# Agile Software Development

## Software Development as a Co-operative Game

## 2nd edition

## Alistair Cockburn

# 0'

# *Unknowable and Incommunicable: Evolution*

*The Introduction set out two ideas:*

> *Communicating is touching into shared experience.*
>
> *People listen and act at three skill levels: Shu, Ha, and Ri (following, leaving, and fluent).*

*Those ideas could stand a few more paragraphs each.*

## Communication and Shared Experience

At the time of writing the first edition, I didn't fully appreciate the harmony between the idea of "communication as touching into shared experience" and the articles in Appendix B by Pelle Ehn and Peter Naur.

(New readers may want to skip this section until they finish reading Appendix B. I am writing this section as though you have done so and come back to this point.)

Within the last five years, people first applied the idea of "communication as touching into shared experience" to project documentation. Teams are being more creative in using alternate forms of documentation to serve as markers and reminders for other team members[1]. They are separating what is needed as "reminders for the people who had been in the room at the time" from what is needed as "catch-up information for people who were not in the room at the time."

This is, of course, very much in line with Peter Naur's article "Programming as Theory Building" in Appendix B. Naur describes designing a program as creating a theory of the world (we might say *model* instead of *theory,* today) and another theory of how the program operates in that world.

To remind someone about a conversation they participated, we point into their theory (or model), and point as directly as possible into their shared experience. For this purpose, it is desirable to keep the original, messy, handwritten flipcharts. Rewriting them into clean computer text loses the exact referencing mechanism we want to utilize. It also work out, handily enough, that keeping the original flipcharts is fast and cheap, whereas rewriting them online is relatively time consuming and expensive. So using the original flipcharts as *reminders* is both cheap and effective.

However, it is quite different when the purpose is to bring someone else up to the team's current state of understanding (their *theory*, in Naur's terms). The same messy flipchart that is so effective as a *reminder* is unlikely to contain the cues necessary for the new person[2].Most of the time, the team will choose to use a completely different approach to documenting their current state of understanding to help the new person catch up with them. However they approach it, a great deal of information is inevitably lost, just as Naur describes in his article.

In the second article in the appendix, Pelle Ehn described the difficulty he and his programmers had communicating with their computer-naïve users in the 1970s. Surprising most people (including me), he wrote that complete communication is not necessary:

> ". . . paradoxical as it sounds, users and designers do not have to understand each other fully in playing language-games of design-by-doing together. Participation in a language-game of design and the use of design artifacts can make constructive but different sense to users and designers." (see page xxx???).

His insight is relevant again today, when ethnographers and user experience designers still try to understand the work habits of users, and from that understanding construct a good user interface. At a time when we work so hard to understand our users, it is good to be reminded that complete communication is not possible . . . but it is also not required. What is required is to play the language-game, acting, and reviewing the feedback, over and over in never-ending cycles of improving utility.

You can apply the lessons of "touching into shared experience" in your own environment:

- Watch as people *point* to an old artifact to remind themselves of something that happened.
- Watch as they *construct* something to create a "ladder of shared experience" for a new person.
- Watch how communication gets missed, understanding is incomplete, but action gets taken anyway. Notice how many times the people repeat the cycle of explain-act-discuss before they get to an *acceptable* level of non-understanding.
- Notice how people have different tolerances for levels of non-understanding and for the number of times they have to repeat the cycle. Use what you notice to do two things: Raise your own tolerance for repeating the cycle; and (surprise!) raise your tolerance for people who have low tolerances.

[1] See, in particular, the many examples of project documentation in *Crystal Clear* (Cockburn CC 2005).

[2] Unless, of course, the flipchart is recreated – redrawn, in the same sequence – for the new person as it was for the original audience. Then the new person builds an internal memory of the session rather similar to those who were in the room for the original meeting. I have used this technique on occasion to good effect.

## Shu-Ha-Ri

The Shu-Ha-Ri distinction dates back, as I learned, not to the origins of Aikido the early 1900s, but to Japanese Noh theater, almost four centuries ago.

Understanding the three levels of listening has turned out to be of greater value than I expected. It was originally placed as a small section in an appendix, but so many people wrote to me during the early drafts of the book, describing how they were using the Shu-Ha-Ri distinction to smooth meetings, that I moved it forward into the Introduction.

You can apply Shu-Ha-Ri to designing courses and writing technique materials. Recognize that some people come to the course in the *Shu* state, looking for *the* technique that solves their problems. Other people come already knowing many techniques, afraid that you will try to convince them that your technique is *the* technique that solves their problems.

Organize your material to reassure the advanced people in your course that it is not being presented as a cure-all, but simply another technique in the toolbox of the professional. Then teach the technique at the *Shu* level, so that the beginners will have a starter technique to take home with them. But before you finish, put the technique into a larger context, so the beginners can start to see where they will expand to, and the advanced people can see how this one technique bridges to others they might know or want to learn.

I got caught in a Shu-Ha-Ri mismatch, shortly after publishing the first edition, and was saved by someone who knew the trio.

In 2002, Jim Highsmith and I were organizing the first Agile Development Conference (alert readers will notice here that conference organization is a cooperative game of invention and communication, but one in which the incremental / iterative and collocation development strategies are difficult to use, if not impossible). It was my first-ever conference-organizing experience, and I was very much in the Shu stage, but without anyone to follow. Accordingly, I was incredibly detailed in listing everything that needed to be taken care of for the conference. I had sheets and sheets of paper containing all manner of minutiae.

We signed an experienced conference organizer, who was baffled by my excruciatingly long lists. A colleague watching our tense conversations pointed out that my long lists were immaterial to her, as she operated in Ri mode and could make the necessary plans in her head rather than on slips of paper.

That observation gave me the peace of mind I needed to relax and let her take over. I still kept lists of details to watch for, but only the ones I was specifically concerned about.

You can apply the lessons of Shu-Ha-Ri in your own environment:

- Look for level mismatches at work. Introduce the Shu-Ha-Ri concept and see if that might help to smooth the discussion.
- Update your training materials to highlight what sections are for *Shu* level learning, and where the *Ha* and *Ri* levels fit.

# 1'

# *A Cooperative Game of Invention and Communication: Evolution*

*I once thought that software development was unique in its being a cooperative game of invention and communication. As I gave talks on the subject, however, business executives started telling me that their own professional lives fit that description. Business, it turns out, is very much a cooperative (and competitive!) game of invention and communication. So are law, journalism, conference organization, entertainment, and running governments. More surprising to me, personally, was discovering that engineering is also largely a cooperative game of invention and communication.*

*Researching the topic of engineering, I found that, possibly due to the success of applied physics in World War II, it lost its original association with craft and experiment only quite recently. Reconsidering engineering in its more historical interpretation, as a combination of applied science, craft, experiment, and case lore, gives it a new and richer meaning. This view of engineering shows the way for us to reconstruct "software engineering" in a way that is both provides a good educational base, and also is practical on live projects.*

## The Swamp Game

Rock climbing has one limitation as a comparison partner to software development: When people finish a rock climb, they walk back down to their cars and drive off. When they finish a software project, they stay around and enhance the system or build the neighboring system. They have to live with the results of what they did, and they have to integrate other people into the world they created.

There is an activity that is closer to software development than rock climbing. It is an imaginary activity, for now, but who knows, people may really start playing this game, or simulating it on a SimCity type of computer game to train project leaders.

Imagine a competition in which each team must build something somewhere in a swamp. They don't know exactly what they have to build, they don't know where they have to build it, they don't have a map of the swamp, but they are in a race to build it. They will create subteams. Some will scavenge for building materials, others will explore the swamp, and so on. Note that this corresponds to people finding different specialties in software development.

If they have to play only one round of the game, they will find it optimal to build the weakest, sloppiest bridges and the most careless maps possible to get to the end of the game. However, if they know there will be another team coming in after them, they may choose to make better maps, better bridges, better paths. This difference between strategies is what corresponds to a software project team updating their system documentation and training their junior people.

It is this difference in strategies that makes the swamp game a better comparison partner than rock climbing. Every software project contains a carryover to the next project, and so the strategies to use in software development should pay attention to both *this* game and *the next* game.

Observe that *people* are part of the carryover from one round of the game to the next. In an ideal world, the team would arrange that none of the original team would leave. These people carry a lot of tacit knowledge in their heads, information that can't be put into the paper documentation (this is the exact point made by Peter Naur in Appendix B). However, in a real-world setting, people flow into and out of the team. The team's strategy needs to include ways to transfer as much knowledge as possible to the arriving people.

The swamp game lets us explore in our heads different strategies the teams might use, and map those strategies map to software development.

- We can imagine keeping the original team intact for as long as possible.
- We can imagine junior players on one round becoming team leads on the next round.
- We can imagine using an apprenticeship model to train new people joining the team.
- We can imagine making the paths and bridges weak on the first round, but taking the time to improve them on subsequent rounds (think here of refactoring code in subsequent releases).

The good news is that delivering the result in the current round is a finite task. The bad news is that setting up advantageously for the following round is essentially an infinite task: there is never enough time, money, staff, mental and communication capacity to do it perfectly.

This informs us as to why the documentation and training never seems to be satisfactory at the end of a project. Project managers always feel like they are under-resourced, and they are. They are trying to balance finite resources across a finite task plus an infinite task. They will always come up short.

The trick is not to aim for perfection, but to construct strategies that serve a useful purpose in getting to the next move or round of the game.

## Competition within Cooperation

I am such a big fan of cooperation and collaboration that I sometimes forget that there are other things even more powerful – competition, to name one. The question is how to harness it so that the competition does not destroy the collaboration.

Darin Cummins[3] found a way, as far as I can tell, and although he only used it twice, it points to a set of possibilities other teams might like to experiment with. He used the following "game" twice, which is a good sign, and decided that particular theme had run its course.

[3] Of ADP Lightspeed, Inc. See "The Development Game" (Cummins 2003).

Darin cautions that the competition should not be about the code being written, because then the collusion, cheating, and destructive side of competition can ruin the code base or the team. Instead, he and the other managers created a competitive framework that reinforced development practices that they had selected. Interestingly, as the people on the team colluded to optimize their positions in the game, they actually carried out the practices and collaborated more than they otherwise would have. So "cheating" strengthened both the team and the practices.

Darin had read a Harvard Business Review article, “Everything I Know About Business I Learned from Monopoly” (Orbanes 2002). Orbanes lists six principles that make a great game and describes how those they can be applied to business.

1) Make the rules simple and unambiguous.
2) Don’t frustrate the casual player.
3) Establish a rhythm.
4) Focus on what’s happening OFF the board.
5) Give them chances to come from behind.
6) Provide outlets for latent talents.

Darin applied those principles to improving their team's software development practices.

- He got his company executives to put real money into the pot.
- They bought gadgets that might interest the people on the team.
- They made fake money (with the VP's face on it!) to reward people for specific actions. See Figure 8-1 for a partial list of the rewards.

As Darin writes:

"The developers get paid for having their code reviewed, reviewing someone else’s code, completing tasks on schedule, reusing code, creating unit tests, etc. We also minimized the process as much as possible by getting rid of the huge checklist that was quite difficult to get the team to use and replacing it with a simple, 8 item checklist that is turned in at the end of each cycle. This checklist and a simple code review worksheet are used to help determine how much each “player” gets paid. . . .

The management team also has the ability to reward items not included on the list. For example, when we started having people come in late to our daily standup meeting, one of the managers chose to pay everyone $5 for being on time. The message was received and the problem became minimal. Players also have the ability to pay each other if they choose. . . .

One important principle that the game article discussed is giving the players the chance to come from behind in the endgame. This prevents a player from falling behind to the point where playing the game is hopeless and they lose interest. As we approached the end of the game, we devised some extra activities that could earn a developer more money."

What I found interesting was that the (inevitable) collusion improved the outcome for the team as a whole. People figured out that they would get points faster by trading code to review, and more code got reviewed!

At the end of the three-month game period, they held an auction in which the fake money was used to bid for the real items purchased. To make it more fun, they brought in a fast-talking salesman as auctioneer.

"The results from implementing the game have been far better than expected. The one result that we had hoped for in creating the game was to increase morale. While it wasn’t intolerable, there were some bad attitudes and bickering beginning to surface. Left unchecked, I feared that these attitudes would escalate until the project and/or the team suffered. The game helped attitudes rise back to where they needed to be for us to be able to complete the project by giving an outlet and providing focus.

The first unexpected thing we noticed was that the process now had a purpose. Everyone who has ever written code knows that software developers just want to be left alone to code. Logically, they know that the process is necessary, but emotionally the attitude is simply “If you want it done, go away and leave me alone.” The game had the effect of not only giving them an emotional reason to accept the process, but to give it some excitement too."

The game worked for them, and they ran it a second time before deciding it would get old after too many uses.

After reading the article, I'm keeping my eyes open for other ways to use competition to improve (not damage) team quality and team output.

| **Description** | **Reward** |
|---|---|
| Creating Unit Tests / Test Plans | $20 |
| Review code for someone else | $30 |
| Receive an “Excellent” average rating in code review | $50 |
| Receive a “Acceptable” average rating in code review | $20 |
| Receive a “Unacceptable” average rating in code review | -$20 |
| Don’t have code reviewed | -$50 |
| Implementing suggestions from Code Review | $30 |
| Warnings in build | -$20 |
| Errors causing build failure | -$50 |
| Errors in test after build | -$30 |

**Figure 8-1.** Some of the rewards for the Development Game (Cummins 2003).

## Other Fields as a Cooperative Games

When I wrote the first edition of this book, I promise I was thinking only about software development. However, as I gave talks on the subject afterwards, I kept hearing business executives, actors, lawyers, journalists, and others hearing their professions in the talk. It seems that quite a lot of fields involve people inventing and cooperating, and the outcomes in their fields are highly sensitive to the quality of their invention, communication and collaboration.

Most recently I was introduce to a company in Salt Lake City, O.C. Tanner, using the lean manufacturing approach in their manufacturing line for corporate rewards (plaques, pins, medals and the like). I had thought that surely line-based manufacturing was quite far from the agile and cooperative game ideas, but it turns out that even (well-run) manufacturing lines are sensitive to the quality of ongoing invention, communication and collaboration of their workers.

In the upcoming chapter on agile techniques (Chapter 5), I will revisit the story of O.C. Tanner and Tomax, among other companies playing the cooperative game deliberately. Those of us using this vocabulary share the feeling is that despite all the books written in the business world, there is still much to be learned from the cooperative game approach.

## Software Engineering Reconstructed

I have twice reported on the mismatch between our use of the term *software engineering* and the practice of engineering[4]. As part of, and following those reports, I have been researching the question of what "engineering" consists of in practice, where we went wrong with our interpretation of the phrase *software engineering*, and how we can put software engineering back on solid footings. This section contains what I have so far.

### Where the term came from

The term "software engineering" was coined in 1968, in the NATO Conference on Software Engineering. It is fascinating to read the verbatim reports of that conference, and compare what those attendees were saying with what we have created since then.

The first thing to note is that the conference organizers did not *deduce* that software development was like engineering. Rather, they were deliberately being provocative in coining the term "software engineering." We read, in the introductory note, "BACKGROUND OF CONFERENCE" (p. 8):

> In the Autumn of 1967 the Science Committee established a Study Group on Computer Science. The Study Group was given the task of assessing the entire field of computer science, and in particular, elaborating the suggestions of the Science Committee.
>
> The Study Group concentrated on possible actions which would merit an international, rather than a national effort. In particular it focussed its attentions on the problems of software. In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

### Where we went wrong

The second interesting thing to note in those proceedings is the mismatch between what they claimed engineering consists of, and how they described their own methods and recommendations for developing software. We read from their motivational comments, such as this one (p. 12):

> We build systems like the Wright brothers built airplanes — build the whole thing, push it off the cliff, let it crash, and start over again.

