, but it need not be the same as the real thing in other respects, provided that removing these other aspects does not interfere with the accuracy of the model. For example, the lack of interior fittings is obviously not an issue, except to the extent they affect weight distribution.

For a model to be useful, it must also be cheaper to build the model than to build the real thing. In this way, we may experiment with the model, or use it to verify correct behavior, at lower cost than building the real thing.

The mistranslation enters the discussion because there are at least three meanings of “model,” and each one denotes diverse usages and connotes different processes. This is the genesis of the bogus war.

THREE MEANINGS

One denotation for the word “model” is a *sketch*. We sketch out the shape of a wing on the back of a beer mat, show a few lines indicating air flow, and write an equation or two describing how the two interact. The sketch is not complete, nor is it intended to be. The purpose of the sketch is to try out an idea. The sketch is neither maintained nor delivered. Who wants to fly a wet beer mat?

A second denotation for the word is the model as *blueprint*. The physical model of the airplane in a wind tunnel is one example. More commonly, we think of a blueprint as a document describing key properties needed to build the real thing: the blueprint is the embodiment of a plan for construction.

We may treat the blueprint as advisory, building the real thing as conditions dictate, in which case the blueprint will be at odds with the finished system. Alternatively, we could maintain the blueprint and the real thing so they match. Third, we may maintain the blueprint for posterity so the maintainers of the real thing can understand it at a more abstract level. These last two approaches frequently assume delivery of the blueprint to some agent other than the blueprint’s primary users, namely the developers. It is this use that causes the most angst for the agile crowd.

The third denotation for the word is the model as an *executable*. The model of the airplane can be transformed into the real, physical airplane. The transformation requires other inputs, in this case the metal plates, bolts, and screws that make up the body, yet the model is complete in every detail in the one aspect of the problem related to the shape-that-flies. Today, it is literally possible to take a computer rendition of a shape and transform it into instructions for making a shape. The Boeing 777 was built in just this way, entirely inside a computer, as a model.

©2002 by Stephen J. Mellor. All rights reserved.

Under this interpretation of "model" as an executable, a program in a high-level language — Java, say — is a model too. The Java program can be transformed into the real thing (byte code). The builder of the model, the programmer in this case, need not know how a Java compiler works, nor what the compiler does to make a program run. Of course, the byte code produced by the compiler is itself a model that can be replaced by ones and zeroes, one layer of abstraction removed, and those ones and zeroes in turn define the desired behavior of the underlying hardware, yet one more layer of abstraction away.

At each layer, information is added that is inherent to the subject matter involved. Java byte code can be produced only with knowledge of the Java virtual machine, and ones and zeroes can only be produced by making decisions about registers and memory organization.

CONNOTATIONS

The different denotations of "model" affect how we think about using one; that is, each meaning for "model" connotes different processes.

Sketch. Few agile exponents are so rabid as to be unwilling to sketch out their classes and use cases, sometimes called "stories," and perhaps even use UML to do it. There's no fight here: even the most agile use sketches to outline their plans for the code. No one fights over a beer mat.

Blueprints. The connotations of a model-as-blueprint cause conflict. The very idea of a "blueprint" evokes images of factories and manufacturing, together with uncreative drones. "Heavyweight" methods and processes have certainly encouraged the idea of model as blueprint. The manufacturing analogy is drawn repeatedly in the Software Engineering Institute's Capability Maturity Model (CMM), for example. But we know software is a highly creative new-economy thang, not at all like old-fashioned manufacturing.

This contrast is the heart of the agilists' view of models as "heavy." In an environment that is 80% construction and 20% design, like manufacturing, it makes sense to view the blueprint as the plan that is predictive of the construction work to be done. To the contrary, software is known for its creative aspects, more like 80% design and 20% construction. In this case, developers need to be adaptive rather than predictive in their relationship to any model, effectively putting the kibosh on the use of models as blueprints.

Executables. An executable UML model, as the name suggests, is executable. An executable UML model can be compiled into code using a model compiler. The construction of an executable UML model has the same connotations as writing code, though at a higher level of abstraction. When we build an executable UML model, we have described the behavior of our system just as surely as we had when we wrote a program in Java.

Even the most agile use sketches to outline their plans for the code. No one fights over a beer mat.

Many of the principles of Extreme Programming (XP) and the Agile Alliance involve process and customer relationships and their management, not code. As such, the agile process principles for the construction of code apply just as well for the construction of executable models. For those principles that do mention "code," executable UML models serve just the same purpose.

WHAT IS EXECUTABLE UML?

Only recently has UML become executable. Earlier versions of UML had but seven simplistic and incomplete actions, such as Create, SendSignal, and (my personal favorite) "Uninterpreted-String," which taken together do not allow for modeling computation. However, in 1998, the Object Management Group (OMG) released a request for proposal for a Precise Action Semantics. A consortium of companies, led by Project Technology, Inc., Kabira Technologies, and Rational Software, proposed a set of actions sufficiently complete to specify computation, which has recently

©2002 by Stephen J. Mellor. All rights reserved.

been adopted by the OMG. The actions in the submission include LoopAction, ConditionalAction, and actions for collections and non-traditional control structures.

The action semantics specification differs from 3GLs in that actions are specified to manipulate only UML elements, such as attributes and state; to constrain sequence minimally; and to allow the specification of computation independently of the data structure. For a more detailed description of the requirements for the action semantics, see [1, 4].

The action semantics specification does not define a notation. Rather, vendors can specify syntaxes that target specific markets; some could even be graphical. Vendors can also supply in-built operations appropriate to their target usage, such as net-present-value computations, string manipulation, or matrix multiplication.

To atone for the addition of still more stuff into UML, executable UML subsets UML radically, by creating a *profile*. A profile is the UML mechanism for creating defined subsets of the UML for specific contexts. Profiles support the notion of UML as a family of languages.

For the purposes of executable UML, there is no requirement to model the structure of the software, such as deployment diagrams, tasking structures, and the several forms of queuing of messages between classes. These topics are abstracted away and managed instead by an executable UML model compiler, analogous to a programming language compiler.[1]

It would seem, therefore, that the battle is over. Executable UML is the code for the system, and the principles of the Agile Alliance can be argued on their merits, without calling UML onto the battlefield.

CONTINUED MISINTERPRETATION

Unfortunately, this is not yet to be. It seems difficult to achieve, and especially to maintain, recognition of the notion that the model is a complete aspect of the system. After convincing someone that the model is executable, they go right back, half a hour later, to treating the model as blueprint. Some of this maddening behavior comes about because of the several connotations of the word "model," but it also comes about, I believe, because we all have so much invested in code and coding.

The fundamental misunderstanding is the false notion you need to do everything twice. First, you build the model, and then you write the program on the model, making all the decisions about the software structure either in the model graphic or the logic. This false notion adds to the perception that the model is getting in the way: you're trapped inside the graphical structures of the UML model and coding with less flexibility. The urge to express code in the model leads misguided UML-based developers to demand still more modeling constructs in UML to allow them to "say more" and "be more flexible."

After convincing someone that the model is executable, they go right back, half an hour later, to treating the model as blueprint.

This view is akin to writing a program in a high-level language — Java, say — and then adding tags to the various elements of the program to indicate the register to use or the stack location to employ, organizing the program into components to make best use of memory segmentation for a particular hardware configuration. Every new construct in a computer requires extension to the programming language to "say more" and "be more flexible."

It is surely reasonable, if one holds this view, to argue that it's easier just to write the system in assembly code and dispense with programming in Java. It's not obvious what value the program adds, and the entire process seems "heavy" because there are two components to manage: the program and all those tags. Worse, when we need to transport the program to another platform, we have to revise all the tags and reorganize the

[1]For a description of executable UML and how to use it, see [2]. For an example of an executable UML model for a real-time system, see [9].

components to take account of a different organization of physical memory.

Obviously, the argument above should not be taken seriously, yet this is exactly the argument used by code-centric folk to justify not using an executable UML.

High-level programming languages such as Java are intended to abstract away the details of register usage, stack location, and the organization of physical memory. In other words, the programming language makes the programs independent of the hardware platform. Adding tags to the program to do the compiler's job defeats the whole purpose of the exercise.

Similarly, adding code or a multitude of tags to a UML model defeats the purpose of executable UML, which is to make the executable UML model *software platform–independent*. In other words, the executable UML modeler expresses the content of an application without any reference to its implementation environment. There is no mention of CORBA, no mention of JavaBeans, no mention even of distribution of classes into tasks. The model compiler takes care of targeting the software platform just as a programming language compiler takes care of targeting the hardware platform on which the program runs.

MODEL COMPILERS

A model compiler is like a programming language compiler. It takes a program, written in an executable language, and turns the program into a different representation that has the same functional behavior. There are multiple model compilers, each targeting a different software platform.

For example, one model compiler might apply to applications that can benefit from a high-volume, distributed, concurrent, transaction-safe implementation with rollback of incomplete transactions. This kind of model compiler is suitable for telephone billing and stock trades, for example, but it is not suitable for an embedded system [3].

Alternatively, another model compiler might apply to embedded systems with very tight memory and speed requirements. There are no tasks and no operating system. The resulting code runs directly on the silicon [8].

Still another model compiler might apply to problems that require multitasking, concurrency, and persistence but be optimized in the sense that it does not provide sophisticated capabilities such as rollback [7].

The programming language produced by the compiler and other services such as JavaBeans and CORBA would need to be known only if the generated code has to interface with other components.

The model compiler makes these decisions about the software platform. The job of the developer is to select the correct model compiler for the application, based on its performance characteristics.

When invited to attend the inaugural meeting of what became the Agile Alliance, I introduced myself as a "spy."

UML AND MODELING SOFTWARE STRUCTURE

A reasonable objection to this entire line of reasoning is that UML has continued to grow, especially to include additional modeling elements to capture software structure.[2] Because UML is intended to be a family of languages that can support the visualization of software artifacts, these additions are inevitable. However, by carefully layering the semantics of the system-modeling portion of UML, we can restrict the deleterious effects of this growth to those who prefer to use heavyweight processes.[3]

ENDING THE WAR

I have recently become a peacenik. When invited to attend the inaugural meeting of what became the Agile Alliance, I introduced myself as a "spy." As an author of two methods that rely heavily on modeling, one of which uses UML, I felt compelled to ascertain what

[2]See, for example, a response to the UML 2.0 Superstructure RFP [6].

[3]For example, see another response to the UML 2.0 Superstructure RFP [5] that layers the definition of UML.

©2002 by Stephen J. Mellor. All rights reserved.

the opposition was up to, all the better to derail their evil plans. Yet as I read more and listened carefully, I found myself in agreement with much of what was said, as well as hearing many echoes from my own work.

There are elements of the agile movement I still find distasteful. The allergy to "documents" is one. The most important document of all is the one that describes the design not chosen and the rationale for that decision. That simply cannot be self-documented in the code. My skin crawls at some of the touchy-feely "interactions" talk that seems to require the developer to be emotionally one with the customer, and as for the communist notions of collective ownership and the 40-hour week, well, these are cause for, er, heated discussion. But with respect to the use of agile processes and executable UML, there should only be peace.

REFERENCES AND WEB SOURCES

1. Mellor, Stephen J., Stephen R. Tockey, Rodolphe Arthaud, and Philippe LeBlanc. *An Action Language for UML: Proposal for a Precise Execution Semantics, "UML 98."*

2. Mellor, Stephen J. and Marc J. Balcer. *Executable UML: A Foundation of Model-Driven Architecture*. Addison-Wesley, forthcoming 2002.

3. Kabira (www.kabira.com).

4. Object Management Group (OMG). Action Semantics for the UML, Request for Proposal. OMG, 1998 (http://cgi.omg.org/cgi-bin/doc?ad/98-11-01).

5. Object Management Group (OMG). Updated Joint Initial UML 2.0 Infrastructure Submission. OMG, 2001 (http://cgi.omg.org/cgi-bin/doc?ad/01-08-34).

6. Object Management Group (OMG). Updated UML2 Infrastructure Joint Initial Submission. OMG, 2001 (http://cgi.omg.org/cgi-bin/doc?ad/01-09-02).

7. Project Technology, Inc. DesignPoint MC-2020 (www.projtech.com/prods/mc/mc2020.html).

8. Project Technology, Inc. DesignPoint MC-3020 (www.projtech.com/prods/mc/mc3020.html).

9. Starr, Leon. *Executable UML: A Case Study*, Model Integration, LLC, 2001.

Stephen J. Mellor is the coauthor of two methods, one of which can be used in an agile manner with UML.

His work has emphasized the engineering of high-assurance systems, especially in real-time and embedded environments. Most recently, Mr. Mellor has been working to make UML executable by, first, incorporating actions into UML, and, second, as coauthor of the upcoming book Executable UML: The Handbook.

Mr. Mellor is cofounder, with the late Sally Shlaer, of Project Technology, Inc., a company focused on tools to compile UML models, where he now serves as vice president.

In his copious spare time, he is also a member of the IEEE Software Industrial Advisory Board.

Mr. Mellor can be reached at Project Technology, Inc., 7400 N. Oracle Road, Suite 365, Tuscon, AZ 85704. Tel: +1 520 544 2881; Fax: +1 520 544 2912; E-mail: steve@projtech.com.

A Resounding "Yes" to Agile Processes — But Also to More

by Ivar Jacobson

Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to *appropriately* respond to changes. Change is what software development is very much about [1]. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product the team builds or the project that creates the product. Support for changes should be built into everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people and their ability to collaborate are critical to the success of the project.

For the purpose of illustration, here is an agile project as described by Svante Lidman, a project manager at the Swedish software company Jaczone AB:[1]

I took over the job as project manager in November 2000, at which time the team consisted of four people. They had developed a simple prototype barely demonstrating the feasibility of the idea: using intelligent agents to support the individual developers in software development. The product would include a knowledge base described as a large number of rules formulated in a rule language, a compiler generating executable code from the rule language, and a rule engine — not exactly something I had done before. The company was a start-up company funded with seed money only. We needed to get a first workable product through the doors as quickly as possible, to be extremely focused.

In the first phase, we developed a vision in writing for the first product release. We needed as a team to know where we wanted to go. We also needed to get consensus on this vision with sales so that we would get something we believed we could sell. We were, in particular, explicit on what we wouldn't do. We knew that the released product would not be all we said; it would contain something less and maybe something more. This took about a month, and I grew the team to seven people; four of the team developed the compiler, the rule engine, and generic GUI, and three of them developed the knowledge base. The team members were very competent in their respective roles.

In the second phase, we developed a skeleton executable system from which we could grow to the first product. We worked out the software architecture. We decided on a software platform (C# and .NET). We described each use case and made a simple object model using UML (we knew Rational Rose, so we used that). We worked with mock-ups and storyboarding to design the user interface — the team had now grown, with the addition of a creative and experienced UI designer. We also identified all significant risks and worked through them all. To verify our work, we implemented and were able to run the use cases that mitigated the risks. This took about three months, during which time we iterated the solution twice.

There was some significant rework during this period based on customer interactions and for technical reasons. That was no surprise to me. We had to adapt our solution as we better understood what we were doing before we thought we had something we could grow from. At the end of this phase, we felt comfortable in committing to a release date. We still didn't know exactly what we would deliver, but we knew it much better than after the first phase.

The third phase was basically growing the product in several internal releases. Thanks to having spent so much effort on the skeleton system in the second phase, we could grow the product quite smoothly. That doesn't mean that we didn't have to do some changes, but no major ones. We could move forward as we had committed. We didn't do all we said we would do at the end of phase two, but nothing we cut off was very important. Since I was absolutely committed to releasing a "defect-free" product for our first customers, we spent the necessary time on peer reviewing and testing of each internal release.

[1]For the record, I am a member of Jaczone AB's advisory board.

©2002 by Rational Software Corporation. All rights reserved.

In the fourth phase, we released the product to a handful of customers committed through contracts to work with the product. During this phase we didn't expect to find many bugs — and we didn't. We wanted to know if the excitement that customers felt when we demonstrated the product would hold during real use. Does the product give the value that we expected? Is there anything more we need to do before making the product generally available? This is where we are right now.

The process that Lidman describes is a light version of the Rational Unified Process (RUP) [2, 3]. It was chosen because Lidman and his team didn't need to invent something new; they believed that it would create working software fast with very little documentation. Being based on the RUP, the process itself was designed to be changed and to support changes in the product as normal occurrences. Moreover, they wanted the product to grow and the team that developed it to be able to grow in many dimensions: size, geographic distribution, customers, and so on. These are some of the properties I require of an agile process.

As the title of this article suggests, I am a strong believer in an agile process. No one can afford to develop software by slavishly following a predefined process. Instead, the project should follow a process that helps the team to achieve its goal, but that project may be described using a predefined process. The key is that you continually adjust the process as you go by reflecting on what you're learning, what works, and what doesn't. In this article, I will discuss some other properties of a good process that go beyond what is commonly thought of as "agility."

No one can afford to develop software by slavishly following a predefined process.

A PROCESS SHOULD EMPOWER YOU TO FOCUS ON CREATIVITY

I believe that software development is the most creative process that we humans have ever invented. I also believe that processes or tools can never replace true creativity. However, not everything we do when developing software is creative.

In every project, a large portion of the work done by developers is not genuinely creative — it is tedious, routine work that we should try to automate. The problem is that the creative and the routine work are interspersed in micro-steps, each such step only lasting from maybe tens of seconds to tens of minutes. Since these two kinds of works are interleaved, the developers may still have the feeling of being engaged in creative work, but fundamentally they are not.

Some of you would probably argue that, in fact, you don't do the unnecessary work. You focus on solving the business problem and ignore much of the other work that does not deliver business value. This is, of course, partly true.

However, even in projects where the focus is on code, people have different approaches to good coding standards. What most people thought was good code some years ago is considered bad code today. Thus people spend time arguing these things. With a strong, good leader, these problems may be smaller. However, most teams won't have such a leader.

In every project, a large portion of the work done by developers is not genuinely creative — it is tedious, routine work that we should try to automate.

Aside from process issues, we also spend considerable time on many other small, technical issues. For instance, if you want to apply a pattern, there are many small steps you have to go through before you have instantiated the pattern. These small steps could, with proper tooling, be reduced to almost a single step.

I don't think anyone would argue with the fact that some portion of the work developers do is noncreative. I have discussed how large the portion is with colleagues from several camps, and based on my own experience and these discussions, I believe it is as large as 80%. That number assumes a light programming environment. There is no significant difference if you develop with or without reuse of existing components. Reuse should reduce the total work, but it doesn't significantly reduce the non-creative portion of the work. I have only seen one very old scientific study on this subject and would welcome one based on current practices and tools.

To make software development truly agile, we need to attack and reduce this 80% of the work effort. I don't think we can eliminate it entirely, but we may be able to get from 20/80 (20% creative content and 80% non-creative) to 80/20. The way to attack it is to understand *in depth,* not just on the surface, what is not genuinely creative. Then, once we understand it, we can eliminate it by training, mentoring, and proper software tools. Developers will be able to communicate and collaborate creatively, supported by new or improved tools to verify that they are consistent, complete, and correct.

If developers can rely on such tools to eliminate much of their not genuinely creative work, they will be in a much better position to maximize the time they spend on developing business solutions. This use of tools will also increase the quality of the developers' lives. They will be able to spend more time doing the creative work they enjoy and less time on the routine or repetitive work they dread. This would enable the agility we all are looking for. It would empower people to do the maximum amount of creative work.

A GOOD PROCESS ALLOWS YOU TO LEARN AS YOU GO — WITHOUT SLOWING DOWN THE PROJECT

Developing software has never been as hard as it is today. You need to have a knowledge base that is larger than ever before. You need to know about operating systems, database management systems, programming languages and environments, system software, middleware, patterns, object-oriented design, component-based development, and distributed systems. You also need to know a software process, a modeling language, and all kinds of tools. And if you succeed in learning something, you can be sure it will soon change. Change is the famous constant in software!

There is simply no way you will be able to learn all this before you start working on a project. You have to learn a little before, but most of it you will need to learn as you go.

The way we learn is very different from individual to individual. Some learn different things better than others. We learn at different speeds, and in different ways. Some people learn better by reading, and others by listening.

Who will teach us? Teachers give classes and mentors participate in projects with the role of being available to help the team while going. As we know, not everyone is a good teacher or a good mentor, and even fewer can be both. Good teachers and good mentors are very valuable but not easy to find.

Since we are all different, we should ideally be allowed to learn in our own way as we go — from teachers, from mentors, or from reading. Traditional education has consisted of attending some basic classes, possibly reading a textbook, and then learning by watching or being instructed by a more experienced person.