Actually, the Wright brothers practiced engineering in its best possible sense: They used theory, experiment, guesses and feedback. The Wrights account (Wright 1953, pp.15-18) reads as follows:

> In order to satisfy our minds as to whether the failure of the 1900 machine to lift according to our calculations was due to the shape of the wings or to an error in the Lilienthal tables, we undertook a number of experiments to determine the comparative lifting qualities of planes as compared with curved surfaces and the relative value of curved surfaces having different depths of curvature. . . . In September we set up a small wind tunnel in which we made a number of measurements. . . but still they were not entirely satis-

[4] The first edition of this book and "The End of Software Engineering and the Start of Cooperative Gaming" (Cockburn 2003).

> factory. We immediately set about designing and constructing another apparatus from which we hoped to secure much more accurate measurements. . . . we made thousands of measurements of the lift, and the ratio of the lift to the drift with these two instruments.

But engineering is more than just calculating tables. It consists of interacting with a material (air, in the Wright brothers' case). The activity, "doing engineering" is learning the qualities of the material and devising (inventing) ways to accomplish what one wants with the material. There are almost always conflicts, so that a perfect solution is not available. Finding a solution that works adequately in the particular situation requires reframing the problem over and over as more is learned about the material, the problem, and the set of possible solutions.

Donald Schön and Chris Argyris, professors at M.I.T. refer to this constant reframing as having a "reflective conversation with the situation." In *The Reflective Practitioner* (Schön 1983), they catalog, among others, chemical engineers tackling a new problem. These chemical engineering graduate students asked to create a manufacturing process to replicate a particular patina on china that came from the use of cow bone, which was soon going to become unavailable. The students progressed through three stages in their work:

- They tried using chemical theory to work out the reactions that were happening with the bone, so they could replicate those. They got completely bogged down using this approach.
- Abandoning theory as too difficult, they next experiment with processes in whatever form they could think up. This also got them nowhere.
- Finally, their professor coached them not to replicate the bone process, but to devise a process that resulted in a similar effect. They combined bits of theory with experiments to make incremental progress, getting one category of improvement at a time and reducing the search space for future work.

Their third stage shows their reflective conversation with the situation. As they learned more about the problem and the solution, they were able to make better moves in their game.

Schön and Argyris's account meshes perfectly with Pelle Ehn's description of software design as a language game (see page [Ehn, p. 230] of this book):

> "In the conversation with the materials of the situation, the designer can never make a move that has only intended *implications*. The design material is continually talking back to him. This causes him to apprehend unanticipated problems and potentials, which become the basis for further moves."

Engineering, then proceeds as a cooperative game of invention and communication, in which the moves come from theory or inspiration, and are checked with experiments that reveal more about the situation.

How, then, did engineering get so misunderstood? Schön and Argyris relate:

> Harvey Brooks, the dean of the Harvard engineering program was among the first to point out the weakness of an image of engineering based exclusively on engineering science. In his 1967 article, "Dilemmas of Engineering Education," he described the predicament of the practicing engineer who is expected to bridge the gap between a rapidly changing body of knowledge and the rapidly changing expectations of society. The resulting demand for adaptability, Brooks thought, required an art of engineering. The scientizing of the engineering schools had been intended to move engineering from art to science.
>
> Aided by the enormous public support for science in the period 1953-1967, the engineering schools had placed their bets on an engineering science oriented to "the possibility of the new" rather than to the "design capability" of making something useful . . . Practicing engineers are no longer powerful role models when the professors of highest status are engineering scientists. . . . by 1967 engineering design had virtually disappeared from the curriculum, and the question of the relationship between science and art was no longer alive. . . . (Schön 1983, pp. 171-172)

People exaggerate the reliability of mathematical prediction in expecting engineering projects to complete on cost, time and quality. Civil engineers fail in the same way as the average software developer when put in a similar situation. As just one example, the project to build a highway under the city of Boston was estimated in 1983 to cost $2.2 Billion and be completed in 1995. It was still incomplete in 2003, with a cost estimate of $14.6 Billion (Cerasoli 2001, Chase URL). The 600% cost overrun was attributed to the project being larger than previous projects of this sort, and its using new and untried technologies.

After studying this and other civil engineering projects, Martin Fowler quipped in a public talk, "Compared to civil engineers, software developers are rank amateurs at cost overruns".

Popular expectations about engineering are faulty because popular understanding of engineering as an activity is faulty. Once we understand engineering as an economic-cooperative game, the difficulty in accurately predicting the trajectory of an engineering project is one of the things that becomes understandable.

## Reconstructing Software Engineering

If we reconsider engineering as a cooperative game of invention and communication and as reflective conversation with a situation, can we create a better understanding of the term *software engineering*? I believe we can, and that we can fit software development in the proper engineering tradition by building both on the foundations of craft, the cooperative game of invention and communication, and the lessons we have learned from lean manufacturing.

The craft model captures individual behavior. The cooperative game model captures team behavior. Lean manufacturing theory captures the economics of structures and strategies.

The key in transferring lessons from manufacturing to engineering is to recognize that the "unvalidated decision" is the unit of inventory in engineering (or cooperative games of invention and communication, in general).

Here are the components of the restructuring.

## Craft

In *The Reflective Practioner,* Argyris and Schön follow the working habits of architects and engineers. In their case , we see people making predictions as to what their situation will support, then they do som of their work and watch what the situation actually produces, and from that, they alter their understanding. They update their understanding of the problem and the proposed solution, both, according to the changed situation and their new learning. This "reflective conversation with the situation," is just what the Wright brothers did in designing their airfoil.

*Craft* implies skill in working in a medium, mental or physical. There are many materials in an engineering project, and therefore many skills or crafts to become adept at. Some people need skills around managing people; others need mathematical skills; others need visual design skills, and so on.

Each skill and material brings its own specialized vocabulary: Ceramicists "wedge" clay and study the shrink rate of different glazes compared to clays. Civil engineers work with the tensile strength, fatigue rates and thermal expansion of their materials. Writers look for flow and development in their texts. Programmers look at cohesion and coupling, testability, clarity of expression, and computational complexity in their algorithms. Testers work with test coverage and probabilities of occurrences. User interface (UI) designers work with cognitive-motor task switching times, recognition times, color scales, and user work patterns. Project managers work with people, and are concerned with what builds or tears down trust, community and initiative.

In each case, craft practice requires practitioners to have a reflective conversation with the material, using the specialized vocabulary. They work with the constraints offered by the materials, the vocabulary, and the project.

On a software project, the UI designer is constrained by the two-dimensional screen, the operating system's rules, and the UI component set that has been selected for use on the project. The UI designer will learn what is possible and not possible in the framework (and possibly also what is not supposed to be possible, but can be done anyway). As work progresses, using the evolved understanding of what can and cannot be done, the UI designer may counterpropose a different work-flow to the users from what they originally requested.

The materials for a system architect are architectural standards, application program interfaces (APIs), and signal propagation technologies. The architect balances performance degradation with ease of change as they add APIs and separate the system across different boxes or locations. This work is partly a mathematical activity and partly heuristic.

The materials for a programmer are the programming language, the operating system, and the components or APIs the programmer is obliged to use. Some programming languages are very malleable, LISP, APL, Smalltalk, Python, Ruby, being examples. Other languages are less malleable, Ada, C++, and the other strongly typed languages being examples. Programmers sometimes compare working in the former languages to working with clay, and working in the latter languages to working with marble. Wonderful sculptures can be made from either clay or marble, but the skills and the approach differ significantly. So, too, for the different programming languages.

In software engineering, the crafts include at least

- project management,
- "requirements" elicitation (in quotes because the development team can usually counterpropose alternatives, which meant they weren't really requirements in the first place),
- user interface design,
- system architecture,
- program design and programming,
- database design,
- algorithm design,
- testing, and
- communicating what they are thinking.

Developing professionals in our field means recognizing the multiple crafts needed to carry out a project. It means training newcomers in the basics of those crafts. It requires professionals in the field to continue developing craft skills development as a natural part of being in a craft profession. It means honoring the different crafts required to complete a project.

The crafts define the specific moves that the people make in the cooperative game.

## Cooperative Games

Engineering is seldom done alone. Engineers exchange their thoughts and proposals as they proceed along their intertwined paths. They communicate with sponsors and the users: The sponsors must find ways to fund the work. The users must respond to the unexpected ways in which the system will change their lives.

Below is a description of engineering work at Lockheed's Skunkworks facility. In this extract, it is easy to see the principles of the cooperative game in action and the cross-coupling of the crafts. This is an outstanding example of a group holding a reflective conversation with the situation as a team:

SKUNK WORKS ROOMS

"Kelly kept those of us working on his airplane jammed together in one corner of our [building]... My three-man thermodynamics and propulsion group now shared space with the performance and stability-control people. Through a connecting door was the eight-man structures group. ... Henry and I could have reached through the doorway and shaken hands.

"... I was separated by a connecting doorway from the office of four structures guys, who configured the strength, loads, and weight of the airplane from preliminary design sketches. ... [t]he aerodynamics group in my office began talking through the open door to the structures bunch about calculations on the center of pressures on the fuselage, when suddenly I got the idea of unhinging the door between us, laying the door between a couple of desks, tacking onto it a long sheet of paper, and having all of us join in designing the optimum final design. ... It took us a day and a half. ..."

"All that mattered to him was our proximity to the production floor: A stone's throw was too far away; he wanted us only steps away from the shop workers, to make quick structural or parts changes or answer any of their questions." (Rich 1994)

## Lean Manufacturing

Lean manufacturing may seem a peculiar foundation for software engineering. Surely, if there is one difference between manufacturing and engineering, it is surely that manufacturing is about making the same item many times, and engineering is about approaching situations that are unique each time[5].

[5] However, see Mike Collins' description of lean manufacturing in a mass customization environment, on page ???.

Replace manufacturing's flow of materials with engineering's flow of decisions, however, and the rules governing the two are remarkably similar

One remaining difference is that the flow of decisions in an engineering project has loops (requests for validation and correction), which designers of manufacturing lines try hard to avoid. These loops complicate the optimal strategies for the team but do not change the rules of the situation.

Figures 8-2 and 8-3 show the flow of decisions in a software development project. In Figure 8-2, I erased the various feedback loops in order to highlight the basic idea of decision flow; Figure 8-3 includes them show the actual situation. The loops are critical – without them, the team is missing badly needed feedback about their work. So whatever you do, please don't copy Figure 8-2 as a rendering of how *good* software engineering is done! If anything, show it as a rendering of how *poor* software engineering is done.

Although Figure 8-2 is wrong in as guide to how to set up your development shop, it is still useful as a starting point for this discussion, because it is simple enough to inspect at a glance. Let us look at it:

- The users and sponsors make decisions about what they want. They feed those decisions to UI designers and business analysts (BAs).
- The UI designers and BAs, collaboratively with the users, or on their own, make detailed decisions about the appearances, behaviors and data to be put into the system. They feed those decisions to the programmers.
- The programmers make a decisions about the program. Those decisions probably change some of the users' requests and the decisions made by the UI designers and BAs, but let's skip that topic for the moment. However they make their decisions, those decisions go to the testers.
- The testers make decisions about where the foregoing people have made mistakes. (The dashed arrow in Figure 8-2 represents another oversimplification picture, that I do not say whence the testers get their idea of what is "correct". For this figure, it just comes from somewhere ahead of them in the chain of decision making.)

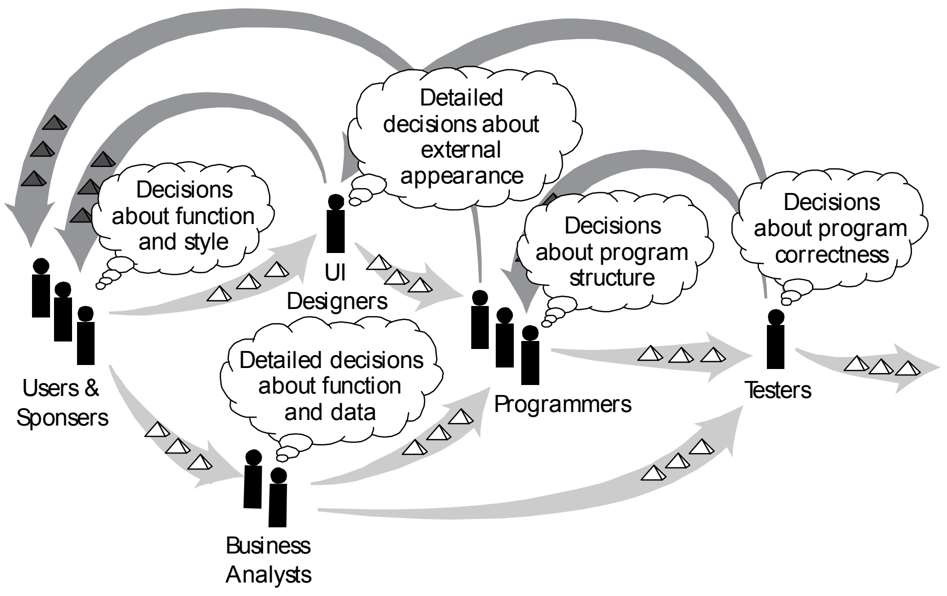
The testers can't tell whether the program is correct until they get running code; the programmers can't know what to code until the UI designers and BAs make their detailed choices; and the UI designers and BAs can't decide what to request until the sponsors and users make up their minds.

As with manufacturing, there is "inventory," there are "work-in-progress" queues between work stations, and there is the problem of selecting batch sizes for those handoffs.

The engineering team's inventory is made up of un-validated decisions. The more decisions saved up and passed from one person to another at one time, the larger the batch size involved in that handoff.

Texts on lean manufacturing (for example, Reinertsen 1997) stress that increasing the batch size generates almost uniformly negative consequences on the system's overall behavior. Small, natural variations in work speed produce unpredictable delays at the handoffs. In the end, it is hard to predict the flow through the system, even for a well-designed system. The cure is to reduce batch size at the handoff points, then choose from a collection of techniques discussed in the lean manufacturing literature to deal with the inevitable bubbles of inventory that pop up from place to place within the system[6].

[6] The simplest is to cross-train people to be able to do the work of those adjacent to them in the flow. When inventory starts to build up at a handoff point, the upstream worker joins the downstream worker in clearing it.


Detailed decisions about external appearance
Decisions about function and style
Decisions about program structure
Decisions about program correctness
UI Designers
Users & Sponsers
Detailed decisions about function and data
Programmers
Testers
Business Analysts

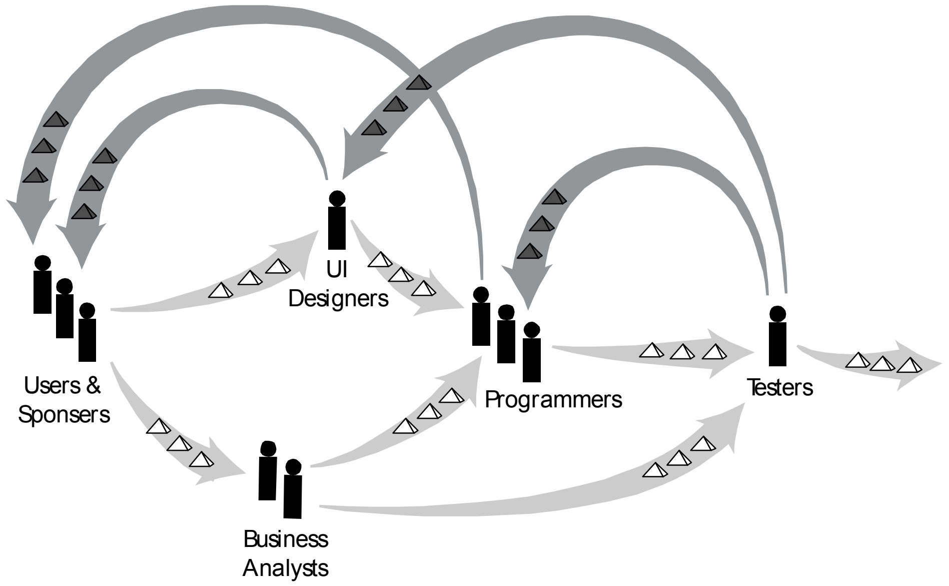


**Figure 8-2/3.** (new figure) replacing both 8–2 and 8-3. Decision flow with feedback

In software development, reducing batch sizes corresponds to designing, coding, testing and integrating smaller amounts of functionality at one time. It is hard to imagine what just "one" piece of functionality is, but it is easy to see that on most software projects, a large number of unvalidated decisions are batched up in the requirements document, a large number of unvalidated decisions are batched up in the UI design, and a large number of unvalidated decisions are batched up in the assembled code base before the testers are given anything to test. Reducing the functionality requested in one delivery increment is the first step in reducing batch sizes.