Just taking some classes on different subjects — for instance, in using a programming environment, in a methodology, in testing — is a poor start. Software development is much more than that. Going away to different classes that are *not correlated* with a bigger picture — such as a software process — is not efficient.

Learning from a more experienced person is very effective, particularly if that person is a mentor. However, it is hard to get good mentors — competent individuals with unusual personal skills. There is always the risk that a good mentor may be given other project responsibilities or that time pressure may force him or her to leave the mentor role. Moreover, someone taking on the role of mentor may veer into inventing his or her own process, which certainly will slow down the project.

Thus having a mentor on site (maybe only part-time) is a good investment, but we need to reduce the risks I just mentioned. What should we do?

©2002 by Rational Software Corporation. All rights reserved.

In 1986, I had a dream. I wanted to create a knowledge base, not just a book,[2] for everyone that could play a role in a modern software project: component-based, object-oriented, use case–driven … you name it. Whereas a book may be a nice introduction, it cannot possibly provide depth in all areas. I also wanted this knowledge to grow as we learned more, to change as we better understood how to develop software — to enable us to throw away what was bad and incorporate new, good experiences. Moreover, the knowledge base should be able to adapt itself to new technologies as they became available. Since then, technology has exploded — the Internet, J2EE, .NET, new middleware, and much more. Later versions of the knowledge base should accommodate such innovations.

In other words, what I had in mind was to derive knowledge equivalent to 15-20 books dealing with relevant subjects like requirements, architecture, design and implementation, user experience design, business engineering, testing, software reuse, configuration management, and project management. For ease of use, this information would be written in a common style with consistent terminology. Of course, no single individual would read all of these book equivalents right off the bat. Rather, he or she would get an overview of the whole approach in order to understand how to go from a business problem to a deployed system. The individual developer would have the base on which to grow when he or she went into a project that had already started.

Is the RUP a light process? No, the RUP is very rich (in content), but you can use it in a very light way.

To make that dream a reality, the Objectory [4] process was created. Objectory grew for more than 10 years and evolved into the RUP in 1998. What I just described has become a reality in the RUP.

With the RUP, you can go into any depth you wish. It is possible for you to skim the surface in learning about a particular topic, or it is possible to delve deeper to get very detailed information. It all depends on your situation and needs. You have it at your fingertips, just in time. You can often avoid having to ask questions that no one has the time to answer and can avoid asking repetitious, dumb questions that consume the time of good mentors. You don't need to worry about being disciplined by the teacher/mentor (shades of second grade!) until you do it right. Is the RUP a light process? No, the RUP is very rich (in content), but you can use it in a very light way. (I will return to this point in the next section.) Moreover, since you can learn as you go without slowing down the project, it is an agile process.

Despite all this, the authors of the RUP books will probably never win the Nobel Prize!

A "GOOD" PROCESS ALLOWS YOU TO BE FAST — WITHOUT HAVING TO REINVENT THE WHEEL

For years I have claimed — with perhaps a twinkle in my eye — that the fastest way to create software is not to develop anything yourself but to reuse something that already works. This approach is not just very fast, it is also cheap. It delivers software that works. In practice, however, you may still need to develop something new in many situations — at least the "glue" between the components that you can reuse.

We don't develop our own operating systems, database systems, programming languages, and programming environments anymore. We usually start with a software platform and some middleware as a base — not much more. However, much more can be reused.

Process. You shouldn't have to reinvent a process. Instead, you should use a well-proven process designed to be reused. This is what we at Rational call a process framework. A process framework is both generic and specific. It is generic by being based on a number of sound best practices;

[2]I feel that books (such as my own books) can give no more than an overview of an approach. They don't tell the entire story, and they don't enable the team to do the entire job. There is much more to it than that.

these best practices must (if applicable) be integrated and provide a common vocabulary and presentation. The process framework is specific in that it can be successfully specialized or tailored to any application area, to any type of product (pacemakers, claim systems, auction systems, banking systems, command and control systems, etc.), and to any platform being used. It must also reflect the competencies of the developers, the maturity and size of the organization, and whether the work is distributed or not.

Since the process framework is generic, it contains more activities and artifacts than any individual development team would apply. For a particular software product, you create a process by picking and choosing from the framework.

This is what the RUP is about. It is a process framework from which your team, with the help of special tools, can create its own process.

Software. You shouldn't have to reinvent the same software over and over again. You should use a process that helps you harvest components and incorporate existing components — legacy systems or other reusable components — into your design (with the appropriate tools to do so, of course).

If the software community is ever going to be more efficient — developing good software fast with the desired quality — we need tools to help us do more by doing less.

What is unique about this process framework approach is that you can get a product team quickly up to speed. The team can focus on solving the business problem instead of having never-ending discussions about what needs to be done — that just slows down the project. You may still need a mentor who assists the team as it goes, and you still need competent people on the team. The process doesn't replace them, it empowers them.

A GOOD PROCESS USES TOOLS TO DO MORE BY DOING LESS

Whatever you do, in order to be efficient, you need good tools. Good tools are tools that are developed integral with your process. The process and the tools go together. To take a real-world example, if you want your carpenter to drive nails in a wall, he needs a hammer, not a screwdriver. If you want your baby to eat on her own, give her a spoon, not a knife. The same goes for software.

We obviously need tools for programming (coding, debugging, compiling, etc.). Let me call them light tools (even if I can hear many people object to my calling these tools "light"). Light tools must go with a light process to be agile. Even so, we have been using light tools for more than 40 years now and got "implicit" requirements, "implicit" architecture, "implicit" design, self-documenting code, and we tested like hell. "Implicit" refers to all the diagrams we drew on the whiteboard that were thrown away by the cleaner.[3]

Sure, there were some successes — thanks to the incredibly good people that made them happen. However, incredibly smart people are a scarce resource. Moreover, projects that *didn't* succeed were in the vast majority.

If the software community is ever going to be more efficient — developing good software fast with the desired quality — we need tools to help us do more by doing less. Using only light tools will hold us back where we have been all these years.

I agree with the idea that we want a process that is very light. In fact, I want a much lighter process than I have heard people speak about. It will be light because tools will do the job — you will be doing more by doing less. I want much more in my tools than the light tools that go with a light process. Thus, my tools are not really "light" tools, and to tell the truth, my process is in reality rich. However, thanks to my not-so-light tools — my powerful tools — that go with the rich process, the process will be *perceived* as very light. And that is all that counts, as long as the process is still agile.

[3]Parenthetically, I can't understand — given that we have good tools — why we would throw away something that we needed to create the first release of a software product. What we needed for that release we will probably need even more for the next release.

©2002 by Rational Software Corporation. All rights reserved.

These tools were part of my 1986 dream. Such tools would help you to create requirements, use cases, architecture, design, tests — everything a developer does. What you would create is not documentation but models. When you created a use case, for instance, it would be part of a model that would actually become a collaboration among objects and eventually code and test cases. With proper tools, we would get to the point that "the models are the code," or "the code is the models." Of course, all these work products would be explicit. They would not disappear as the whiteboard drawings did when the project was over. You would have them for the next project.

Thus, the tools do the job for you. Once you have created the architecture (in the form of the architectural baseline), you also have executable code. Once you have created the design, you have 80% or more of the code that implements it — without having had to think about it. The majority of your test cases follow from your design. Integration tests are created from your use cases. And on we can go. You use blueprints to express your architecture and your design in a way similar to what engineers have been doing for hundreds of years. We have a standard for making these blueprints — we call it a modeling standard (i.e., UML). Of course, you use code, too, but just to express what is best expressed by code, namely actions (or operations, methods).

Does this sound like magic? Well, we do some of this today. We get better with every release. It is part of our vision for the future.

An agile process relies on tools for whatever you do from "womb to tomb."

We use the RUP, UML, and tools to support it. We have created light specializations of the RUP, which are supported by light tools. There are even lighter versions, but we will not suggest that you water down the process. Rather, we will convince you to keep our best practices: develop iteratively, manage your configurations, find the right architecture first, refactor when you don't succeed, and so on. Within these invariants, your team can create the process it wants. It will be a rich and light process, a process that is perceived to be light even though below the surface it is very rich. On top of all this, you can scale up your process to a larger team, to a team of teams distributed around the world, and to a product with many different features allowing a lot of customization. Still, your light version will not and shall not be burdened by this ability to scale up. We call this "right-sizing" the process.

An agile process relies on tools for whatever you do from "womb to tomb." An agile process is rich and light — perceived to be light, even if it is very rich. It is thanks to the rigorous tools that go with the rich process that the process becomes light. You will be able to do more by doing less. And there is more to come.

LOOK OUT FOR THE NEXT BIG THING

A process framework with tools significantly empowers you as a developer. The knowledge base about good software that comes with the RUP continues to grow, the process framework becomes easier to understand and to use, and more and better tools evolve. You should be making this effort.

However, many others will not make these investments. They will jump for light processes with light tools and assume that good people can work wonders. They will rely on light development practices that have been around under other names for decades. The problem is that software development will still be hard. In fact, it will become even harder. We will see many software companies fail. For those others that won't make these investments, there is to my mind only one way out that dramatically — not just marginally — changes this picture. It is the next big thing.

In 1980, I wrote a letter to the president of Ericsson, the essence of which was that the component-based approach we were then using would evolve into a world standard. We should now go forward by (1) developing our modeling language, (2) evolving our process into a world standard supported by tools, and (3) developing expert system support on top of process, language, and tools.

I realized that expert systems could not be powerful enough to do the job (1) before we got a standard modeling language like UML, (2) before we had a

Twenty years later, we can take the next big step. It is "intelligent" agents.

knowledge base like the one in the RUP, and (3) before we had supporting tools. Twenty years later, we are there; we can take the next big step. It is "intelligent" agents.

We can put a layer of agents on top of the RUP and its tools. These agents are programmed to trigger for different events. They recognize patterns and anti-patterns; they suggest resolutions. They assist the developer dynamically in using the knowledge base. They suggest the next micro-activities to be performed, many of which follow from the context of the work being done and do not require anything creative by the developer. The developer is empowered to "think" and not be bothered by all the small pieces of work that today constitute what we usually call "the hard work."

The process will be very agile. It will be perceived as a very light process, but it is based on very rigorous tools. Obviously, you learn as you go. It can easily adapt to different roles that people play, and it can scale up as needed.