As with manufacturing, there is a cost associated with reducing batch sizes. In manufacturing it is the cost in switching from one job to another. Lean manufacturing companies spend a lot of money reducing job switching times so that they can gain the benefits of smaller batch sizes. (Important note: they don't make excuses that they *can't* have small batch sizes *because* they have high job switching costs; they redesign their job switching procedures to lower that cost in order that they can use smaller batch sizes. This difference is significant.)

Engineering groups have not only the task-switching costs, but also the cost of injecting new work into the system through the feedback loops and the cost of revising the designs as the requests change. Unresolved arguments still rage in the software development community about how small the batch sizes can be made in UI and database design.

Figure 8-2 shows one other important characteristic of a development team: There are a different number of specialists at each point in the flow, so they have different capacities for clearing their decisions. Alert managers do two things when studying these decision-flow capacities:

- They work to balance the relative capacities at the respective stations. It does little good to have lots of BAs and not enough programmers, or lots of programmers and not enough testers.
- They decide what to do with the specialty groups where there is excess capacity. If they can't simply send the extra people home, they have to decide how to use those people. This is where Principle #7 from Chapter 4 comes in ("Efficiency is expendable at non-bottleneck stations"). That principle points to a wide set of management strategies that can be used in different circumstances (Cockburn 2005 ICAM).

Now to deal with Figure 8-2's oversimplifications. First, the dashed-line flow of information into the testers. The testers will get their information either from the users, or the BAs, depending on the organization. To repair this aspect of Figure 8-2, simply connect the dashed line to the appropriate source.

Next, consider the loops. When the testers find a mistake, they send a report back to the programmer or the UI team. These defect reports constitute additional inventory for the programmer or UI team to handle. A loop exists from the UI designers to the sponsors and users. The UI designers make a mock-up of the user interface, show it to the users, and come away with a stack of rework decisions. A third loop may exist from the programmers to the users and sponsors, creating a more decisions being fed to the UI designers, BAs, and programmers.

Figure 8-3 shows this more realistic, picture of the decision flow in software development. Note that although the flow is more complicated than Figure 8-2, the same theory applies. The same results apply about batch sizes at the handoff points and also about the use of creative strategies at non-bottleneck stations.

All the above dependencies exist even when the people are sitting next to each other in the same room.

## Software Engineering Reconstructed

I suggest that those three foundations provide a model that is both sound and practical. It applies in general to cooperative games, and in particular to both engineering and software development (or *software engineering*, as it is now safe to write).

The model helps us gain insight into both education and project management. On the education side, people should be trained

- in the vocabularies of their respective crafts, whether that be civil engineering, user interface design, requirements negotiation, programming, or testing;
- in the principles and techniques of invention and communication;
- in the mathematics and lessons of lean manufacturing systems, and
- in strategies for dealing with dependency flows and loops.

On the management side, managers should

- learn how to work from queue size and flow to improve their project planning strategies,
- pay close attention to the quality of the communication, morale and community present in their teams,
- help their teams to become better skilled at all facets of their respective crafts, and
- see that people get cross-trained in other specialties so they can help each other out, to smooth the inevitable bubbles that accrue randomly at the handoff points.

# 2'

# *Individuals: Evolution*

*The characteristics of individuals have, of course, not changed in the last five years. The item that I want to highlight here is the little section, "Supporting Concentration* and *Communication." The theme to discuss is strategy balancing.*

## Strategy Balancing

Many people think I want people *always* to sit closely together, just because communication is most effective when they are close. The idea I wish to develop here is:

*Just because a strategy is good much of the time, doesn't mean it is good all of the time.*

The point is encoded in the project leadership Declaration of Inter-Dependence (see http://pmdoi.org): "We improve effectiveness and reliability through situationally specific strategies, processes and practices." The cooperative game model serves to remind us that different situations call for different strategies, and that different strategies are called for at different moments in the same game (an example is chess, with it's different opening, mid-game, and end-game strategies; consider that the same applies to projects).

A good set of principles calls for a balancing of forces (this is, to my mind, a basic test of soundness). The principles should show when something is too much or too little for a given situation. It is the balancing of the forces that generates optimal strategies.

Let's look again at the matter of collocation and communication: Communication is *most effective* when people are face to face, but it is not always the case that "most effective communication" is the top priority.

- Sometimes we want peace and quiet (see the discussion of drafts in Convection Currents of Information, in Chapter 3). Sound barriers help create peace and quiet.
- Sometimes we want time to think, or to run a conversation when we are not all present at the same time. Asynchronous mechanisms, such as voicemail, email, and text messaging serve well here.
- Sometimes we want to be able to study an idea at our leisure, and so want a written form, perhaps on paper, so we can study it, think about it overnight, reread it, mark it up.
- Sometimes we want to dilute a particular person's overly strong personality. In this situation we will prefer a less rich communication technology such as the telephone (I have been told stories of people who deliberately use the phone with their bosses, for just this reason).

The principles plus your priorities help you create an optimal balance for the situation. For example, when setting up your office, balance osmotic communication against "drafts" to get a space that works for you.

As a larger example, here are two, opposing strategies created from the same principle. Watch how the strategies can be combined in interesting ways.

The strategy of Osmotic Communication (Chapter 3) suggests that people who need to communicate a lot should sit so they overhear each other in their background hearing. They learn who knows what and which issues are current. They can respond very quickly to the information flow around them. The *Expert in Earshot* strategy (Cockburn 2002z) applies Osmotic Communication to professional education.

The opposite strategy is the Cone of Silence[7] (Cockburn 2005cos). In that strategy, one or more people are sequestered away from everyone else, for the sole purpose of *avoiding* communication.

How can the Cone of Silence possibly be useful? The situation that clarified the matter for me was a project in which the technical lead was supposed to do the most difficult programming, and also educate all the people around him and also sit in meetings to plan future extensions to the system. He never had quiet time to do his own work. The project was falling farther and farther behind schedule.

After trying several other strategies unsuccessfully (including asking his manager and colleagues to stop pestering him), we used the *Bus-Length Communication Principle* to try one last-ditch strategy.

***Bus-Length Communication Principle:*** Communication between people drops off radically as soon as their (walking) distance from each other exceeds the length of a school bus.

(What is interesting as a side topic is that we were so attached to the strategy of Osmotic Communication that this was our *last* thought for fixing the problem!)

Th Bus-Length Communication principle is nothing more than a mnemonic restatement of what researchers have found and even graphed (Allen 1998, Olson 1999?). However, until facing that particular situation, I had always interpreted the research and that principle to mean, "Put people closer together." I had always wanted more communication. In the project at hand, however, we wanted the opposite effect; we wanted people to *stop* disturbing the technical lead. So we found him an office upstairs, at the far end of the rather large building, around

[7] The name of this serious strategy playfully parodies the gadget of the same name from the 1960s spy sitcom "Get Smart." There is no connection between the two except a sense of humor.

many corners, certainly farther away than the length of a school bus.

The result was nothing short of remarkable. Exactly according to the principle, people stopped bothering him. He found the time and peace of mind to program at full speed, and within a couple of months the project went from being impossibly behind schedule to ahead of schedule.

(An additional topic to consider is another strategy in play here: give the best programmer time, peace and quiet, and that person can sometimes get astonishing amounts programmed in a remarkably short time. But that is a different strategy with different boundary conditions.)

Having seen the Cone of Silence succeed, I suddenly started noticing its application in many places (Cockburn 2002cos). One example was when IBM wanted to enter the PC business – they located the PC development team away from any IBM development laboratory, so the team would not be disturbed. Another example is the fairly common strategy to send a group to an off-site location for a weekend (or week) to get a lot accomplished in a short time.

The Cone of Silence and Osmotic Communication are two, diametrically opposed applications of the Bus-Length Communication principle. They can be used together, in alternation, or one inside the other.

Putting a team off-site in a war room setting is applying Cone of Silence to the team with respect to its external connections, Osmotic Communication within the team. In the project described just above, we used Osmotic Communication within the project team, and only the technical lead occupied the Cone of Silence.

Strategy balancing should be used when arranging office furniture. As more people gather, their conversations become less relevant to each other. At some point, it becomes better to separate them than to cluster them. Many would-be agile developers forget this. They fill a space with more and more people. Those people become depressed, because the noise gives them headaches and because they get less work done.

Balance the strategies. Decide when the group size becomes too large, and then decide where osmotic communication is useful, where it is not.

Every strategy has built-in limitations. For Osmotic Communication, the first limitation is that optimal seating will change over time. At some point, you must decide whether to leave the now-suboptimal seating in place, or to disturb people by rearranging the seating.

Its second limitation is that osmotic communication is difficult to arrange when people are working on several projects at one time. The immediate strategy to apply in this situation, of course, is to rearrange the project staffing and time allocations so that they aren't working on many projects at once (this strategy has been heavily studied, see Goldratt 199x? and Anderson 2003?, for example). When you do this, you are likely to run into the next problem, which is that people have become too specialized.

The principles, problems, and strategies link one to another. Use of a strategy introduces a new problem, which requires a new strategy. Sooner or later, you choose one principle or strategy as dominating the situation and driving the rest.

As a starter suggestion, do whatever it takes to let people work on one or at most two projects, but not more. Serialize the projects, and apply osmotic communication as practical. Balance the strategies and principles from that starting point. In all cases, balance your strategies according to their boundary conditions.

# 3'

# *Teams: Evolution*

*As with individuals, not much as changed about the way teams work. What has changed in the last five years is the attention given to team building and team quality within software development. The project leadership Declaration of Inter-Dependence raises team behavior to a top-level concern: "We boost performance through group accountability for results and shared responsibility for team effectiveness." There is, however, one extension I need to make to the discussion of the office layout described in the previous chapter.*

## A Sample Office Layout Revisited

I decided to save some page space on Figure 3-13 by showing only the programming room in Ken Auer's building. Ken correctly took me to task for showing only that one room.

My mistake showed up as I gave talks on the idea of convection currents of information. Having more space available in the talk slides, I showed the entire office plan (Figure 5'-1). During those talks, I found myself discussing the uses of the private work area, the kitchen, the library, and the meeting room, and noticed they each contributed a separate value:

- People go to the private work area when they want to do personal computer work.
- They go to the kitchen to decompress and get some refreshment. There, they discuss both out-of-work topics and some work topics. Both help the team. People relax, talk and increase the amicability and personal safety with each other. Increasing these reduces their inhibitions about speaking to each other, which in turn speeds the flow of information within the team (Cockburn 2004).
- They go to the library to get some peace and quiet, and do research.
- They take noisy and directed conversations to the meeting room.

What I find interesting is how people declare their mental state as they move between zones. When they walk out of the programming zone, they are declaring, "I need a break (for some reason)." When they walk into the kitchen, they indicate their availability for social chitchat. When they walk into the library, they indicate their need for quiet. And when they walk back into the programming zone, they indicate that they are once again back in full force and ready to make headway.

Delighted with my findings, I told all the above to Ken. He replied, "Of course. You mean you didn't include the full floor plan in your book? Why would you show just the programming area?" Argh, that I missed so much!

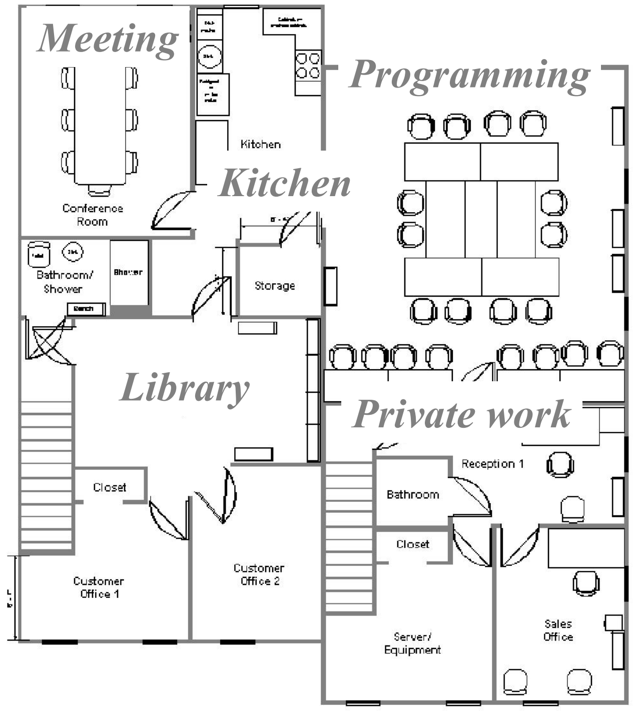


**Figure 5'-1.** Completed office layout (Courtesy of Ken Auer, Role Model Software)

# 4'

# *Methodologies: Evolution*

*Five years ago I did not make a conscious separation between methodology and project management. It's not that I mixed the two myself, but I didn't warn others against it. This is the time for that warning.*

*The first section in this chapter argues for separating project management strategies from process or methodology policies.*

*The second section addresses the tricky question: once these two are separated, and supposing the organization has a wide-ranging portfolio of projects to run, what can be said about methodologies across the organization?*

*The third section contains my best current description of how to talk about incremental, iterative and cyclical process.*

*The fourth section addresses how to write down your team's methodology in just an afternoon or two.*

## Methodologies versus Strategies

After running a series of successful projects, it is tempting to extract the success factors from those projects and declare them policies for future projects. This is a dangerous thing to do, for three reasons:

- We may have extracted the wrong "success" factor, and are now setting as policy something that was either incidental or even detrimental to the success of the project (which is what I suspect happened with waterfall development).
- Even if we extract the correct success factor for that one set of projects, there is nothing to promise that that factor will be the success factor on the next project (see the discussion of balancing strategies on page ???).
- The project manager, or the team on the ground in the situation, is probably best suited to judge what is the best strategy to apply at any given moment.

Strategies that generated success on a small set of projects should not be cast as global process or methodology policies. Yet this is exactly what most people think methodologies are for.

A methodology is just the set of conventions a group of people agree to follow (less optimistically: the *rules* they are *told* to follow). These conventions can touch on any topic at all, from clothing to seating to programming to documentation to work-flow.

The group's conventions are likely to shift over time. This is proper. And, as simply a set of conventions, it is quite possible that the group will never write down the full list of conventions.

At some point, someone may decide to write down a selected subset of the conventions to apply to other projects. At this point the methodology becomes a formula, a theoretical construction that abstracts projects and people. Necessarily, the people and the situations on the next project will be different.

Knowing all the above, the highly experienced leader may still decide to cast as policy selected conventions that *usually* work well, that are *likely* to work well for *most* projects, as a way of training and guiding less experienced project teams. This is sensible, as long as experienced leaders are around to tell when the conventions are a misfit to the situation at hand.