Does this "next big thing" sound like science fiction? Maybe, but I don't think it is far ahead.[4] Am I saying this because I am personally involved in it? I can understand that I risk my credibility, but I have to take this risk. I wouldn't support this without believing in it and believing that we can get there soon. And we can only get there thanks to the existence of UML, the RUP, and supporting tools.

[4]See www.jaczone.com.

Let me conclude by saying that although I have in this article primarily discussed processes from an agile perspective, we expect much more of a process than just agility. We want it to empower you to solve your business problems and to satisfy your users. We want your systems not only to work 24/7, but to grow gracefully as the world changes. To get there, you need a process framework, a modeling language, and supporting tools, not the least of which will be the "next big thing," intelligent agents.

REFERENCES

1. Jacobson, I. "Language Support for Changeable Large Real-Time Systems." In *Proceedings of OOPSLA'86,* special issue of *SIGPLAN Notices,* Vol. 21, No. 11 (November 1986), pp. 377-384.

2. Jacobson, I. "Object-Oriented Development in an Industrial Environment." In *Proceedings of OOPSLA'87*, special issue of *SIGPLAN Notices*, Vol. 22, No.12 (December 1987), pp. 183-191.

3. Jacobson, I., G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

4. Kruchten, P. *The Rational Unified Process — An Introduction.* 2nd ed. Addison-Wesley, 2000 (see also www.rational.com/products/rup/index.jsp).

Dr. Ivar Jacobson is vice president of process strategy for Rational Software Corporation. Dr. Jacobson is the founder of Objectory AB in Sweden, which merged with Rational in 1995. He is one of the three original designers of the Unified Modeling Language (UML), which was officially adopted as a standard by the Object Management Group (OMG) in 1997.

Dr. Jacobson's seminal contributions center in five major areas, beginning with the architecture-centric approach to software development. Developed at Ericsson beginning in 1967, this approach was adopted by the telecommunication standard SDL. Emphasizing use case–driven development, the process was an early example of component-based development. By 1987, Dr. Jacobson had developed this development process into Objectory, which he expanded in the early 1990s to include business engineering. After several more generations, and after joining forces with Rational, Objectory evolved in 1998 into the Rational Unified Process (RUP).

Dr. Jacobson is the principal author of five influential and best-selling books: Object-Oriented Software Engineering — A Use Case Driven Approach*;* The Object Advantage — Business Process Reengineering with Object Technology*;* Software Reuse: Architecture, Process, and Organization for Business Success*;* The Unified Software Development Process*; and* The Road to the Unified Software Development Process*. He is the coauthor, with Grady Booch and Jim Rumbaugh, of* The Unified Modeling Language User's Guide *and* The Unified Modeling Language Reference Manual*.*

Dr. Jacobson can be reached at Rational Software Corporation, 18880 Homestead Road, Cupertino, CA 95014. Tel: +1 408 863 9900; E-mail: ivar@rational.com.

©2002 by Rational Software Corporation. All rights reserved.

Agile or Rigorous OO Methodologies: Getting the Best of Both Worlds

by Brian Henderson-Sellers

Agile, or lightweight, OO methodologies appear to offer a new approach to building software. They appeared a few years ago as a reaction to the perceived "heaviness" of some of the more "rigorous" OO methodologies existing at that time. While proponents of each extreme are adamant about the rightness of their cause, is it possible that each, in being right, is only right in certain specialized conditions? Indeed, it seems an impossible dream that we can create a single methodology that will be perfect in all circumstances: from safety-critical systems to systems with lower precision demands, from software built by a team of "whiz kids" to one built by the regulars.

In this article, I present a framework in which the rigorous and the agile can coexist; a framework that itself provides a flexible platform from which a wide range of different kinds of processes can be created, including both lightweight and heavyweight. The OPEN Process Framework (OPF) [3] is an exemplar of this kind of approach, often called method engineering or situational method engineering [10], and I will show how the OPF can be used to configure process models for a wide variety of situations.

ONE SIZE DOESN'T FIT ALL

The needs of software developers in today's rapidly changing world of technology are diverse. The applications they create to address specific business-level needs and opportunities are diverse. And the skills sets of the individuals in the software development teams are diverse.

We shouldn't therefore expect that a *single process* would ever be perfect for every company. The perfect process (for you) is one that ideally suits your way of working, your degree of "ceremony," your organizational culture, the degree of safety criticality of the domain in which you work, and so on — not to mention the *people* involved in executing the process. The process is, after all, only going to be successful if the participants in all the software development teams like the process and feel that it helps them in their daily work. Good people may well need less help than less good people. But people move on, and the notion of an organizational process is one that is embodied in the knowledge of the organization rather than the knowledge of individuals. Organizational knowledge leads to a higher CMM or SPICE level of process capability than does individual knowledge.

In other words, one size doesn't fit all, and seeking a single process for software development would appear to be a fruitless occupation.

In order to get your personalized "perfect process," then, what are the options? There are several alternatives that the software industry (here, the OO part of it) can pursue. Consultants can create from scratch a perfect process for each individual organization. This is hardly feasible and not cost effective. A second approach is to create a suite of "standard processes," one for each *type* of organization/software product/skills set (see Figure 1). This is essentially the mindset that has created the agile versus rigorous debate. Methodologies are pigeonholed and labeled as "lightweight" or "heavyweight" and can never escape from these boxes. Their scope has been limited to only a subset of the possible applications.

An obvious problem with the suite of processes in Figure 1 is that if your organizational characteristics change and you move from a situation ideally suited to, say, XP to a different situation where RUP would be more appropriate, then

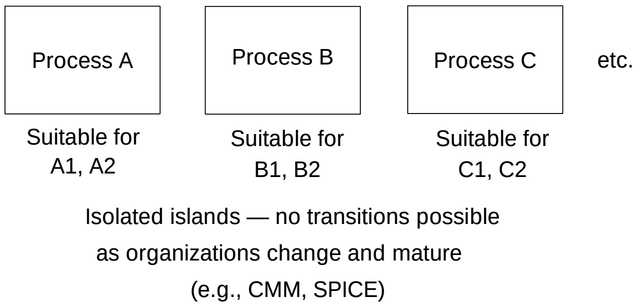

Process A
Process B
Process C
etc.
Suitable for A1, A2
Suitable for B1, B2
Suitable for C1, C2
Isolated islands — no transitions possible as organizations change and mature (e.g., CMM, SPICE)

Figure 1 — A suite of processes, each only suitable for a highly restricted set of circumstances (after [7]).

you have to do a major process refit and literally throw one process out and introduce another. There is no chance for software process improvement.

The third option is the one I advocate here — creating a set of process components, underpinning them with a metamodel, and using these components, as in a constructor set, to piece together the complete methodology/process. Since such a constructor set is equally useful in building something small, by using only a few components, or building something large, by using almost all the process components, then changes in the organization can be accommodated within the single process framework. Software process improvement is also strongly supported by such a flexible approach.

METHOD ENGINEERING

Method engineering focuses on creating process components or fragments [2] and then putting them together in such a way as to support individual circumstances [11]. This requires one, preferably two, critical elements: the existence of a repository of process component descriptions and for each of these process components to be an instance of an element in a metamodel. Process construction is then a matter of selecting from the process components and creating the configured and "personalized" process, which may itself then be further tailored (i.e., minor modifications to the specifications as abstracted from the repository).

EXEMPLIFYING METHOD ENGINEERING USING THE OPF

One good example (there are many others) of the third option described above is the OPF, described most recently by Donald Firesmith and myself [3]. The OPF contains three major elements:

- A *metamodel*, which defines the rules of the process components
- A *repository* of process components, which are instances of each metalevel process element
- A set of *usage guidelines* that are helpful for constructing and tailoring the process itself to a specific problem and organizational context

In the metamodel of the OPF, there are three major metaclasses: producer, work unit, and work product; these are supplemented by two other top-level classes: stage and language (see Figure 2).

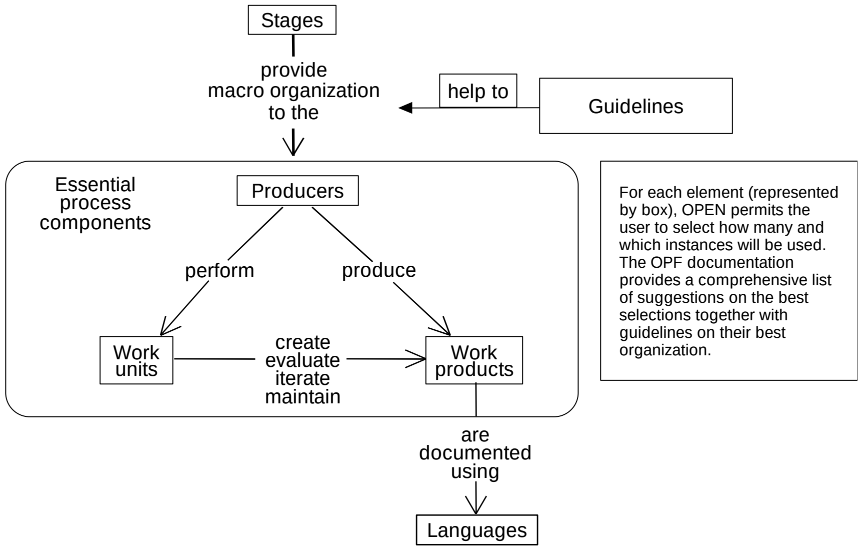

Stages
provide macro organization to the
help to
Guidelines
Essential process components
Producers
perform
produce
Work units
create evaluate iterate maintain
Work products
are documented using
Languages
For each element (represented by box), OPEN permits the user to select how many and which instances will be used. The OPF documentation provides a comprehensive list of suggestions on the best selections together with guidelines on their best organization.

Figure 2 — The basic metatypes of the OPEN Process Framework (OPF) (after [3]).

©2002 by Brian Henderson-Sellers. All rights reserved.

Producers use work units to create work products. Calendar time sequencing is done by the application of stages, and languages are used to document the resultant work products.

Each of the five top-level classes in Figure 2 has subclasses (still within the M2 or metamodel level). There are many kinds of work products: for instance, documents, metrics, requirements, architectures, components, applications, and so on. Subtypes of the work unit metaclass are fewer, the main ones being activity, task, and technique. Producers are described as either being "direct producers" or "indirect producers," the former being either tools or persons playing roles, the latter being organizations and teams. Finally, stages may be phases, cycles, builds, or milestones, and languages include natural languages, modeling languages, and implementation languages.

Each of the above metaclasses generates instances, which are "process components" stored in the model-level (or M1 level) repository (see Figure 3). This repository thus contains myriad possible process elements, all fully described and ready to be used in creating your process. In addition, users can add extra components from their own best practices. The process engineer within the organization will select appropriate process components and put them together to form an actual process, or process instance, within the given guidelines (see below) and ensure compatibility between all portions of the process. This is referred to as process construction (see Figure 4).

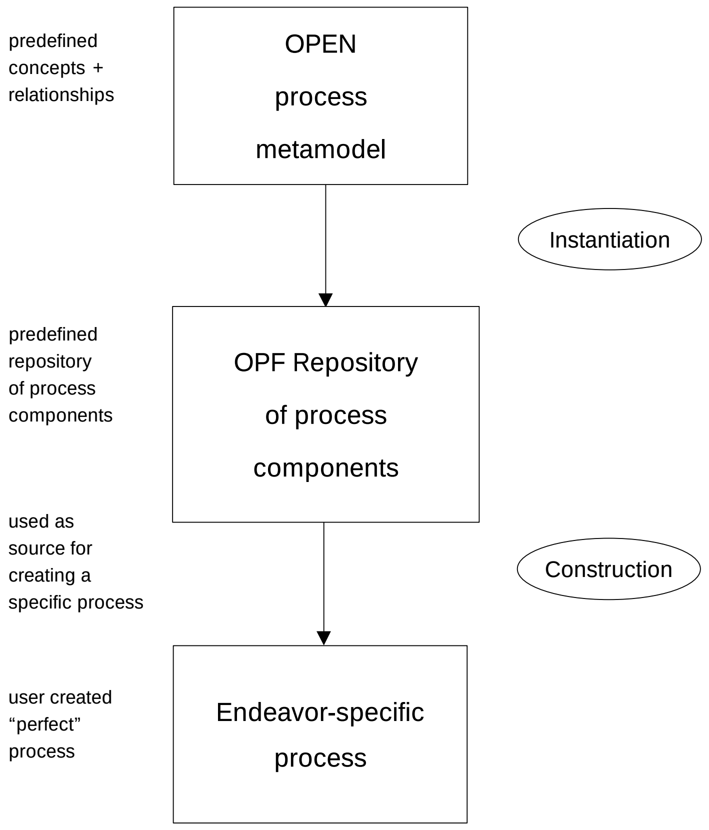

predefined concepts + relationships
OPEN process metamodel
Instantiation
predefined repository of process components
OPF Repository of process components
used as source for creating a specific process
Construction
user created "perfect" process
Endeavor-specific process

Figure 3 — Creating an endeavor-specific (enterprise or project) process (after [7]).

Since many such process instances can be created from the one framework, we can create a family of processes for various industry demands. These might range from critical safety software to background financial processing, real-time stock market modeling, and so on. Construction decisions need to take into account myriad variables pertinent to the development organization. These include, but are not restricted to, CMM level of maturity, available skills, available tools, criticality of the software, quality level desired, time scales, degree of high/low ceremony, and size of the team.

There are also a number of usage guidelines provided as part of the OPF to help the inhouse process engineer. They are divided into three parts: construction guidelines, tailoring guidelines, and, of less interest here, extension guidelines.

Since many such process instances can be created from the one framework, we can create a family of processes for various industry demands.

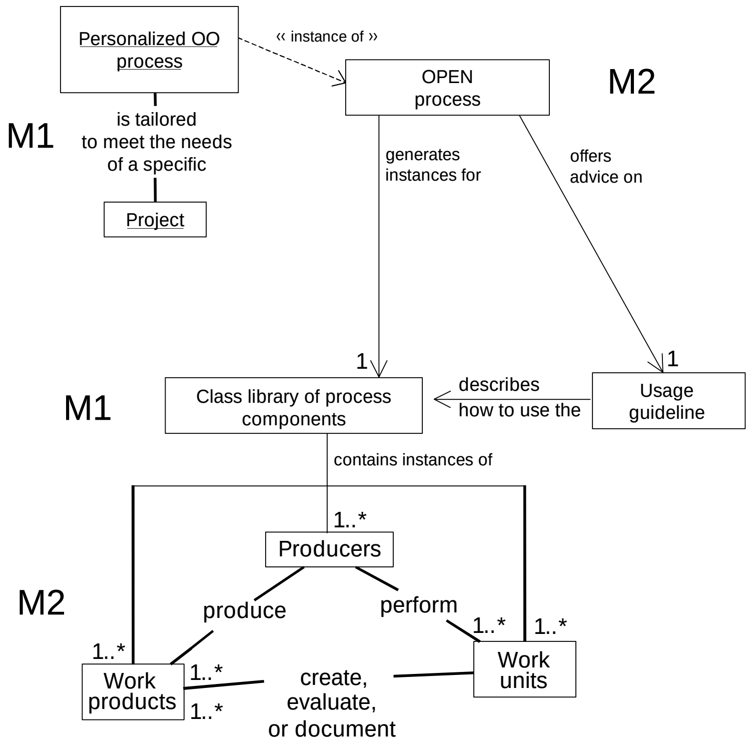


Figure 4 — Process component instances are generated from the M2 metamodel (OPF) and stored in a repository (M1 level) prior to construction to form individually configured and tailored process instances.

Construction Guidelines

A process construction guideline helps process engineers both to instantiate, when necessary, the development process framework (metamodel) to create process components and also to select the best process components (from the repository) in order to create the process itself. Specifically, guidance is provided on the selection of appropriate work products, producers, and work units, as well as advising on how to allocate tasks and associated techniques to producers and how to group the tasks into workflows, activities, and so on. Finally, developmental stages (including phases and lifecycles) are chosen.

Tailoring Guidelines

Once the process framework has been instantiated and use on real projects has begun, the development team frequently recognizes the necessity to fine-tune the process. This is called tailoring; the OPF contains tailoring guidelines to help process engineers undertake these minor modifications of the instantiated process components.

Extension Guidelines

Finally, extension guidelines are provided to permit alterations to the metamodel itself. This is not part of normal process construction and therefore outside the scope of this article.[1]

[1]For further details, see [3].

The advantages of the "constructor set" approach are manifold. The constructed process has been created *by members of the organization* themselves. They thus have "ownership" of the resulting OPEN process. Everyone has bought into the process because they have had the opportunity to contribute to its formation throughout the period of process construction. In addition to local ownership, there is also global support, because the framework and repository from which *your* process has been constructed are identical to the ones used by many other companies worldwide. You can thus participate readily in user group meetings,[2] and your new hires will potentially have the necessary skills gained from a worldwide "community standard" rather than the use of an idiosyncratic, inhouse approach.

As I have shown here, the OPEN approach supports a high degree of flexibility in process construction. Use of process components from the repository (created by both the OPEN Consortium members and the end users) leads to a highly tuned process that is suited to your individual, local needs, which may be agile or heavyweight.

[2]For instance, we regularly hold an OPEN "birds of a feather" gathering at international conferences such as OOPSLA, and there is a listserv discussion group hosted on the OPEN Web site at www.open.org.au.

©2002 by Brian Henderson-Sellers. All rights reserved.

EXAMPLE PROCESSES

To construct an agile or a rigorous instantiation of the OPF, there are a number of features you must recognize and steps you must follow. Firstly, the overall architecture of OPEN is one in which a number of activities (types of work unit) are connected together to form a problem-specific or organization-specific OPEN process, as described above. Permissible sequences of events or paths through the process are prespecified by the organization's process engineer to meet the demands of the endeavor (see Figure 3). A project team can only move between activities when it has met both the postcondition of the activity about to be left and the precondition of the activity about to be started. This is the contract-driven lifecycle model used by OPEN [4]. The process engineer creates these contracts, and they consist of testing criteria, deliverables, quality standards, and so on.

Activities in OPEN, which are modeled as objects, are coarse-grained descriptions of what needs to be done. The combination of activity objects and forward and backward linkages forms the process. The actual scheduling comes from the planning activities and tasks and the project management elements embodied in appropriate tasks. A specific process instantiation may concentrate on a single project or on a program of several projects, which introduces new foci such as domain modeling and reuse, as well as resource allocation.

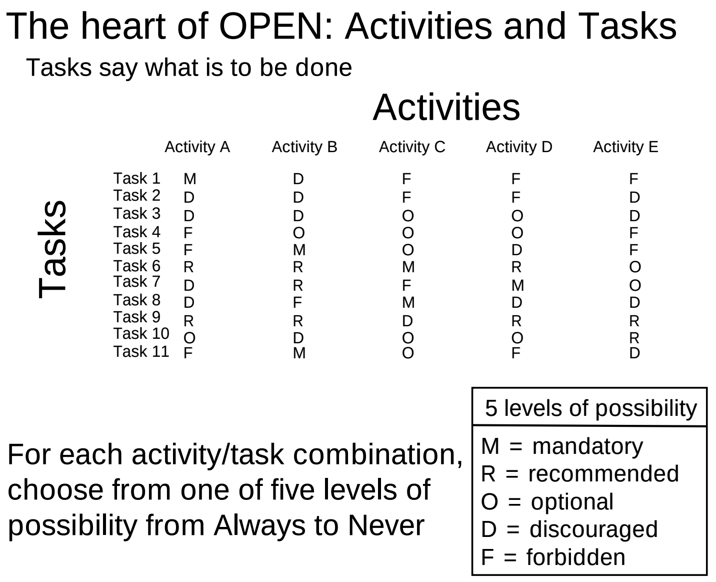
Having selected appropriate activities, the project team can then identify what tasks might be appropriate. While OPEN's tasks are like activities in the sense that they describe things that have to be done (but not how to do them), they are different in that activities are conceptual in nature while tasks are linked to a project management mindset — being the smallest unit of work that results in a deliverable and can be managed.

The relationship between activities and tasks is many-to-many, and the OPF construction guidelines provide a matrix template to assist the process engineer in making these connections (see Figure 5). A similar matrix template is also applied at the next stage, which is the determination of which object-oriented and/or component-based development techniques might be most appropriate to select from OPEN's toolbox [8] in order to provide the necessary support for each task.

To make this use of the two matrices more real, we will now consider a couple of examples — a rigorous/heavyweight process instance and an agile process instance.

"Heavyweight" Process Example

First let us consider the construction of a more rigorous or heavyweight process. The types of activities you might select are shown in Figure 6. Here project management is identified as a specific activity, whereas in other, small process instances, it would be integrated across a number of activities. Roughly speaking, if there is a team responsible for something, it is likely to require an identified activity in OPEN. Similarly, testing is identified as an activity rather than being an integral part of one or more other activities.