However, the experienced leader is likely to move on, leaving the organization with an outdated set of strategies hardening into corporate policies. At this point, the social conventions become ritualized procedures. The organization has a *Shu* type of description for a *Ri* class of activities; and as Russ Rufer quipped, "We can't make one *Shu* fit all."

If project management policies, practices and strategies do not belong in the methodology, where do they belong?

They belong in a strategy manual, a collection of project stories and strategies (see for example Appendix A of *Surviving Object-Oriented Projects* (Cockburn 1998) for a sampling of  such strategies). The organization can use such a collection to train junior project managers and development teams. This training will highlight the strengths, weaknesses and limitations of each strategy. The strategies, taken away from the process policy handbook,are placed where they belong, as part of the project management *craft*.

Project managers should be evaluated, in part, on how well they adjust their strategies to the situation.

It may seem to certain skeptical readers that the situation I am describing is unique to software development. It is not. In *Simultaneous Management* (Laufer 1999?), which deals primarily with civil engineering projects, Laufer describes how project managers have to move from cost-prioritized to flexibility-prioritized strategies from one day to the next (and indeed, between any priorities that may arise). That book is good reading for project managers in any field.

## Methodologies across the Organization

Notwithstanding the appropriateness of extracting the project management strategies from the corporate methodology, and the recognition that some tailoring is needed within each project, executives quite reasonably want their projects to have *some* conventions in common. They want people to be able to understand what is happening when they transfer onto another project. This requires having *some* conventions across the organization:

- *Definitions of common vocabulary*. People need to be able to understand each other. They need to understand just what is meant and what variations are allowed by particular phrases. For example:
    - Does *iteration* only mean "planning window," or is it required, when saying "iteration" that code is written and unit-tested; or not only integrated, but tested, documented and ready for deployment?

   - What is really supposed to happen in an *iteration planning meeting*? Is the demo to the customer to happen inside that meeting or before it; are the user stories supposed to written before it or during it; does it contain design discussion or is design done later?
   - What does *delivery* mean? Does it mean real deployment to the full user base, test deployment to a friendly user, or deployment to a machine in the test department?

I was called in once because every team used those terms differently, and they could neither understand what other people meant nor what they were doing on their projects.

- *Definitions of common formats.* Although two teams may not have to create the same work products, it is useful if they use common formats for whatever work products they do have in common.
   - Every project should probably have a vision and scope document.
   - Every project needs a way to report its status. Finding ways to represent status for all possible projects is very difficult (see "A Governance Model For Agile and Hybrid-Agile Proj-ects" (Cockburn 2005) for a progress-reporting format that covers a lot of projects).
   - The Unified Modeling Language (UML) helps people use a common language when drawing a domain model or sequence chart. The organization may add additional rules to how those models are presented (naming conventions, complexity in a single drawing, use of vertical and horizontal placement, and so on).
- *Communication requirements*. The testing and marketing departments need advance notice of upcoming product releases. The organization may set policies in place about how much advanced warning they get ("two months"), or how reliable the schedule should be ("when the schedule is estimated to be within 10% accuracy"), and what sort of information they are to be given.
- *Baseline project management strategies*. The organization may require every project to run an Exploratory 360º (see Cockburn 2004cc) before receiving committed funding. It may require each project to deliver system increments to real users every three months or less. It may require automated testing, and perhaps test coverage policies. These policies may be challenged on a project by project basis.

Among the strategies most often mistakenly put within the methodology are the sequence and timing of gathering information. It is common to require that all requirements be gathered before any design is done, or that all user interface design be done before any software architecture is done. By now it should be clear that any strategy calling for "all-of-one-before-another" has severely limited applicability.

Even sophisticated developers and managers fall into this trap. In teaching how to write use cases, I describe how to select *what* to write, and the format in *which* to write. People – even experienced people – immediately jump to the conclusion that I want all of the use cases to be written before any design is done, or that each use case should be written to completion at one time. Neither is true.

The choice of how much of one use case or how many use cases to write at one time should be reevaluated continually during a project. The alert team will come up with different answers at different moments on a project, and certainly different answers on different projects. This is in keeping with the cooperative game model.

## Processes as Cycles

Most people describe the development process as a linear process over time, as I also did until 2001. These description were never correct, or were too complicated to read. Some people simply draw a big circle, or perhaps circles connected to each other either in a bigger circle. These are either look wishy-washy or are again too complicated to use. The generic circle doesn't map to a calendar in any way that helps a project manager. Around 2001, Don Wells solved the problem, in part, by illustrating XP using nested cycles at different time spans for different practices.

Figure 10-1 shows an incremental development process using nested cycles. This nested-cycle idea fits most projects, agile projects particularly well. The version in Figure 10-1 is intended to show both the linear aspects of the project (read the activities clockwise around each ring) and the cyclical aspects (go around the inner rings multiple times for each outer ring). A linear, or time-based unfolding of Figure 10-1 is shown in Figure 10-2.

Neither the circular nor the linear version is completely correct, because there are interactions between the inner and out cycles. For example, both iterations and deliveries contain planning, reflection and celebration. Of course, one typically does not hold two planning sessions at the start of the first iteration in a delivery cycle, or two reflections and two celebrations at the end of the last iteration in a delivery cycle. One merges and adjusts them accordingly. Trying to show these interactions on either form of diagram is too complicated. In practice, people have no trouble understanding that the planning at the start of a delivery cycle will include both delivery and iteration time horizons, and similarly for the end-of-cycle activities. This is where practice is for once simpler than theory.

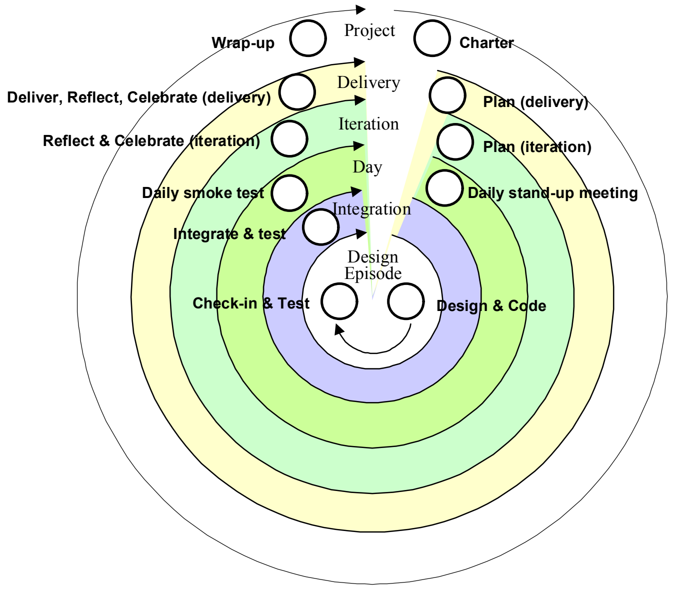


**Figure 10-1.** Nested-cycle model of an agile process

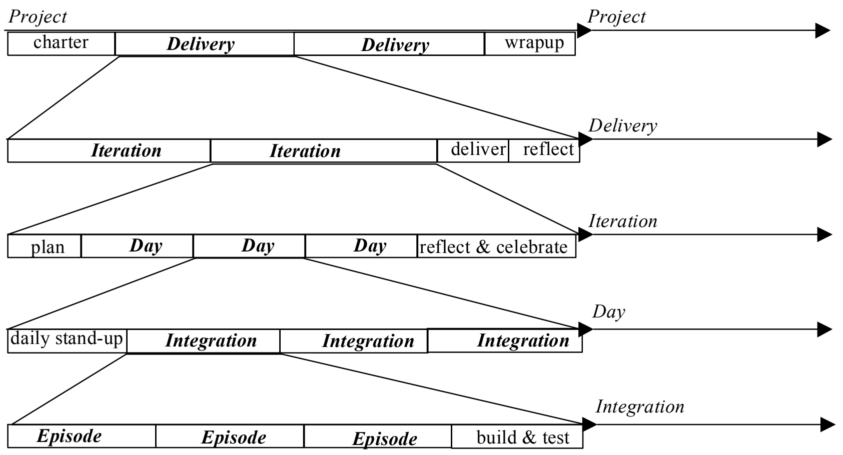


**Figure 10-2.** The cycles unrolled (needs fixing to show delivery planning!!!???)

Read Figure 10-1 as follows:

- A project starts with a chartering activity, which could take from an hour to several weeks. Within the project are some number of deliveries, after which the project ends with a wrap-up activity of some sort.
- The delivery cycle starts with an update to the project plan, and contains some number of iterations. It ends with the delivery of the system to some number of users. Agile teams tend to reflect after each delivery, to improve their process and the product.
- Each iteration starts with a planning session. The iteration ends with a reflection workshop and, in the best shops, a celebration event of some sort. For the last iteration in a delivery cycle, it is probably the case that the iteration reflection session is held after the delivery, as part of the delivery reflection workshop.
- Within an iteration, there are two cycles that operate at their own, independent rates. There are regular weekly activities, such as the Monday morning staff meeting, the daily stand-up meeting, and the daily smoke test of the system. There are also regular daily activities, such as the daily stand-up meeting and the daily full integration with testing.
- Within and overlapping the daily and weekly cycles are integration cycles and design episodes[8]. Each design episode consists of doing some design, programming the design (and the unit and acceptance tests to match), checking in the new code and tests, and running the tests. The programmers may go through several design episodes before checking in and integrating their code. There are usually multiple design episodes within a day, which is why I show the design episode as the inner cycle. Some teams integrate mul-

[8] Thanks to Ward Cunningham for this term. See http://c2.com/ppr/episodes.html

tiple times a day, other teams integrate every few days.

What I find particularly useful about Figure 10-1 is that it allows us to capture periodic "operations" activities such as the weekly status meeting and the daily integration in the same format as "progress" activities such as chartering the project and designing new features. Each cycle is quite simple in itself, which helps in the presentation of an otherwise complex process.

People, as they go about during the day, have no trouble in knowing what cycle each activity belongs to. While attending the weekly status meeting or the daily stand-up meeting, they know they are in an operations activity. When they walk out, they know they are entering a design episode, and vice versa. It is not confusing to them that they switch between multiple cycles in the same day.

Presenting the process as nested cycles has one more benefit – most people misinterpret a linear drawing of a process as being "waterfall," even when it is specifically not. This misinterpretation is because the linear process drawings really describe *a dependency graph between work products*! The dependency graph should be read right-to-left, for its dependencies, not left-to-right for a process.

The "process" diagram shown in so many presentations shows *Gather requirements* pointing to *Do design* (and *Write code*), which points to *Run tests*, and so on, leading to *Deliver system*. Although to the unwary it *looks* like a process diagram, what it really is saying is that the people can't deliver the system until the tests are run. They can't run the tests until the code is written. They can't write the code until they learn what the requirements are, and so on. Although these are all true statements, they are not really informative, and they particularly don't say the really interesting things that need to be said:

- *How many* requirements, *at what level of completion,* are needed before design can usefully get started? (The answers are usually "relatively few" and "relatively low", by the way.)
- How does feedback from the design team affect the amount of detail put into the written requirements?
- *How much of the system* must be designed and programmed before useful integration and testing can be done? ("Relatively little")
- *How much of the system, at what level of correctness,* is needed before being useful to the users or suitable for user review? ("Relatively little" and "relatively low").

There are two important points to take from this discussion:

- Consider processes as nested cycles. This simplifies their description, allowing capture of both progress and operations activities, and allows incremental / iterative development to be described much more accurately.
- Don't confuse dependency diagrams for process diagrams.

## Describing Methodologies More Simply

In Chapter 6, I describe Crystal Orange, Crystal Orange Web and Crystal Clear using three different formats. The Crystal Orange Web format was the newest, and I experimentally recommended it as a faster and simpler way to capture methodology rules.

I am happy to report that several people have used that format, and have reported back to me that it works well, and fast.

Bill Barnett at Wachovia Bank was setting up five concurrent agile teams for the first time. His first group were using an experimental set of conventions, so of course they didn't write those down. A month later, when he set up the second group, the first team conveyed the current set of conventions to the second team verbally, and again they didn't write them down. But by the time the next three teams were getting started, both management and the teams themselves were asking to have the conventions written down.

Normally, writing down a methodology is something that takes several months, results in a long document that almost no one reads. Bill feared his task accordingly. I suggested that he simply write the conventions as a list of sentences, and then cluster them into topics. This he did, and wrote to me a few days later, saying that he had drafted the entire, approximately 12-page document in one afternoon! Another afternoon of review with the team (and touching up the text), and he was done.

Greg Luck, then at Thoughtworks (working out of Australia), also picked up that documentation style, and became proficient at it. He was able to join a project and within a few days have captured a first draft of their methodology.

Why is this important? For Bill Barnett, a short, readable document was needed so each new team could go through it and see what they had to set into place.

In Greg's case, it was more for the Thoughtworks client than for the development team. The people on the development team knew what they were doing. The client didn't know what conventions the development team was operating under. By producing a document quickly, and making it short and readable, the team was able to reassure the client about what was happening, the client did not have to pay months of consultant salary for it, and other departments could see how they would interface with the development team.

Here is a summary of how to do this writing:

- Plan one afternoon for the first draft, then review each sentence with the team and update it. You may need to update it quarterly as the team conventions change.
- If you are a member the team and have been working in a consistent way for several iterations, you may know enough to just sit down and do the writing. If you are coming from outside the team, gather the team for the afternoon and create the sentences together.
    - Have a (fast typer) scribe use a word processor projected onto a screen so everyone can see and comment on what is being written. You will probably want a facilitator in addition to the scribe.
    - Capture the sentences in a list.
    - The person doing the typing asks questions for clarification, the people in the room comment on the content.
    - Don't get locked up in wordsmithing details, just get the ideas down. There will be a review session for correcting small errors.
- Write down one sentence for each convention in use. These can cover any topic at all. Here are some sample possible sentences:
    - "Everyone sits in one room."
    - "Each multi-specialist function team sits together; the different function teams sit as closely together as they can manage."
    - "The marketing and sales representative visits the team each morning to see the progress, answer any questions, and report on changes in –direction of new observations about the market and the competition."
    - "Programmers check in code several times a day."
    - "2:00 to 4:00 p.m. is reserved for focus time each day; people who agree can work on a topic together, but no meetings are called, and outside calls are blocked."
    - "The traffic light (or other information radiator linked to the automated build machine) turns green after a successful build and red when a build fails. Sound broadcasts (e.g. cheer or baby crying) are up to the discretion of each team."
    - "Teams located across time zones have a daily stand-up meeting once a day by telephone to pass along what each has done or is planning to do."
- Cluster the sentences by topic (this can be done later).
    - Some will describe basic workflow, who hands what sort of decision or document to whom.
    - Some will describe formats, such as the format of the story-tracking board or the use case or the user story.
    - Some will describe communication patterns, times of meetings, quiet time, and so on.
    - Some will describe morale raising conventions, such as when Doom gets played, or group volleyball games, or the Friday afternoon wine and cheese gathering.
- Review the document with the team.
    - Collect the team for about two hours. Go through the document cluster by cluster and collect corrections.
- If you think you will have trouble completing this review in two hours, get someone who is good at facilitation to lead the session and make sure it doesn't get bogged down.
- If there is real disagreement on some point, appoint a separate working session between just with the people who are concerned with that point, and move on. After that working session, see if you can get agreement to the changed session in the daily stand-up meeting or by email.
- Review and update the document after the next iteration or after three months at most.

That's it, that's the whole thing. Now try it.

# 5'

# *Agile and Self-Adapting: Evolution*

*This chapter is quite long, because our understanding of agile development has evolved a great deal since the writing of the agile manifesto.*

*At first, people asked, "What does agile development really mean?" That was accompanied by many interesting ways of misconstruing the message (which continue to this day). Then the questions became: "Where is the sweet spot of agile, and what do you do outside of the sweet spot?" and "What is the role of a project manager in a hyper-agile project?"*

*We shall take a look here at these questions, along with how agile methodologies have evolved.*

*Also suitable for discussion is the way in which the agile approach gets applied outside software development. The cooperative game model applies to business, journalism, law, and entertainment, as well as software development. Agile development applies to those areas, but also to civil engineering and construction projects, such as house or airport construction.*

## Misconstruing the Message

Over the last five years, people have been quoting agile rhetoric without adopting the elements that make the agile approach work. This section is my little attempt to correct some of the misquotes of the agile model that overzealous would-be agile developers inflict upon their managers.

Before getting into ways to misconstrue the message, let's first review what the message is supposed to be (or at least how I think about it).

- First, create a safety net for the project, one that increases the chances of a successful outcome. This safety net consists of the following:
  - Everyone on the team sits within a small physical distance of each other (osmotic communication).
  - They team uses an automated tool suite that integrates the code base every short while (continuously, each half hour or each hour), runs the test suite, and posts the result on an information radiator.
  - The team, working in small increments so they check in code and tests several times a day, creates unit and system tests for the automated integration system to run.
  - The sponsor and key users visit the project team often (daily or once a week) to offer insights, feedback, and direction.
  - The team delivers running, tested features (RTF)[9] to real users every week, month, or quarter.
  - Following each delivery, the sponsors, key users and development team get together and review their mission, the quality of their product, and their working conventions.
- *After* the safety net is set in place, see what can be removed from the development process. Typically, requirements documents can be lightened, planning documents put into a simpler form, and design documentation done in a different way. Also typically, more training needs to be brought in, more test automation done, designs simplified, and community discussions increased.

The point I wish to make is that simplifying the process is a reward the team gets for putting the safety net into place. Many would-be agile developers take the reward without ever putting the safety net in place.

Here are some of the harmful misquotes of the agile manifesto I have heard.

### "Iterations must be short"[10]

A macho culture has developed around making iterations shorter and shorter. Teams that fall into this trap end up with iterations that are too short to complete features within. The iteration devolves to a mere planning period.

A telltale sign of that the iterations are too short is that the team gets preoccupied with the word *iteration*. We see teams charting one iteration's worth of work without seeing how the various specialists' work fits together or how the output of the iterations grow toward a useful delivery.

The mistake is assuming that short iterations are uniformly good. An iteration should, in theory, permit the following:

- The business experts work out the details of the features to be built in this iteration.
- The programmers program them.
- The users, business experts and testers check that what got built is adequately correct, and comment on what needs to be changed.
- The programmers change the software to take those comments into account.
- They repeat the last two steps until the business experts, users and sponsors feel the features are suitable for deployment.

Now imagine a fairly normal team of business experts, programmers and testers (UI design experts will notice the usual absence of UI designers!) who have chosen an iteration length of two weeks. The team finds it cannot do all five steps in just two weeks. So, they decide to pipeline their work:

- In the first two-week cycle, the business experts work out the details of feature set 1.
- In the next two-week cycle, the business experts work out feature set 2 and the programmers program feature set 1.
- In the next two-week cycle, the business experts work out feature set 3, the programmers program feature set 2, and the testers test feature set 1.

They run into a problem immediately: The groups are all working on different feature sets. Discussion that felt like collaboration when they were working on the same feature set now feels like interruption. When the testers in the third iteration ask the business experts a question about

[9] A neat term coined by Ron Jeffries (Jeffries URL).

[10] This issue is discussed at length in the article, "Are Iterations Hazardous to Your Project?" (Cockburn 2005z)

feature set 1, they force the business experts to remember issues examined two cycles ago, a month in their past.

To protect themselves, *so they don't get bothered* while they work out their new decisions, the business experts write documents to give to the programmers and testers. Similarly, the programmers create documents to hand to the testers.

In pipelining their work, they have created a time barrier, which can be crossed only using archival forms of documentation, which as we saw in Chapter 2, are expensive, tedious, and relatively ineffective forms of communication.

Some agile experts are really opposed to lengthening the iteration, and this for good reason. As the iteration lengthens, people have a tendency to delay their work, pushing it all toward the end of the iteration. As Dan Gackle once wrote, describing their team's use of one-month iterations:

""Meander, meander, meander, realize there isn't much time left, freak out, get intense, OK we're done. Repeat monthly."

Good agile coaches steer carefully to avoid these two traps, shortening the iteration to get results quickly, lengthening it to get proper communication and feedback, keeping people focused on making progress at all times.

*"So what should we do?"*

Become preoccupied with the word "delivery."

Understand that each iteration builds toward the next delivery, and know exactly what each iteration contributes to each the next delivery. With this in mind, determine the length of the iteration from factors having to do with knowledge of the domain, the technology, the user base, and the communication patterns. Consider at what pace the team can deliver running, tested features to real users, while also collecting and incorporating feedback from those users. Balance brevity of iteration with the quality and completeness of team interactions.

Retain in all cases is the idea that a feature is not "*done*" until the key users have had a chance to see it and change it at least twice. Although this is not in the agile manifesto, it is a critical policy for the software to be considered successful in use.

## "Agile teams must be collocated"

Distributed teams lose the high levels of communication, trust, and project safety afforded by collocation. Project managers who once led collocated teams but then got assigned distributed teams repeatedly comment to me about the loss of trust and the difficulty in rebuilding it. This is independent of whether they were using an agile approach or not.

While it is clear that working with a distributed team makes agile development harder, it doesn't mean you can't use the agile model. The would-be-agile distributed team must put more energy into its communication and team-building strategies, *and* find times to get the team together face to face.

Automation tools and high-speed communication technologies help. Web cameras and microphones attached to workstations let people to see and talk to each other in real time. Instant messaging lets them trade code in real time. Net meeting software lets them pair program together. Automated build-and-test tools let people in different time zones work from the same code base.

Distributed teams must be careful to work in small increments, integrating and testing multiple times a day. Because communication is not as fast, synchronizing the code base becomes more important as the team gets separated.

A distributed team must put more energy into simplifying the system's architecture (Cockburn Cutter ref). The reason has to do with Naur's "Programming as Theory Building." Since the communication channels between people are constrained, it is more difficult for them to build a shared theory of what they are developing. The more complicated the architecture is, the more complicated the theory is, and the less likely they will work to the same decision principles. Conversely, with a simpler architecture, they have a greater chance to keep the theory intact across distance and intermittent communication.

Even with all that, there is still no substitute for personal, face-to-face encounters to build personal safety and trust. Good distributed teams get together initially and periodically for both technical work and social interaction. Through these get-togethers, they build shared mental models of technical topics and personal trust.

Part of the trouble with distributed development is that executives and managers have never learned how to develop software *here*. They come to think that perhaps they know how to develop software *there*, or if there is going to be a mess, it is better out of sight, and if there is going to be a failure, the failure will be less expensive if it happens *there*. What I am interested in is learning and teaching how to develop software *here*, for any here. Once we learn this, then we can develop software both *here* and *there*.

I don't expect to reverse the trend of distributed development. I do wish executives to consider more of the issues involved, particularly the higher communication costs.

A success in this regard was when one executive, who was running projects split between Pennsylvania and North Carolina, turned to her colleague, and said (roughly): "I think that we could have run that last project just in Philadelphia. Even though Jim (the best programmer) is in North Carolina, we could have used Martin (in Philadelphia) instead and saved a lot of trouble." She was balancing the increased expertise she could get in multiple sites against lower communications costs and higher team productivity she could get with a collocated team.

*"So what should we do?"*

Pay even closer attention to the agile values than collocated teams do. Simplify the architecture. Add new face-to-face communication opportunities. Use an automated build machine and check your code in every few hours.

Keep your ear to the grapevine and pick up new ideas being tried in other teams.

## "Agile teams don't need plans"

Early critics of the agile movement worried that we were dropping project planning, which would be dangerous to the project. Sadly, I have seen exactly this (dropping planning) happen all too often, and with the expected damaging consequences.

It seems that quite a number of over-zealous would-be agilists say to their managers, "If we are sufficiently agile, we don't need plans – we can always react quickly enough. If we need plans, we are just not agile enough." I have lost count of the number of executives who tell me that this is how their programmers present agile development to them.

The final value comparison in the agile manifesto states: "Responding to change over following a plan."[11] That is followed by the tempering statement, "While we value the items on the right, we value the items on the left more." Planning is useful in both cases.

The value comparison in the agile manifesto is intended to compare

- making a plan and periodically reconsidering whether the reality currently facing the team is better served by keeping to that plan or by creating a new one; against
- making a plan and continually trying to get back onto it.

When some aspect of the project doesn't work out as expected, a sudden staff change hits the team, or some external event changes the sponsors' priorities, the team must decide whether to get back to the written plan, or to change both the goal and the plan. It is not part of the agile manifesto to assert which is better – to get back onto the plan or to create a new one. It *is* a part of the agile manifesto to assert that one should make that choice deliberately.

[11] http://agilemanifesto.org

Except on certain exploratory projects, there is almost always enough information to make a coarse-grained plan that is aligned with the project and the organization's overall vision. This coarse-grained plan serves to make coarse-grained cost- and time estimates. Those estimates are badly needed by the sponsors to weigh the project against other possible initiatives.

We worked to repair the situation as we could in the Declaration of Inter-Dependence. The third principle in the DOI states:

> "We expect uncertainty and manage for it through iterations, anticipation, and adaptation."[12]

This phrase highlights that uncertainty is a factor that cannot be completely eliminated, but can be mitigated with the feedback mechanisms in standard use by agile teams (iterations, reflection, adaptation), *plus* making use of information that is already available – anticipation.

It remains to be seen whether upper management will accept that uncertainly cannot be eliminated. It also remains to be seen whether agile zealots will accept that anticipation has value.

In practice, agile teams tend to produce plans at two levels: long-term, coarse-grained plans, and short-term, fine-grained plans[13].

On one 18-month, $15m fixed-price project, we worked from 240 casual use cases[14], a list of external systems that would have to be touched, an estimate of the number of domain class and screens that would have to be built, and a release plan that consisted of a dozen bubbles with dependency arrows between them, with lines marking the five deliveries that would be made. Such coarse-grained long-term plans are typically used for projects greater than three months.

The short-term, fine-grained plan is used for time frames between one week and three months. Inside the one-month time frame, work items are typically broken down into pieces between half a day and three days long.

These two-level plans have several advantages:

- They paint an overall picture of the project and still allow room for movement.
- The long-term, coarse-grained plan show the overall direction, the timing, the approximate cost, and how the iterations fit together to deliver the best value over the long term.
- The short-term, fine-grained plan is used to coordinate teams in detail.
- Each level can be updated quickly.

[12] http://pmdoi.org
[13] Examples of each are shown in Cockburn CC 2004, pp ???-???.
[14] "Casual" use cases are written in only two paragraphs. They are described in (Cockburn weuc).

- The team can balance the needs of setting overall direction with tracking detailed work.

Any change in a long-term plan invalidates a large amount of work in a detailed plan, and we can be certain that there will be changes in the long-term plan. By expanding the detail of the plan just prior to starting an iteration or a delivery cycle, we reduce rework in planning.

*"So what should we do?"*

Create a coarse-grained long-term plan to know where the target is, and a fine-grained short-term plan for the next week, month or quarter.

***Sidebar:* Helmuth von Moltke**

Agilists often motivate their aversion to detailed long-term plans by quoting the German military strategist Helmuth von Moltke: "No battle plan survives contact with the enemy."

I found this a compelling quote, until I learned that there were *two* Helmuth von Moltkes who might have said it.

Moltke the Younger (Helmuth Johann Ludwig von Moltke, 1848 to 1916) has been (controversially) blamed for Germany's loss in the Marne campaign and consequently the First World War.

> "As Chief of the General Staff Moltke was responsible for the development and execution of the strategic plans of the German Army. There is considerable debate over the nature of his plans. Critics from the so-called "Schlieffen School" argue that Moltke took his predecessor's plan. . . modified it without understanding it, and failed to execute it properly during the First World War, thus dooming German efforts."[15]

If Moltke the Younger said that no battle plan survives contact with the enemy, then those opposed to the agile viewpoint would respond that this was spoken by someone who clearly hadn't planned well enough.

On the other hand, Moltke the Younger's uncle, Helmuth Karl Bernhard von Moltke (1800 to 1891, known as Moltke the Elder), didn't lose any wars for Germany, was very successful in battle and was eventually promoted to Field Marshall. If it was Moltke the Elder who said the quote, then the above retort can't be given, and "No battle plan survives contact with the enemy" meaningfully supports the view that even the best plans need frequent updating.

Fortunately, it was Moltke the Elder who said it.

## "Architecture is dead, refactoring is all you need"

Similar to not needing plans is the overzealous view that architecture is not needed: "If you are sufficiently agile, you don't need an architecture – you can always refactor it on the fly," say these people.

Let's look at both sides of the argument, with the idea that the extreme of either probably is not good for most projects (but possibly is good for some project somewhere).

Good agile developers start with simple architectures and then evolve them. I have seen good developers start with simplistic inter-computer messaging mechanisms and later substitute commercially provided messaging systems without difficulty. This pattern of development is called *Walking Skeleton with Incremental Rearchitecture* (Cockburn Cutter ref, Cockburn CC).

The advantage gained is that the team, sponsors and users have a system up and running early. From there, they can steer the feature set, the architecture, the staff composition, the use of technology, and the development process. The cost to the strategy is that the team must make changes to the system while development continues.

The countering argument is that a more complex original architecture would probably also be wrong in unexpected ways and would also need to be changed.

I have seen and heard of projects saved by deploying an early, simple architecture and changing it later after they saw where its shortcomings were; and I have seen and heard of projects that failed due to an overly ambitious initial architecture that could never be fully developed and deployed at all. I therefore consider the cost of upgrading a simple initial architecture as nothing more than the insurance premium for increased project safety and for reduced risk afforded by having an early, running system[16].

Changing the architecture is possibly a good thing in itself. I ran across the following argument somewhere along the way, and have examined it closely with some friends. We think it is sound. See if it makes sense to you.

[15] http://en.wikipedia.orfg/wiki/Helmuth_von_Moltke_the_ Younger and http://en.wikipedia.org/wiki/Helmuth_Karl_
Bernhard_von_Moltke

[16] A longer discussion of paying extra for flexibility is part of the articles series "Learning from Agile Software Development" (Cockburn Crosstalk agile ref)

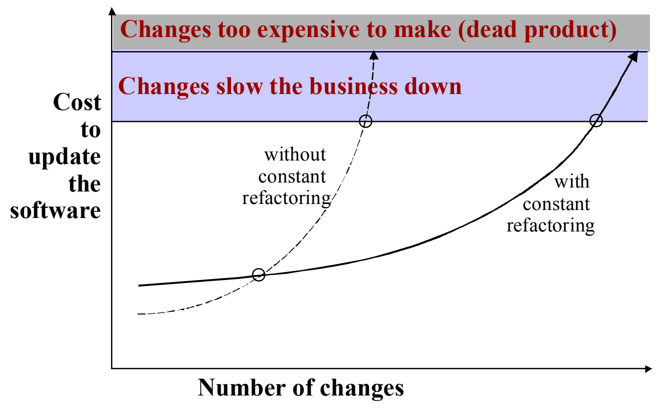


**Figure 5'-1.** The graphical argument that businesses get held up by not refactoring systems constantly earlier than when they refactor constantly (see text).

Figure 5'-1 shows the cost of refactoring versus not refactoring code over time. The core idea is that when people make changes to the code, they add small, extra bits of complexity, perhaps an extra branch condition or code duplication. These little bits of complexity are not independent, but start to intertwine. A new branch condition is based on a previous branch condition, code duplication gets copied, modified and gets its own new branches. Once these little bits of complexity start to intertwine, the total system complexity goes up, not linearly as it would if they were completely independent, but exponentially.