The heart of OPEN: Activities and Tasks

Tasks say what is to be done

Activities

Tasks	Activity A	Activity B	Activity C	Activity D	Activity E
Task 1	M	D	F	F	F
Task 2	D	D	F	F	D
Task 3	D	D	O	O	D
Task 4	F	O	O	O	F
Task 5	F	M	O	D	F
Task 6	R	R	M	R	O
Task 7	D	R	F	M	O
Task 8	D	F	M	D	D
Task 9	R	R	D	R	R
Task 10	O	D	O	O	R
Task 11	F	M	O	F	D

For each activity/task combination, choose from one of five levels of possibility from Always to Never

5 levels of possibility

M = mandatory
R = recommended
O = optional
D = discouraged
F = forbidden

Figure 5 — OPEN matrix linking activities to tasks (after [5]).

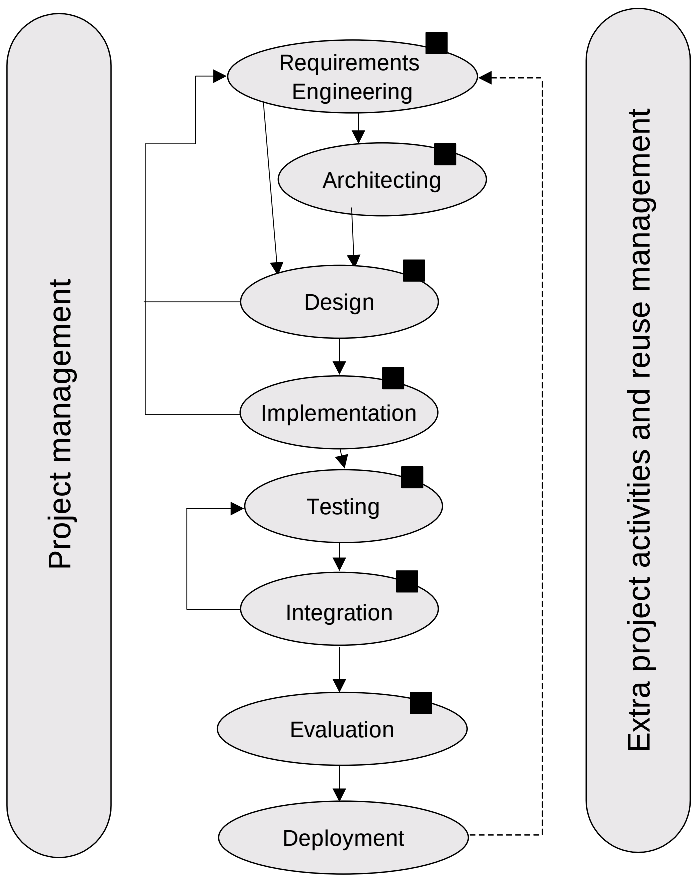


Figure 6 — An example of an OPEN process instance for a larger-sized project with explicit project management and testing (after [3]).

The next step is to identify appropriate tasks for each of these activities. This is done by completing the matrix template of Figure 5. While the activities are written across the top, the possible tasks are written down the left-hand side (see Figure 7). Then one of the five values of possibility is assigned (although I only use the two extreme values in Figure 7). If an inappropriate task is inadvertently selected, then the line corresponding to this task will comprise all "Fs" (see Figure 5) and can then be eliminated.

Finally, you must identify techniques that facilitate accomplishment of the tasks. As with the selection of tasks, you use a matrix to select techniques (see Figure 8 on page 32). Which techniques you select can depend not only on suitability for the job at hand (the accomplishment of the task) but also the preferences and skills of the various development teams and team members. Another consideration is the degree to which your organization wishes to standardize on specific techniques or considers the technique used to be immaterial (the only criterion being whether a technique is effective and efficient in gaining closure on the task for which it is being used).

Although I have described the process construction in essentially a top-down fashion, there is no rule to say that this is the only way. Often it can be better to start with a focus on what deliverables are needed, then ask what tasks and techniques are going to be useful in facilitating their creation, and only then group tasks into appropriate higher-level abstractions of the activity.

"Lightweight" Process Example

In the lightweight example, we identify only four major activities: coding, designing, review, and testing (see Figure 9 on page 33). The associated tasks and techniques make up a relatively small list and are more informally determined than in the formal matrix template approach above. They include, for example, those recommended in Extreme Programming [1] such as pair programming, the planning game, system metaphors, and refactoring — all elements that have been "borrowed" for inclusion in the more recent versions

(text continued on page 32)

Although I have described the process construction in essentially a top-down fashion, there is no rule to say that this is the only way.

©2002 by Brian Henderson-Sellers. All rights reserved.

Task	Activity									
	1	2	3	4	5	6	7	8	9	10
Undertake feasibility study									Y	
Undertake project planning									Y	
Manage human resources									Y	
Identify project roles and responsibilities									Y	
Analyze user requirements	Y									
Maintain trace between requirements	Y		Y			Y	Y		Y	Y
Create a software architecture		Y								
Prototype the architecture		Y								
Develop capacity plan		Y							Y	
Construct the object model			Y							
Identify classes, objects, roles, types, interfaces			Y							
Design the database model			Y							
Design the human interface			Y							
Refactor			Y							
Integrate components						Y				
Code				Y						
Plan integration						Y			Y	
Plan testing strategy					Y				Y	
Design test suite					Y					
Code test suite					Y					
Execute tests					Y					
Report on test results					Y					
Deliver product to users								Y		
Train users								Y		
Identify appropriate reusable work products										Y
Develop reusable work products										Y
Manage library of reusable components									Y	Y
Undertake program planning										Y
Identify program funding										Y
Evaluate quality							Y			
Undertake postimplementation review (a retrospective)							Y			
Evaluate the design			Y				Y			
Evaluate the potential components							Y			

KEY:

1. Requirements engineering
2. Architecting
3. Design
4. Implementation
5. Testing
6. Integration
7. Evaluation
8. Deployment
9. Project management
10. Extra-project activities and reuse management

Figure 7 — Task/activity matrix values for the "rigorous" OPEN instance (only a subset of exemplary tasks are shown).

Technique	Task								
	1	2	3	4	5	6	7	8	9
Cost estimation	Y								
Critical success factors	Y								
Beta testing									Y
Regression testing									Y
Defect detection									Y
Abstract class identification		Y							
Contract specification		Y							
Generalization and inheritance identification		Y							
Qualification, examination, specification, transformation, aggregation (QESTA)					Y				
Granularity		Y							
Rule modeling		Y							
CRC card modeling		Y							
Pattern recognition		Y							
Dialogue design in UI			Y						
Usability testing			Y						Y
Class internal design				Y					
Implementation of services				Y					
Implementation of structure				Y					
Checklists					Y				
Computer-based training						Y			
Lectures						Y			
Roleplay						Y			
Workshops						Y			
Genericity specification							Y		
Revision of inheritance hierarchies							Y		
Class/type indexing								Y	
Library class incorporation								Y	

KEY:

1. Undertake feasibility study
2. Construct the object model
3. Design the human interface
4. Code
5. Evaluate the potential components
6. Train users
7. Develop reusable work products
8. Manage library of reusable components
9. Evaluate quality

Figure 8 — Part of the completed task/technique matrix for the "rigorous" OPEN instance. Only nine illustrative tasks (from Figure 7) are shown. (The full matrix would comprise several pages for such a heavyweight process.)

of the OPF [3, 6, 8]. In this newer, agile environment, you may also wish to augment these OPF process components with new ones of your own.

CONCLUSIONS

The OPEN Process Framework has been shown to offer extensive flexibility in support of a wide range of process types — from small agile processes to larger, so-called heavyweight processes. This is accomplished by using a process metamodel from which to generate process components and then constructing the process itself from these components. Thus, rather than seeing agile and

©2002 by Brian Henderson-Sellers. All rights reserved.

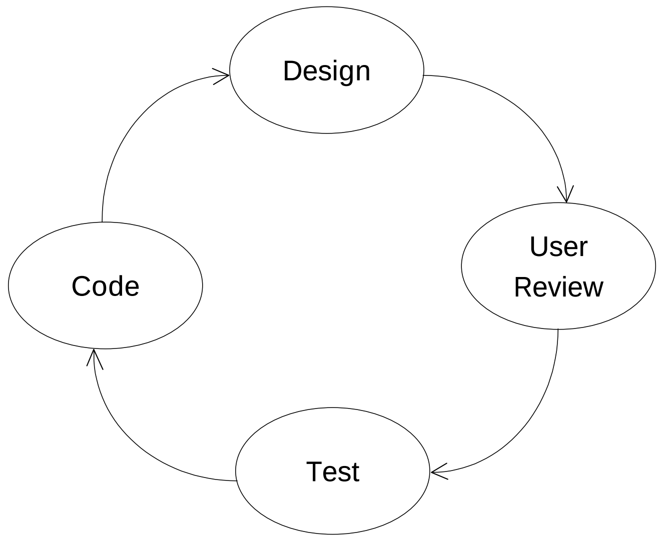


Figure 9 — An example of an OPEN process instance for a lightweight or agile process

heavyweight processes as two ends of a discontinuous spectrum, the OPF supplies an underpinning framework in which both can not only coexist, but effectively transmute, the one into the other, as the demands of a project or an organization change with time. Thus software process improvement becomes possible within the one process framework of the OPF.

REFERENCES

1. Beck, K. *Extreme Programming Explained*. Addison-Wesley, 2000.

2. Chroust, G. "Software Process Models: Structure and Challenges." In *Proceedings of Conference on Software: Theory and Practice*, edited by Y. Feng, D. Notkin, and M.C. Gaudel. NCP, 2000.

3. Firesmith, D.G., and B. Henderson-Sellers. *The OPEN Process Framework: An Introduction*. Addison-Wesley, 2002.

4. Graham, I. "A Non-Procedural Process Model for Object-Oriented Software Development." *Report on Object Analysis and Design*, Vol. 1, No. 5 (1995), pp. 10-11.

5. Henderson-Sellers, B. "Activities, Tasks, and Techniques in OPEN." *COBOL Report* (May 2001).

6. Henderson-Sellers, B. "Enhancing the OPF Repository." *Journal of Object-Oriented Programming (JOOP)* (October 2001), pp. 10-12, 22.