The cost for updating the software looks like interest growth on a financial loan: $(1+c+u)^n$. Here, *c* is the necessary complexity added to the code through the new business rules, *u* is the cost for understanding the unnecessary intertwining across the changes, and *n* is the number of changes put into the code. Even if *u* is a small number, over thousands of changes across a period of years the cost to understand and modify the code grows to a point where the system simply can't be updated in the time frame needed by the business.

The graph shows how it comes about that modifications to a software system start out easy, but become harder and harder over time until finally the programmers tell the business owners, "We can't make any more upgrades to this system. We have to start over."

Refactoring constantly to clean up the code, the curve becomes exponential with a slower exponential. It does cost time to clean up each new piece of complexity, but these clean-up activities don't intermix with each other, so the cleanup activity costs. The cost is $(1+c)^n + (r*n)$, where *r* is the cost of doing the refactoring. The code still grows more complex, but the growth is slower, and so the system lasts longer before slowing down the business (rather like paying a 5% loan instead of an 11% loan).

Notice that initially, the refactoring cost is greater than the cost of understanding a few extra bits of complexity, but over time, the two trade places, as shown in Figure 5'-1.

I note that the increased cost of understanding the system as it grows in complexity is part of what Peter Naur talks about as "theory building" in Appendix B. It is an intrinsic cost, meaning that it can't be removed.

So far in the argument, we have seen that starting with a simple architecture is good and refactoring is good. Where's the problem?

The problem is that sometimes the development team has information available that can reduce the refactoring cost. Ron Crocker's example illustrates (Cockburn cutter arch article ref):

Ron Crocker was lead architect at Motorola on a new mobile phone platform that covered from the transmission towers to handheld and PC-based devices. They were faced with me one core architectural question, whether to design the mobile phones to handle both circuit-switching and packet-switching traffic right from the start, or just circuit-switching. They suspected they would need both. The refactoring approach would have them design the system to handle circuit switching, to add the capability for packet switching only when that became strictly necessary, and to refactor their architecture at that time.

Ron's development team consisted of 250 people in six countries, and I knew that this was a factor in his decision to design for both right from the beginning. So I asked Ron to estimate the cost of communicating the new redesign under the simplest circumstances, that their project consisted only of 30 people, all located in the same building on the same floor. After some pause, he estimated that they might have been able to migrate the developers to understanding the new architecture in "merely" a few weeks.

"But why would I want to do that?" he asked, meaning, why would they want to cost the entire team two weeks of their time, when the requirement was known from the beginning?

In fact, with 250 people in six countries, the cost of communicating the revised architecture would have been enormous. Since the requirements were known from the start, it made sense for his team to design for both right from the start. It is relevant to mention that his team had designed these types of systems before, so they were embarking on a familiar, rather than unfamiliar, path.

On another project, the lead architect, who was contracting out the work to an XP team, watched in frustration as the subcontractors ignored the known requirements to support a variety of peripheral devices, and implemented a series of designs one after the other, adding each peripheral device as a surprise addition each time. He was frustrated because he knew they could have cut quite a few revisions if only they considered the future requirements for the peripherals much earlier.

This is where some balancing of the issues is called for.

We have learned in the last ten years that simple designs with clean interfaces really can be altered and extended with quite low cost. The cost is low enough that very often, a simple, clean design followed by a targeted change comes out at a better overall cost than an overly complicated initial design that takes longer to implement and is difficult to both learn and modify.

What we need to take into account is that *good* agile developers are quite experienced and have good design reflexes. Their designs are simple, clean, and refactor easily. The refactored design is again tidy and clean and allows itself to be refactored again.

Inexperienced designers are likely not to have such good design reflexes. They can benefit from being forced to think through the issues of scalability, performance under growing loads, and probable changes in requirements. The thinking time is well spent, even if they decide to implement a simpler design.

This has been a long section. Architecture is not dead, and refactoring is not a cure-all. Thinking about design is still important, as is developing good reflexes about simple, tidy designs.

*"So what should we do?"*

This quesiton one has no simple recommendation, as far as I can tell. Different people think ahead to different time horizons.

I like Ron Crocker's advice[17]: Create a simple architecture that handles all the known "big rocks" (requirements particularly hard to incorporate late in the project). Handle the "small rocks" as they appear during the project.

A good architecture is simple and understood by all on the team. This takes thinking. At the same time, good agile teams have demonstrated that with clean designs, even significant changes can be make remarkably quickly.

## "We don't need no *%@! managers! "

Along with plans and architecture, over-zealous agilists dismiss the role of project manager. Here again, we need to look at the issues involved.

From interviewing project teams and managing projects myself, I have made the following list of ten factors critical to project success, in five areas (see Figure 5'-2):

[17] (Cockburn cutter arch ref)

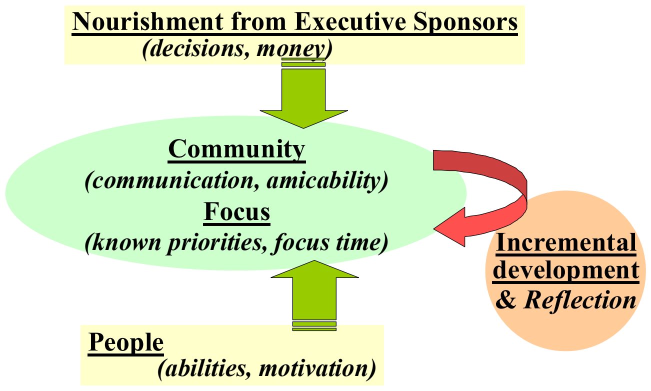

Nourishment from Executive Sponsors
(decisions, money)
Community
(communication, amicability)
Focus
(known priorities, focus time)
Incremental development & Reflection
People
(abilities, motivation)

**Figure 5'-2.** Ten critical factors in project success

- Sponsorship: Without adequate sponsorship, the project will starve for resources. Sponsorship consists of
  - *Money* – for staff, training and equipment
  - *Decisions* – about direction and priorities
- The people: The people have to be both capable and willing to do the work. This includes
  - *Ability* – meaning *talent*, the latent part of ability, and *skills,* the trainable part
  - *Motivation* – the willingness of the people to participate in both social and technical sides of the endeavor
- Incremental development with reflection: Without incremental development, the team doesn't have a chance to detect and correct problems in its direction and process; without reflection, the team won't actually redirect itself. These are
  - *Incremental growth* and integration of the product, preferably with deliveries
  - *Reflection* after each increment about their direction (including target market and feature sets), their rate of movement, and their process.
- Community: The quality of the community affects the speed of movement of ideas; it is one place where dynamic shifts in team productivity can occur. Community includes
  - *Communication* paths – the speed of the project is limited by the speed at which information moves between people (erg-seconds/ meme, see p.???).
  - *Trust* issues (personal safety and amicability) – if people won't speak or won't listen, the information won't get from where it is to where it needs to be
- Focus, meaning both direction and time.
  - *Priorities* – knowing the two or three key items to work on at any time
  - *Focus time* – having quiet time to get the work done

Figure 5'-3 shows role of the project manager.

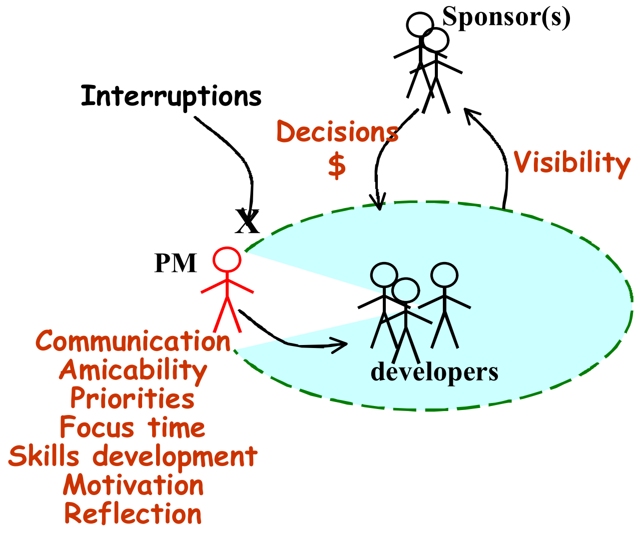


**Figure 5'-3.** Role of the project manager in modern, agile projects

If the team does not make its results visible to the sponsors, the sponsors will cut the supporting flow of money and decisions. Indeed, when I look through stories of failed agile projects, this is a dominant pattern that emerges: The project team dismissed the project manager, did not create an adequate line of visibility to the sponsors, and the support got cut off a short time later. This visibility-providing role need not be in the hands of the project manager – it just usually is, because that is part of the normal job description for a project manager (PM). If an agile team wishes to work without a PM, the team must find another way to get the visibility and support they need.

The PM also serves the team is by acting as a wall against interruptions. Indeed, from the time I was a junior hardware designer in the 1970s through today, I have heard developers say, "The only thing a project manager provides me is keeping interruptions away!"

The largest part of a PM's job is "getting people to work together who would otherwise be sitting in the cafeteria arguing.[18]" This means working on communication, amicability, personal safety, and motivation.

Finally, it is the project manager who helps people identify and get the skills development they need; who helps them clear time for focusing on work; who makes sure everyone knows their priorities, that the priorities are aligned within the team and with the organization's; who organizes and may lead the reflection sessions.

Even after we drop the bureaucratic load of the PM to nil, there is still plenty for a good project manager to do. If you choose to eliminate the role of "project manager," make sure that all these critical factors are taken care of.

## "Agile development is low in discipline"

Notwithstanding the discussion in the 1st edition of this book ("Countering with Discipline and Tolerance," p. 52? and "XP and Discipline," p. ???), many people have formed the impression that agile development is low on discipline.

The best agile teams are extremely disciplined, writing test cases before they write new code, checking in and integrating running-tested code every few hours, refactoring their code to simplify it, showing their results to users every week, and reflecting on their work habits every week or every month. I find these practices difficult to introduce in most organizations. I am always surprised at how many agile groups manage to do them all, enjoy doing them, and insist on doing them.

### *Agility* and *Discipline*

The idea that agility is not disciplined comes from people working with a one-dimensional comparison line (see Figure 5'-(-2)). This view causes trouble because people are forced to place agile development somewhere *between* cowboy coding and plan-driven development.

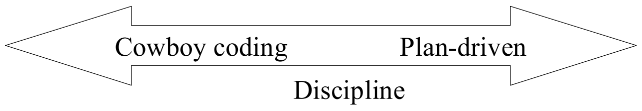


**Figure 5'-(-2).** The old view of a one-dimensional spectrum of *discipline* between cowboy coding and plan-driven.

But it doesn't sit between those two choices at all. There is another dimension in play, namely how much flux or change is happening with the requirements and the code (see Figure 5'-(-1)).

Cowboy coding (what I sometimes call "lost in the woods" development) involves changing direction a lot, as well as missing discipline. The changes in direction may or may not be useful to the company. Plan-driven development intends to slow down the changes by adding a certain level of discipline.

Agile development permits greater changes in requirements and design, by adding a certain level of discipline. There are numerous agile methodologies, some of which call for more discipline than others. XP probably calls for the greatest amount of discipline among the named agile methodologies (see "XP and Discipline, p, ???). Crystal and Scrum are fairly deliberate about lowering the requirements for discipline as low as possible – but both still require more discipline than cowboy coding. Judicious use of agile principles and techniques let the team

[18] Thanks to Carl Ellison, who spoke these mysterious words to me back around 1978. They have been rattling around my head for decades waiting for me to make sense of them!

respond faster to changing requirements and design than cowboy coding does.

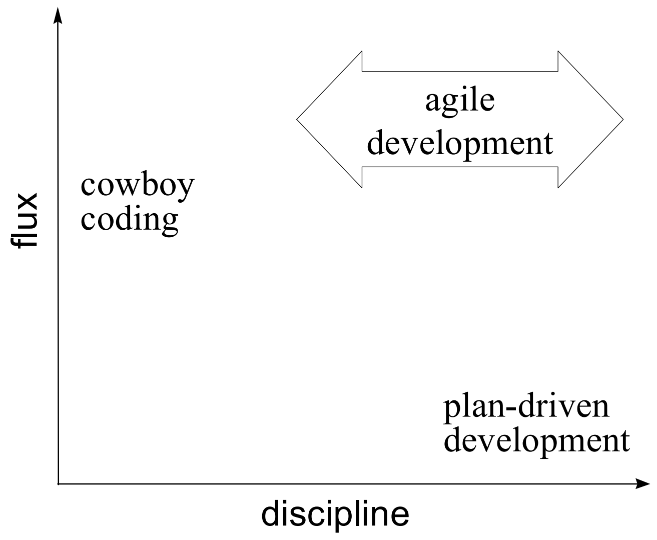


**Figure 5'-(-1).** A more accurate view shows *discipline* and *flux* as separate dimensions. Agile development occupies the high-flux zone and permits a varying amount of discipline.

Thus, agility does not go *against* discipline, but is an independent selection. One management book captures the importance of having both: In *Big Winners, Big Losers*, Alfred Marcus (2006) reports on his study of companies that stayed successful across leadership changes. He found they had all three characteristics:

- Agility
- Discipline
- Focus

*Focus*, which provides knowledge of direction and the ability to not work on things not related to that direction, simplifies people's choices. *Agility*, which provides the ability to change directions quickly, drives effective team work habits. *Discipline*, which provides the capacity to do the unpleasant, keeps the team doing what works.

Agility, discipline and focus are as essential to an effective software team as to the company as a whole.

### *Tolerance for Uncertainty*

This is the appropriate time to offer a graph that Jim Highsmith and I drew up in one of our early meetings, but which we haven't published up to now (Figure 5'-5). It relates to the organizational tolerance toward flux.

The most productive strategies operate with a high level of flux, ambiguity and uncertainty. Incremental development, iterative development, and concurrent development are three strategies in this line. Anyone who has lived through a concurrent-engineering project understands the higher levels of flux, ambiguity and uncertainty it generates.

The horizontal axis shows the level of ambiguity and uncertainty the people in the organization can tolerate. As they can tolerate more, they can apply the more productive concurrent strategies. As they can tolerate less, they are restricted to simpler, less productive strategies (waterfall, for example).

Each project requires a certain level of productivity to complete in time. If the project requires a lower level of productivity, then a team with lower tolerance for ambiguity and uncertainty can still complete the project with their simpler strategies. However, if the project requires the highest levels of productivity, the same team will fail, because the people are unable or unwilling to use the more productive strategies (that call for the extra ambiguity and uncertainty).

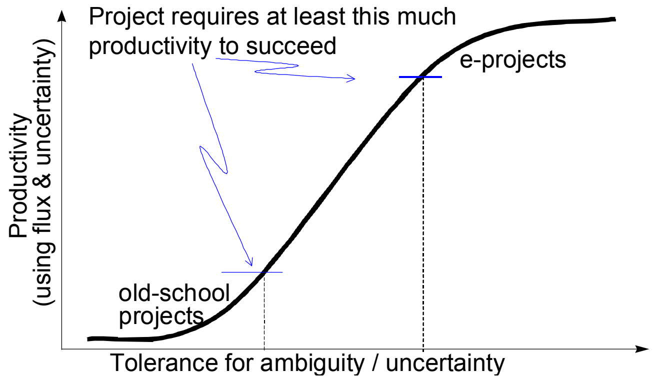


**Figure 5'-5.** Some groups cannot use highly productive strategies because those strategies require more tolerance for ambiguity and uncertainty than the groups can manage.

This is what the Boehm-Turner diagram points at with the Culture axis see Figure 5'-4, in the next section). Some organizations need order, or the fiction of it. Those organizations are less suited to trying out agile strategies, since the agile strategies will bring ambiguity and uncertainty with them. Organizations with higher tolerance will have an easier time with the agile approach.

Figure 5'-5 makes one more, very important, point. Some projects require the more advanced, more productive, uncertainty-generating strategies in order to succeed. Others don't. If your organization's projects don't require that extra level of productivity, then your organization doesn't need the added stress caused by agile methods. If, on the other hand, your project needs that level of productivity, but your organization can't tolerate the needed level of ambiguity and uncertainty, then you're in trouble.

I think that this graph hints at the limits of adoption of the agile approach

*"So what should we do?"*

Do not choose *between* agility and discipline. You can have either, or neither, or both.

## "Agile only works with the best developers"

Many writers have offered a peculiar objection to the agile approach. They observe that very bright designers articulated the principles, and jump from that observation to the assertion that only an all-expert team can succeed with the agile approach.

This objection is peculiar because the argument is so riddled with flaws that I am surprised to find it being offered at all, let alone by people I respect. I include here a discussion of this peculiar objection only because so many people cite it.

The manifesto's authors were mostly consultants at the time of the writing. As consultants, we – and here I certainly speak for myself – were never given the opportunity to work will "all expert" teams. Those are not the teams that call us for help. It is the non-expert teams who call in the consultants. Therefore, we were writing from experience in working with teams having a fairly normal distribution of expertise: maybe a good person or two, some young novices, some mediocre people. The project teams I interviewed in the early 1990s, from whom I learned the agile ideas, had those standard skill distributions. That skill mix did not change when I started consulting. The agile approaches were learned from, debugged on, and practiced with teams having the usual team skill mix.

Besides the above practical objection, there is an interesting logical flaw in the argument. It uses a rhetorical, *cognitive illusion*[19], to fool an audience into believing a false result. The cognitive illusion is that we are quickly drawn to believe the *inverse* of a proposition, when only the *contrapositive* logically follows.

Here is a quick refresher. Suppose that when Jim eats fish he gets sick.

- To form the *inverse* (which won't be true), negate both parts and leave them in the same order: "If Jim didn't eat fish last week, then Jim didn't get sick last week." On the surface, this looks reasonable, but with a bit of thinking, we realize that Jim could have gotten sick from some other cause.
- To form the *contrapositive* (which will be true), negate both pieces and *reverse* their order: "If Jim was not sick last week, then Jim didn't eat fish last week." This is true, as the following table summarizes.

| | | |
|---|---|---|
| Proposition | **A → B** | "If **A**, then **B**" |
| Inverse | **~A → ~B** | Doesn't follow |
| Contrapositive | **~B → ~A** | Follows |

[19] See *Inevitable Illusions* (???), which talks about other cognitive (as opposed to optical) illusions.

The cognitive illusion is that we find ourselves drawn to the unsound, simpler *inverse* rather than to the more complicated, sound contrapositive (the length of the word *contrapositive* doesn't help).

With that refresher, let's take a look at an assertion that I have seen at the top of more than one speaker's slides: "A problem with agile processes is that they require competent and experienced people."

- *Proposition*: "If I'm using an agile process, I need some competent and experienced people."

The simple, unsound *inverse* is:

- *Inverse*: "If I am not using an agile process, I don't need any competent and experienced people on my project ."

The implication being given is that if we use a non-agile process, we can avoid using any competent and experienced people.

This would be an amazing result if it were true! However, if we look at project histories, one of the simplest observations is that projects of all shapes and sizes rely on at least a few competent and experienced people to set it up and pull it through.

In reality, no matter what process we use, we are going to need at least a few competent and experienced people. So, the inverse proposition doesn't make sense.

Now let's look at the contrapositive:

- *Contrapositive:* "If I don't have a few competent and experienced people, I can't use agile processes."

Or anything else, for that matter! Having a certain number of competent and experienced people on a project is simply a critical success factor, no matter what the process type. This contrapositive is true but almost meaningless! And therefore so is the sentence, "To use an agile process, I need some competent and experienced people." True, but basically without meaning.

I had a project consisting only of novices, I would put my money on them doing better with an agile than a plan-driven process. I can't imagine novices coming up with a meaningful plan from the beginning, or delivering a successful project by sticking to that plan, or updating the plan every time they found a mistake in it. An agile process will call for them to work together, integrate frequently, test their code, show their results to their sponsors and users, and update their plans and working practices accordingly. With this approach they have a better chance to learn and improve within the project.

Barry Boehm and Rich Turner came up with an interesting variation of the objection. They wrote, "You need a higher ratio of competent and experienced people on an agile project than on a plan-driven or paper-centric projects."

This statement is true, as far as I can tell, through an interesting and sort of backhanded line of reasoning that also appears to be true. It goes like this:

On an agile project, there is very little extra paperwork to produce. Most of the time is spent writing code or writing tests. Poor performers stand out in this climate[20]. The project can get away with fewer people, because there is so little extra paperwork to produce.

On the other hand, there is a lot of paperwork to produce in the average plan-driven process. It is a waste to have top programming talent doing paperwork, and so it makes sense to hire lower-qualified people to do it.

Referring to them as level-1 (for beginners) and level-3 (for competent and experienced) people, we see the following.

- On an agile project, it is sensible to have a fewer level-1 to level-3 people, 3:1 to 5:1 are ratios I have seen work well.
- On a plan-driven project, it is advantageous to have more level-1 people, so perhaps between 5:1 to 8:1 could make sense.

Barry Boehm and Rich Turner capture this line of thinking in their book, *Balancing Agility with Discipline*[21] (Boehm 2003). Recaptured in Figure 5'-4. they show a starfish diagram with five different characteristics of a project situation:

- ° the number of people being coordinated
- ° the criticality of the project
- ° the ratio of level-1 to level-3 peoplethe rate of requirements change
- ° the tolerance of the organization to change (see also Figure 5'-5).

This starfish diagram looks correct to me, and I commend it to others to study.

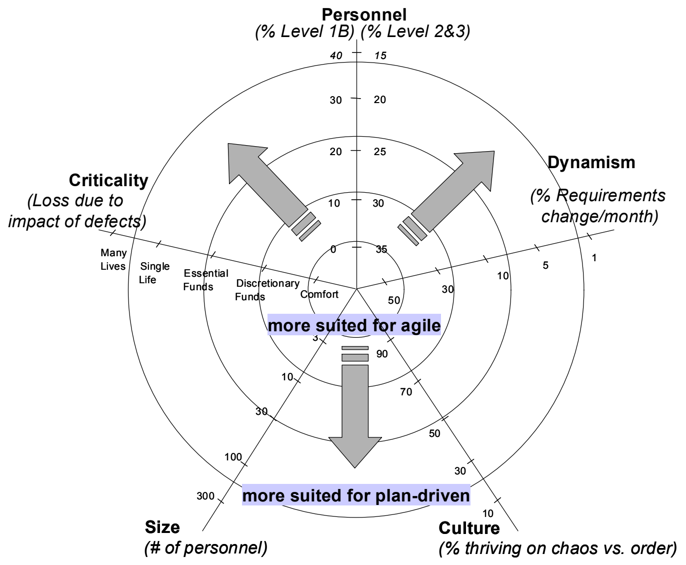


**Figure 5'-4.** Starfish diagram showing the shifting home territories of agile and plan-driven development, from (Boehm 2003).

With respect to the question of level-1 to level-3 people, we might note an implicit assumption in that axis of the diagram, that the organization *is obliged to use all those level-1 people*. That is, someone obliges the project managers to find ways to keep all those people busy. If that were not the case, then I would suggest either of two strategies:

- Get rid of the extra level-1 people and use an agile process with the reduced team size.
- Use an agile process with the smallest number of people possible to get the job done; let the rest of the people do *anything they want* just so long as they don't interfere with the progress of the development team.

Odd though this latter strategy sounds, I have met with two project teams that did just that, and both recommended the strategy highly. They said that payroll costs were fixed with respect to the number of people being paid, but their speed was higher without the level-1 people slowing them down. This unusual strategy illustrates good sense in playing the cooperative game, though perhaps a perverse attitude toward the organization's personnel department.

*"So what should we do?"*

Get some competent, experienced people. Work with agility and focus in all cases. Improve the team's speed by adding disciplined agile practices as the team can or is willing.

## "Agile is new, old, failed and untried"

It is common to object to an idea by saying that it is "not new" – meaning that it had been tried and failed – and si-

[20] One of the objections to the agile approach in some organizations is exactly that the low performers get revealed so quickly. If those people have political clout, this can be enough for the organization to reject an agile process. Sadly, as humorist Dave Barry writes, "I am not making this up."

[21] It is an odd title. Agility and discipline are not opposites to be traded against each other – see "Agile development is low in discipline " on page 34???. Within the book, they shift to the term *plan-driven*, which is better but still not my preference – see "Predictable, Plan-driven and Other Centerings" on page 55???. Those terms aside, they give an outstanding presentation of the tension between planning and moving, and the need for use of the large middle ground.

multaneously that it is new – meaning that we don't know if it works. When I hear these objections, I think that Tom Waits has the only correct answer:

"It's new, it's improved, it's old-fashioned"
--Tom Waits, "Step Right Up"

Every idea in the agile canon has been traced as far back as the 1960s and even the 1950s[22]. We find incremental development, collocation, pair programming, frequent integration, test-first development, customer interaction and so on.

As far as I can tell, good developers have long been using these ideas, but weren't articulate about what they were doing. I ran across one personal account (reference lost, sorry) in which the writer said that he was so struck by the silliness of the waterfall style-methodologies being published in the 1970s that he didn't see the need to respond. He assumed they would fall into oblivion from their own weight, and was shocked when people continued to reference them seriously.

My theory is that agile approaches have always been used, but did not show a market advantage until the 1990s, when business and technology were changing so fast that only the agile practitioners were delivering systems in time to the market. Certainly, that was the case in the early 1990s, when I was just starting to debrief projects for the IBM Consulting Group in order to construct a methodology for object-technology projects. In those initial interviews, there was a very clear dividing line between those project teams who were delivering projects and those who weren't. Those who were delivering were following the values and principles expressed almost ten years later in the agile manifesto.

A better way to think of the agile manifesto is that the manifesto authors "cherry picked" a very small set to pay close attention to out of hundreds of possibly important things,. We were asserting, in essence, that these few items outweighed the rest.

It is natural that the items we chose had been around for a long time. What was new was that we bothered to name a particular and critical combination, and there were certain new wrinkles on the old practices to offer.

Agile is new, it's improved, it's old-fashioned.

[22] See, for example, Craig Larman's *Agile and Iterative Development* (Larman 2003???), and Fred Brooks description of "growing software" in his "No Silver Bullet" article (Brooks 1999???)

## Evolution of the Agile Methodologies

The people who wrote the agile manifesto think and work in different ways, all at the *Ri* end of the scale (see the *Shu-Ha-Ri* scale of skills development, pages ???). Writing the manifesto, they were looking to find what was common in their approaches and still preserve their right to differ. The agile manifesto successfully locates commonality across widely varying, strong-minded people, and does so in such a way that other *Ri*-level people can find their own home and be effective.

That has been, and continues to be good. What its entails is that there is no agreed-upon *Shu* form of agile development for beginners to follow.

Indeed, as the online conversation between John Rusk and Ilja Preuß[23] below points out, not specifying mandatory *Shu*-level practices is core to the agile approach. Any specific set of *Shu* practices would be fragile over time and situations.

> *John Rusk:* As Alistair mentioned, the Manifesto is *the* non-branded resource. But, personally, because it emphasizes values rather than practices, I did not find it to be an ideal starting point for understanding Agile. (Beginners want details, even if they can't handle them!) The Manifesto made much more sense to me *after* I'd learnt about several of the branded processes.
>
> *Ilja Preuß:* I think there is a dilemma here - to me, not following some cookbook **is** at the heart of Agility.

And there's the dilemma: People just starting out need a *Shu*-level starter kit. The agile experts want finally not to be trapped by simplistic formulae. It's Russ Rufer's quip again: "We can't make one *Shu* fit all."

Looking at the evolution of the agile methodologies, we find a growing aversion to simple formulae and increasing attention to situationally specific strategies (not accidentally one of the six principles in the Declaration of Inter-Dependence). Since most of the branded agile methodologies are formulaic by nature, we find a growing use of unnamed methodologies, or even non-methodology methodologies (a contradiction in terms, I know, but it reflects people's desire to break out of fixed formulae).

Let's take a look at the methodology evolution since 2001, changes in testing, the modern agile toolbox, and a long-overdue recognition of the specialty formerly known as user-interface design. The evolution of the *Crystal* family of methodologies gets its own chapter.

[23] http://groups.yahoo.com/group/crystalclear/message/586. Thanks to John and Ilja for letting me reproduce it here.

### XP Second edition

The first edition of Kent Beck's "Extreme Programming: Embrace Change" (Beck 1999?) changed the industry. His second edition, produced just five years later (Beck 2004), rocked the XP community a second time, but for a different reason – he reversed the strong position of the first edition, making the practices optional, independent and variable.

In the first edition, he wrote:

> "XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements."

The book contains a *Shu*-level description of XP. Kent was saying, "Here is a useful bundle of practices. Do these things and good things will happen." Beginners need a *Shu*-level starter kit, and the first edition served this purpose.

Over time, it became clear that project teams needed different combinations of XP and things similar to, but not exactly, XP. Also, it turned out that many of the practices stand nicely on their own (important note: some practices *do not* stand on their own, but depend on other practices: aggressive refactoring, for example, relies on comprehensive and automated unit tests.)

Note the difference in writing in the second edition:

> "XP is my attempt to reconcile humanity in my own practice of software development and to share that reconciliation."

The book contains two big shifts. First, he rejects the idea that XP is something that you can be *in compliance* with. It doesn't make sense to check, "Are you in compliance with Kent's attempt to reconcile humanity in his own practice of software development and sharing that reconciliation?" He correctly notes in the book that people who are playing in the extreme way will recognize each other. It will be evident to the people in the club who is in the club.

Secondly and more significantly, the second edition is written at the *Ri* level. It contains a number of important things to consider, with the advice that every situation is different and the practitioner will have to decide how much of each idea, in what order, to apply to each situation. As predicted by the *Shu-Ha-Ri* model and the dilemma captured by Ilja Preuß, this causes discomfort in people looking for a *Shu*-level description.

Beginners can still benefit from the first edition of the book by using it as their starter kit; as they grow in sophistication, they can buy the second book and look for ways to make *ad hoc* adjustments.

## Scrum

Scrum provides a *Shu*-level description for a very small set of practices and a *Ri*-level avoidance of what to do in specific situations. Scrum tells people to think for themselves.

Scrum can be summarized (but not executed) very simply[24]:

- The team and the project sponsors create a prioritized list of all the things the team needs to do. This can be a list of tasks or a list of features. This is known as the *product backlog*.
- Each month, the team pulls off the top section of the list, what they estimate to be one month's worth of work. They expand it to a detailed task list, called the *sprint backlog*. The team promises to demo or deliver the results to the sponsors at the end of the month.
- Each day, the team meets face-to-face for five to ten minutes to update each other on their status and what roadblocks are slowing them down. This is called the *daily stand-up meeting.*
- One particular person is designated the *scrum master*. It is this person's assignment to remove, or get someone to remove, whatever roadblocks the team mentions in the stand-up meeting.

Scrum avoids saying *how* the team should develop their software, but is adamant that the people act as mature adults in owning their work and taking care of problems. In this sense, it is an essential distillation of the agile principles.

The importance of the first three elements of Scrum listed above should be clear from the theory of software development described all through this book. The fourth point deserves some further attention.

Scrum calls for a staff role, the scrum master, whose purpose in life is to remove obstacles. Scrum is the only methodology I know that trains its practitioners in the idea that success comes from removing obstacles and giving the people what they need to get their work done.