7. Henderson-Sellers, B. "Process Flexibility and Process Construction." *COBOL Report* (December 2001).

8. Henderson-Sellers, B., A.J.H. Simons, and H. Younessi. *The OPEN Toolbox of Techniques.* Addison-Wesley, 1998.

9. Henderson-Sellers, B., B. Haire, and D. Lowe. "Adding Web Support to OPEN." *Journal of Object-Oriented Programming (JOOP)* (June 2001), pp. 34-38.

10. Ralyté, J., and C. Rolland. "An Assembly Process Model for Method Engineering." In *Advanced Information Systems Engineering (CaiSE 2001)*, Lecture Notes in Computer Science 2068. Edited by K.R. Dittrich, A. Geppert, and M.C. Norrie. Springer-Verlag, 2001.

11. Rupprecht, C., M. Fünffinger, H. Knublauch, and T. Rose. "Capture and Dissemination of Experience About the Construction of Engineering Processes." In *Advanced Information Systems Engineering (CaiSE 2000)*, Lecture Notes in Computer Science 1789. Edited by B. Wengler and L. Bergman. Springer-Verlag, 2000.

Brian Henderson-Sellers was the first director of the Centre for Object Technology, Applications, and Research (COTAR; 1994-6, 1999-present) and is professor of information systems at the University of Technology, Sydney. His interests are mainly in OO methodologies and process, metrics, project management, and company migration to OO. Dr. Henderson-Sellers is involved in several metrics projects, leads the OPEN Consortium, and is involved, through the Object Management Group, with the ongoing changes to UML and the new initiative toward a Software Process Engineering Model (SPEM) standard. He has published a significant number of papers and books under the auspices of COTAR, as well as making a large number of international presentations. He is a columnist for the Journal of Object-Oriented Programming (JOOP) *and a frequent speaker at industry conferences.*

Dr. Henderson-Sellers can be reached at The University of Technology, Sydney, P.O. Box 123, Broadway, NSW 2007, Australia. Tel: +61 2 9514 1687; E-mail: brian@it.uts.edu.au.

Big and Agile?

by Matt Simons

If you pay heed to most of the articles and books that have been written on agile development, you probably wouldn't even consider attempting it on a mid- to large-sized project. All the experts seem to qualify their recommendations with something to the effect of, "This will only work for teams of 10-12 developers." Maybe they don't have to.

For the past 18 months, we at ThoughtWorks have been selectively implementing agile methods (inspired mostly by Extreme Programming) on a project to build and deliver the next-generation back-end leasing system to the marketplace. The project team is reasonably big (40 developers, 10 analysts, 15 QA, 10 customers, a few PMs) and the software is complex (750K+ lines of Java code, lots of complicated business rules). Most importantly, the application is in production, and it probably wouldn't be if we had used a different approach.

MODULES VERSUS THREADS

One of the imperatives of agile development is to get software deployed into production as quickly as possible. If you are building a large system, it is likely that you are replacing a legacy application. Even if you aren't, you may not be able to put bits and pieces into production as you go. Instead, you will need to wait until you have created something that provides real business value, something that will allow users to migrate completely to the new system for entire tasks or processes. You will be faced with the choice of whether to develop modules or threads.

Modules are the functional areas of a large application. Billing, accounting, and order entry are some typical modules. Modules are usually reasonably self-contained, with interfaces to other modules in the system. If you are able to identify a module of your system that fills a critical need by itself or doesn't rely heavily on other modules, it may make sense to choose this module as your first "go-live" target. Implementing the module will allow your team to learn how to work together. Going into production that first time will teach you all sorts of lessons about what you might do better when you take on the next one. The earlier you can learn these lessons, the better! Due to their smaller size and more self-contained nature, choosing modules as your initial target may allow you to get something live sooner.

One of the imperatives of agile development is to get software deployed into production as quickly as possible.

Threads are end-to-end processes that touch most or all modules of the system. An example of a thread through a back-end leasing system is: lease entry and booking — billing and invoicing during life of lease — end of lease options — lease termination. A thread is limited in some way: for example, by lease type or by customer. One of the advantages of choosing a thread as your initial go-live target is that in order to realize it, you must develop just enough of most pieces of the system and make them work together. In doing so, you minimize the risk of developing a component that is incompatible with another part of the system, and you remove the need for a separate "integration" phase down the road. Another benefit is that your initial delivery is likely to have much greater business value, since threads often map to the

©2002 Cutter Information Corp.

operations of entire divisions or departments in a business. From the development team's perspective, it is often easier for the members of a large team to work on a thread, since their efforts will be spread out over a wider code base and they will be less likely to get in one another's way. Finally, after implementing a thread, you will be much more capable of estimating the size of remaining development work since everything else in the system should be more or less a variation on a theme.

A team of 20 or more developers working on an agile project can generate incredible momentum and produce code at astonishing velocity.

Overall, building modules first is probably faster initially, but threads are probably better in the long run. Except in special circumstances (such as when you need to get something into production fast or lose your funding), it is wiser to choose threads as your initial go-live target.

ROLES

A team of 20 or more developers working on an agile project can generate incredible momentum and produce code at astonishing velocity. It is important to structure the rest of your teams so that they will be able to feed your development machine the inputs (story cards and test scripts) and have the bandwidth to test and deliver the working software you are constantly creating. In addition to developers, you will probably need the following roles:

Domain Experts

You will need dedicated, full-time, on-site people to play the role of customer. You will need more than one in order to answer the constant questions from the system analysts and to work with the planners to prioritize the flood of new story cards that are constantly being written by project participants. On our team of 40 developers, we had a dedicated team of five domain experts.

System Analysts

Direct interface between domain experts and developers may not be possible (or effective) on a large project. Instead, build a team of "software development" experts who communicate and work well with business experts as well as developers. This team is responsible for writing functional tests for each story card that is "played" based on the understanding of the business need that they have gained through interactions with domain experts. On our team, we had about 10 system analysts.

Release Plan Manager

The care and feeding of a release plan that contains 500+ story cards is at least a full-time job. The release plan manager is responsible for facilitating card-writing sessions and for constantly working with the customer to

The care and feeding of a release plan that contains 500+ story cards is at least a full-time job.

organize the list of story cards into prioritized iterations. This person is also perfectly positioned to facilitate scope review sessions and to ensure that the project is meeting the critical business objectives at every point.

Iteration Tracker

The release plan manager tees up a list of potential story cards before each iteration. During the iteration planning meeting, the team estimates to determine which cards will play in the current iteration. From that point forward, the iteration tracker guides the story cards through actual development. The tracker measures progress on a regular basis, resolves any issues that arise, and maintains communication between the teams working on the new functionality. At the end of the iteration, the iteration tracker feeds incomplete story cards back into the release plan to be dealt with in subsequent iterations.

QA Team

"Keeping everything working" is the main charge of the QA team on a large agile project. Once analysts and developers have completed a story card, they hand it off to QA and move on to the next iteration. QA's job is to continue to run the functional tests (manually at first, automated over time) as the software evolves. It is critical to maintain the ability to deploy after the end of any iteration, and a disciplined regression testing process is the only way to ensure that this remains possible. Our QA team started small (four to five people) and grew to nearly 20 by the time the application was ready to go into production.

BUILDING YOUR TEAM

One of the tricks to pulling off agile development on a big project is to start small and grow. However, you need to be judicious about how you size your teams based on where you are in the project.

The analyst team should start off at nearly full strength so you can quickly attack the problem and generate a full complement of story cards. The development team should start off mid-sized. A core group of around 10 developers can come to a common understanding of the architecture and logistics of the development process. Also, when the system is small, the members of a large development team have a tendency to get in each other's way. The QA team can start small, since there will be a small amount of functionality to regression test at first.

Once the analysts have an understanding of the problem and the developers are successfully producing working software during each iteration, you can begin to scale up your development and QA teams. Pair programming is a great way to introduce new developers to a large project, but be careful about the rate at which you add team members, as adding too many at once can slow down development velocity.

As you approach a major delivery or the end of the project, your QA team should grow proportionally. Depending on your situation, you may be able to transfer analysts or developers to QA as you approach a major milestone.

We have found that not only is it possible, it is critical to invite *the entire team* to the iteration planning meeting.

If you know your project is going to eventually need 50 developers, you may be nervous about starting with such a small number of developers. Don't be. If you lay down solid practices, the velocity you will achieve when your team numbers 50 will more than make up for the time you spent at less than full capacity.

ITERATIONS

Shorter iterations provide the most rapid feedback on your progress, and they allow you to constantly adjust your course. It may be counterintuitive to think that a large team can operate on very short iterations, but with practice it is certainly possible if everyone shares a common understanding of the events of each iteration. If your team balks at starting out with two-week iterations, make your first few a little bit longer — three or even four weeks. Once everyone becomes more comfortable with the flow of the project, it is actually quite painless to reduce your iteration length.

Each iteration begins with an iteration planning meeting. We have found that not only is it possible, it is critical to invite *the entire team* to the iteration planning meeting. If domain experts and system analysts are not present, developers will not be able to ask the questions they need to refine their understanding of a story card and make an accurate estimate. If QA team members are not there, they will be blindsided with new functionality at the end of the iteration, and it will take more time for them to take over the testing of the new functionality.

Running a successful planning meeting for 50 or more attendees requires preparation and careful facilitation. Analysts should meet ahead of time to come to a common understanding on each of the stories scheduled for the coming iteration. They should also prepare test scripts to describe each story card they own. During the meeting, everyone should receive a copy of the list of stories. Analysts briefly describe their card and then a facilitator elicits tasks from the developers. The first few sessions may be a little bit rough, but eventually your team will learn the most efficient and painless way to navigate through the planning meeting. Our first planning meeting lasted two full days, but after a few months, we were able to get through it in roughly half a day.

DELIVERY

Regular delivery of the software under development is critical to validate that the functionality you are building is meeting the business needs. As mentioned above, it is not practical to deliver big systems after each iteration. Instead, you will need to figure out a regular schedule of delivery and an overall release plan. You may wish to plan your deliveries similar to what is shown in Figure 1.

Here you can see that the project is broken into iterations, increments, and releases. An increment is designed to meet a clear business objective. This objective is met by playing the appropriate story cards during the iterations contained within each increment. Proxy users (systems analysts or other groups that represent the user group) test the software after every iteration. Real users test the software after every increment. Planning (through adding, removing, and reprioritizing the story cards in the release plan) happens continuously.

Until a full end-to-end thread exists, the "end-of-the-increment" testing sessions will be limited to unit testing and feature confirmation. However, once a critical mass of functionality has been developed, these sessions should become actual exercises in user acceptance testing. Once you have built a critical mass of business functionality (represented by an asterisk in Figure 1), the business

Regular delivery of the software under development is critical to validate that the functionality you are building is meeting the business needs.

can evaluate the value of the software at any point and make a decision to put the software into production.

WHY BOTHER?

Our experiences using agile methods on large projects have proven that it is possible. This article provides some of the details of how to do it. The only question we haven't answered yet is, "What are the benefits of choosing to

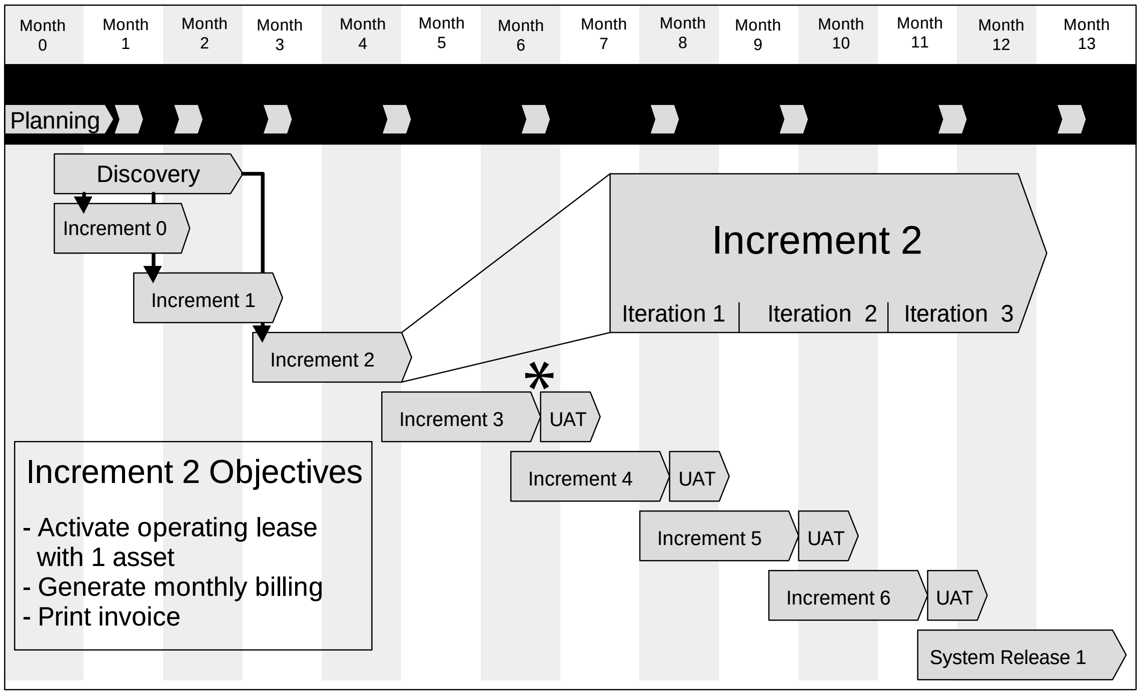


Figure 1 — Iterations, increments, release.

work this way for my large project?"

For purposes of illustration, we will compare the agile approach described herein to a more traditional approach to a large software project (see Figure 2). A traditional software engineering approach proceeds through distinct phases, usually something along the lines of analysis, design, code, test, implement. Agile approaches compress all of these phases into each iteration.

Better Estimates

At the outset of a large project (the planning and discovery phase), neither method is better at making accurate estimates for the time it will take to develop software that meets the business objectives. However, three months into the effort (point A on Figure 2), members of the agile project have been developing software for three months, have completed a number of iterations, and have a very good idea of how fast they can finish story cards. Estimating the remainder of the project is simply a matter of writing the story cards and adding up the estimates. Three months into a more traditional project, you have completed a lot of documents and maybe a few prototypes, but you still don't have any estimates on how long it takes your team to produce working functionality. Agile methods can give you more accurate project estimates sooner.

Practice Makes Perfect

Development on an agile project starts almost immediately. As is shown by the arrows at point B on Figure 2, this means that your development team is writing and delivering software for almost the entire duration of the project. Team members constantly learn and improve and increase the velocity at which they can complete tasks. By the end of the project (when everything becomes most critical), your development machine is well exercised and finely tuned. A seasoned team is much more able to respond to last-minute requests or changes. In contrast, many traditional projects confine development to a short phase of the overall project. Developers may only have several months to practice using their tools, technologies, and processes. At the end of an agile development project, you have more-experienced developers who are

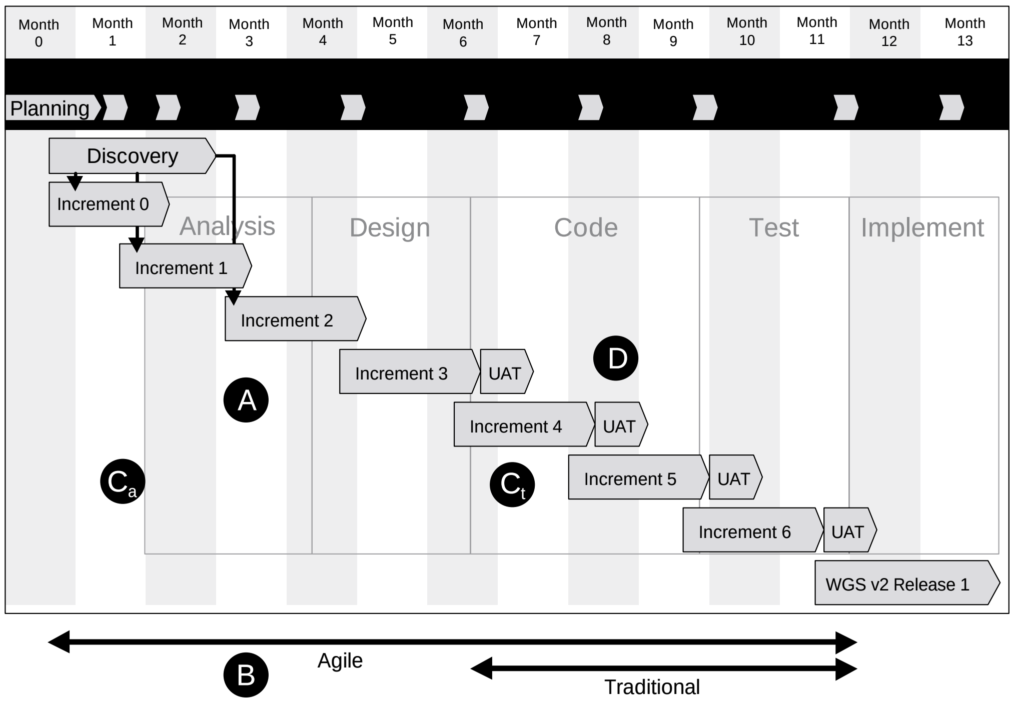


Figure 2 — Agile versus traditional.

more likely to be able to do what it takes to deliver a large system.

Prove Out Architecture/Technologies Earlier

There is significant risk, especially when working with new technologies, that the products you choose will not live up to the marketing hype surrounding them. In addition, it is not uncommon to make mistakes when designing the system architecture before any development has taken place. Changes to either the core technologies or the system architecture can be disastrous in a traditional project where the actual development is compressed into a short time frame. Agile approaches reduce this risk by allowing the development team to prove out the technologies and architecture it has chosen early in the project (C_a instead of C_t, as shown in Figure 2). If changes need to be made, there is much more time to recover, and the impact on the overall project schedule is minimized.

Reduce Risk of False Features and Missed Functionality

Inevitably something will be lost in the translation between the business need and its implementation in software. The only true way to know what you've screwed up is to get the software in front of real users and let them use it. In traditional projects, users may not get a chance to test working software for many months. Instead, they may be asked to review and sign off on requirements documents or analysis artifacts. In a large and complex system, it is very difficult for users to conceptualize the system based on paper documents. But put them in a room and let them take it for a test drive, and you'll get 10 pages of feedback in an hour! Agile methods constantly put software in front of users and constantly adjust based on user feedback. This practice reduces the amount of time you spend building superfluous functionality and ensures that you don't build functionality that doesn't meet the business need.

Business Value Earlier

Finally, and possibly most critically, agile approaches provide business value sooner. Consider a project at point D in Figure 2. At this point in a traditional project, you would have a large amount of documentation and some undetermined amount of untested software. If for some reason you were forced to cancel or reduce the scope of the project at this point (or at any point prior to final delivery), you would likely be left with *absolutely nothing* to show for your time and money. However, if you were developing based on an agile approach, at point D you would have eight months' worth of production-ready software. Furthermore, the functionality that had been developed up to that point would be the functionality that the business chose over all other scheduled functionality. Once you achieve a critical mass that provides actual business value (as discussed above), you remove the risk of stopping the project and being left with nothing at all. From that point forward, the business can make a decision at any time to stop development and realize the value created up to that point.

Changes to either the core technologies or the system architecture can be disastrous in a traditional project where the actual development is compressed into a short time frame.

Big and agile — it may seem like a contradiction in terms, but the teammates and clients I have worked with have become firm believers that not only is an agile approach possible on big projects, it is actually the most effective way to go about delivering large software applications.

Matt Simons is a project manager for ThoughtWorks, Inc. Mr. Simons has experience as both a member and leader of project teams building and implementing mission-critical software applications. He recently finished working on a project that used an agile approach to plan, build, and deliver a J2EE-platform-based back-end leasing system to a large corporate client. The project involved nearly 100 team members (including 40 developers).

Prior to joining ThoughtWorks, Mr. Simons lectured at Chiang Mai University in Thailand, where he published a book on business communication. He is a regular speaker at software development conferences and a contributor to industry journals.

Mr. Simons can be reached at ThoughtWorks, Inc., 651 W. Washington Blvd., Suite 6000, Chicago, IL 60661. E-mail: MTsimons@thoughtworks.com.

Cutter IT Journal

CUTTER IT JOURNAL

Topic Index

Upcoming

Issue Themes

Is Risk Management Going the Way of Disco?

The Technology Myth in Knowledge Management and Business Intelligence

Web Services

Security

Design for Globalization

Open Source

Testing

XP and Organizational Culture Change

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