This idea was illustrated in the section on the Cone of Silence strategy. There, I described how a project went from hopelessly behind schedule to ahead of schedule from the single act of moving the lead developer upstairs and around a few corners from the rest of the team. It was illustrated again in another conversation about another project. A manager came up to me once to tell me how well Scrum was working for them.

[24] See Ken Schwaber's two books (Schwaber 2002, Schwaber2004).

"Our project went from four months at initial estimation to delivering in just six weeks," he glowed at me.

Since I know that Scrum contains nothing specific that would cause that sort of a jump in productivity, I challenged him to tell me what happened in more detail, and what caused the amazing change in fortunes.

He said that at the daily stand-up meetings, the key programmer kept saying he was being pulled away several times a day to fix bugs in other systems.

The manager went to an executive high enough up the management chain to be able to control both projects' resources. He challenged the executive, saying that he had been told this particular project had high priority, and the programmer shouldn't be pulled out like this. The executive agreed.

The interruptions stopped. The key programmer, freed from interruptions, sat down and completed the program in six weeks.

It seems the four-month estimation was based on his expectation of a certain level of interruptions. Without interruptions, he got done much faster (see the project critical success factor, "Focus", on p. 42???).

Remember, success is not generated by methodology magic nearly so much as by giving the people what they need to get their work done.

## Pragmatic and Anonymous

Andy Hunt and Dave Thomas, co-authors of the agile manifesto and authors of *The Pragmatic Programmer* (Hunt 2003???), have been defending that it is not process but professional and "pragmatic" thinking that are at the heart of success. They are defending situationally specific, anonymous methodologies, properly referred to as *ad hoc*[25].

In the five years since the writing of the manifesto, situationally specific methodologies have become more the norm than the exception. A growing number of teams look at the rules of XP, Scrum, Crystal, FDD and DSDM and decide that each one in some fashion doesn't fit their situation. They blend them in their own way.

The two questions that those teams get asked are: is their blend agile, and is their blend effective? A bit of thought reveals that the first question is not really meaningful, that it is only the second question that matters. In

[25] Here I am using *ad hoc* with its proper Latin meaning: "for the situation." *Ad hoc* is not supposed to mean "not thought out." We see the importance of *ad hoc* strategies throughout the book, and the Declaration of Inter-Dependence explicitly calls for situationally-specific strategies.

many cases, the answer is, "more effective than what they were doing before."

The alert reader is likely to ask at this point, What is the difference between an anonymous, *ad hoc* methodology and Crystal, since Crystal says to tune the methodology to the project and the team? This question leads to the question: What is the point of following a named methodology at all?

A named methodology is important because it is the methodology author's publicly proclaimed, considered effort to identify a set of practices or conventions that work well together, and when done *together*, increase the likelihood of success.

Thus, in XP first edition, Kent Beck did *not* say, "Pair programming is a good thing; do pair programming and your project will have a good outcome." He said, "Pair programming is part of a useful package; this package, when taken together, will produce a good outcome." Part of the distress people experienced with the second edition was that he took away his stamp of approval from any particular packaging of the practices, leaving the construction of a particular mix up to the team (in this sense, XP second edition becomes an *ad hoc* methodology). What got lost was any assurance about the effects of adding and dropping individual practices.

Crystal is my considered effort to construct a package of conventions that raises the odds of success, and at the same time the odds of being practiced over time. Even though Crystal calls for tailoring, I put bounds on the tailoring, identifying the places where I feel the likelihood of success drops off suddenly.

When I look at people working with conventions similar to Crystal's, the first thing I usually notice is that they don't reflect on their work habits very much. Without reflection, they lose the opportunity to improve. (They usually don't deliver the system very often and they usually don't have very good tests, but those can both be addressed if they reflect seriously.)

In other words, there is a difference between a generic *ad hoc* methodology and a tailoring of Crystal. I feel there is enough significance to naming the key issues to attend to, and the drop-offs, to make the Crystal package worth retaining as a named package. I include details of what I have learned about the Crystal methodologies in Chapter 9?.

If you want to know if your tailoring is likely to be effective, read the list of practices in "Sweet spots and the drop-off" (p. ???) and in the table shown in "How agile are you?" (p. ???), and compare where your practices are with respect to the sweet spots and the drop-offs.

The above line of thinking suggests that other authors should come up with their own recommendations for specific sets of conventions or practices that improve some aspect of a team's work.

One such set that I find interesting, but has not yet been published, was described by Lowell Lindstrom of Oobaya Consulting. Lowell runs an end-of-iteration reflection workshop that is more thorough than mine, and which I intend to start using. His contains three specific parts, covering product, progress and process. He creates a flipchart for each and asks the team to reflect on each in turn:

- *Reflect on the Product:* What did the users and sponsors think about the product? What should be improved, what added, what removed, and what are the priority directions for product evolution? This reflection topic addresses the team's direction and priorities, key for everything that follows.
- *Reflect on the Team's Progress:* What did the team members say they would accomplish during the iteration, and what did they accomplish? What does that difference mean? What do they want to promise for the next iteration? This reflection topic is subject to changes in direction from the first reflection question.
- *Reflect on the Team's Process:* What worked well in the working conventions from the last cycle, and what does the team want to try differently? This is the Crystal reflection technique (see p. ???) and in *Crystal Clear* (Cockburn 2005 CC)). Note that what the team chooses to change depends on the changes identified in both product and progress.

Lindstrom's reflection package is a good addition to any methodology. Combined with Scrum, it makes a good starter kit for teams looking to get out of the *Shu* stage and into the *Ha* stage.

## Predictable, Plan-driven and Other Centerings

The term "plan-driven" was coined by Barry Boehm to contrast to the term "agile". To understand the term and its implications, we need to look at how these sorts of terms center a team's attention.

"Agility" is only a declaration of priorities for the development team. It may not be what the organization needs. The organization may prefer predictability, accountability, defect-reduction or an atmosphere of fun[26]. Each of those priorities drives a different behavior in the development team, and none of them is a wrong priority in itself.

[26] At one university working on the human genome project, the students were paid so little that the department only hoped that they wouldn't just quit. The students could propose any problem they wanted to work on, choose any technology they wanted to use, set any work hours and methods that suited them. They felt that having a "laid back" methodology was crucial to being able to retain the programmers they needed.

Further, no methodology is in itself agile (or any of the other priorities). The implementation is what is or is not agile. Thus, a group may declare that they wish to be agile, or wish to be predictable, or defect-free, or laid back. Only time will tell if they actually are. Thus, any methodology is at best "would-be agile", "would-be predictable" and so on[27].

There is, therefore, no opposite to *agile* development, any more than there is an opposite to *jumping* animals (imagine being a biologist deciding to specialize in "non-jumping animals "). Saying "non-agile development" only means having a top priority other than agility. But what would the priority be for all the project styles that the agile manifesto was written to counteract? No one has been able to get all those people in a room to decide where their center of attention has been.

Actually, agile is almost the wrong word. It describes the mental centering needed for these projects, but doesn't have a near-opposite comparison term. *Adaptive* does. I don't wish to suggest that *agile* should be renamed, and I am careful to give a different meaning to adaptive methodologies compared to agile methodologies:

For me, and for some of the people at the writing of the agile manifesto, *agile* meant being responsive to shifting system requirements and technology, while *adaptive* meant being responsive to shifting cultural rules. We were discussing primarily the former at the manifesto gathering.

Thus, XP first edition was agile and not very adaptive, RUP was adaptive and not very agile, and Crystal and XP second edition are both agile and adaptive. In the last five years, being adaptive has been seen as so valuable that most people now consider being adaptive as part of being agile.

Using the word *adaptive* for a moment allows us to look at its near-opposite comparison term: *predictable*.

In my view, the purpose of *that other* way of working was to provide *predictability* in software development[28].

We *can* sensibly compare processes aimed at increasing predictability with those aimed at increasing adaptiveness to changing requirements, and we can see what we might want from each. However, although some of us might have preferred the word 'predictable', Barry Boehm chose 'plan-driven' for those processes.

I earlier ("Agile only works with the best developers" on p. 55???) described some the work that Barry Boehm and Rich Turner have been doing with partially agile and partially plan-driven projects (Boehm 2002???, Boehm 2003???). Figure 5'-6 shows another.

Figure 5'-6 shows that for each project, there is a certain type and amount of damage that accrues from not doing enough planning (having a space station explode is a good example). There is also a certain type and amount of damage that accrues from doing too much planning (losing market share during the browser wars of the 1990s is a good example).

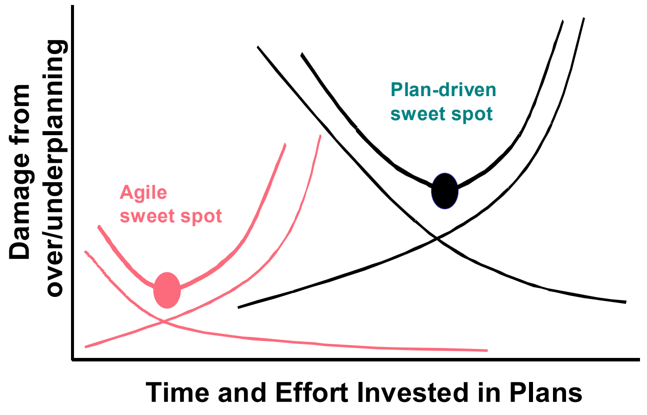


**Figure 5'-6**. Different projects call for different amounts of planning (after Boehm 2002, 2003)

Figure 5'-6 shows that the agile approach has a sweet spot in situations in which overplanning has a high damage and underplanning doesn't. The plan-driven approach has a sweet spot when the reverse is true. The figure hints that there are situations in the middle that call for a blending of priorities. This matches the recommendations in this book and the Declaration of Inter-Dependence.

In the time since the writing of the agile manifesto, people have been working to understand the blending of the two. Watch over time as more plan-driven organizations find ways to blend the agile ideas into their process to get some of its benefits.

## Theory of Constraints

Goldratt's "theory of constraints" has been being examined among agile developers. Three threads from Goldratt's writing deserve mention here.

### *The importance of the bottleneck station*

In the book *The Goal* (Goldratt 1985), Goldratt describes a manufacturing situation in which the performance of a factory is limited by one station. He shows that improving efficiency at the other stations is a waste of money while that one is still the bottleneck. He ends with the point that as the team improves that one station, sooner or later it stops being the bottleneck, and at that moment, improving it any further is a waste of money.

[27] For more on this, see "Agile Joins the 'Would-Be Crowd," Cockburn would be 2003???

[28] There is even dissention about how to provide predictability. Many agilists claim that the agile approach confers greater predictability. The difference in views is discussed in detail in the section on "Buy Information or Flexibility" in "Learning from Agile Development" (Cockburn 2002.10 crosstalk).

I have found a fruitful line of inquiry to consider that one can "spend" efficiency at the other, *non-bottleneck* stations to improve overall system output. This surprising corollary is described in more detail on pages ??? and ??? of this book, and in (Cockburn ICAM 2005???).

Few managers bother to understand where their bottleneck really is in the first place, so they are taking advantage of neither the main result nor the corollary.

### *The theory of constraints (TOC)*

The TOC (goldratt 19??) is very general. It says:

- Locate whatever is the constraint or bottleneck.
- Target all efforts at improving the efficiency at the constraint point.
  - Maximize the utilization of the available capacity in the constraint;
  - Don't make it do work that won't produce throughput[29];
  - Don't have it idle for any other reason;
  - Invest in and increase the capacity of the constraint.
- Improve that area so that it is no longer the constraint. At that moment, the constraint is somewhere else, and the cycle of work starts over.

It is not so much the simple statement of the TOC that is interesting, but all the special solutions being catalogued for different situations. A web search for "theory of constraints" and its conferences will turn up more useful references than I can provide here. David Anderson presented a case study of applying the theory of constraints to software development in "From Worst to Best in 9 Months" (Anderson 2005).

### *Critical chain project management.*

Goldratt's "critical chain" (Goldratt 2001 c-chain) is more controversial. "Controversial" in this case means that I think it is hazardous, even though it has backing in the TOC community. Here is why I worry about it.

The critical chain idea says to start by making a task-dependency network that includes also names of people who will become constraints themselves.

For example, in most software projects, the team lead usually gets assigned the most difficult jobs, but also has to sit in the most meetings, and also does the most training. A normal PERT chart does not show this added load, and so it is easy to create a schedule in which this person needs to be in multiple places at once. Putting the team lead's name on the tasks, and putting that person's tasks into a linear sequence produces a truer schedule (and an unacceptably long one, usually!). One can quickly see that the schedule improves as the load gets removed from the team lead, even when the tasks are assigned to people less qualified. So far, this is a simple application of TOC.

Second (and here is the controversial part), ask the people on the team to estimate the length of each task, then cut each estimate into two halves. Leave one half in place as the new estimate of the task duration, cut the other half in half and put that quarter into a single slippage buffer at the end of the project. Discard the remaining quarter as "time saved" from the people's original estimate.

The thinking is that if people are given the shortened length of time to do their work, they will sometimes meet that target and sometimes not, in a statistical fashion. The buffer at the end serves to catch the times they are not. Since not every task will need its full duration, the end buffer can be half of the original.

The hope is that people tend to pad their estimates to show when they can be "safely" done. Having once padded the estimate, they have no incentive to get the work done early, and so will take up more time than they strictly need to get the task done. By cutting the estimate in half, they have incentive to go as fast as they can. By keeping a quarter of the time estimate in the slippage buffer, the project has protection against a normal number of things going wrong.

Where I find the above thinking hazardous is that it touches upon a common human weakness, being very easy to abuse from the management side. Indeed, in the only two projects that I found to interview about their use of this technique, they both fell into the trap.

The trap is that managers are used to treating the schedule as a promise not to be exceeded. But in critical chain planning, the team members are not supposed to be penalized for exceeding the cut-in-half schedule – after all, it got cut in half from what they had estimated, on the basis that about half the tasks will exceed the schedule. Critical chain relies on having enlightened managers who understand this and don't penalize the workers for exceeding the halved time estimates.

My suspicion is that if the people know that their estimates will get cut in half, and they will get penalized for taking longer than that half, then they will double the estimate before turning it in, removing any benefit that the buffer planning method might have offered. If they don't double their estimates, there is a good chance that they will, in fact, get penalized for taking longer than the cut-in-half estimate.

That is exactly what I found in the first critical chain interview. The project manager on this fixed-time, fixed-

[29] Paul Oldfield notes the interesting contradiction between this point and the need for workers to "sharpen the saw": upgrading their skills, recuperating, and reflecting on their work practices. I note that practitioners of the lean approach take "sharpening the saw" very seriously.

price contract had done the PERT chart, cut the estimates in half and created the slippage buffers as required, and managed the work on the critical path very closely. He was very proud of their result *–they had never had to dip into their slippage buffer at all!*

I hope you can see the problem here. He had gotten his developers to cut their estimates in half, and then had ridden them through weekends and overtime so that they would meet all of their cut-in-half estimates without touching the slippage buffer. He was proud of not dipping into the buffers, even though critical chain theory says that he should have dipped into those buffers about half the time. Missing the point of the method, he and his developers had all suffered.

The company executives were delighted at delivering a fixed-price contract in 3/4 of the time estimated. After all, they turned an unexpected extra 25% profit on that contract. The sad part was that they gave no bonuses for the overtime put in by the project manager and the team. This is what I mean by critical chain touching upon common human weaknesses. I suspect the developers will double their estimates for the next projects.

If you are going to use the critical chain technique, be sure to track work time in hours, not days or weeks. Tracking in days and weeks hides overtime work. Tracking in hours keeps the time accounting correct. If the manager in the previous story had scheduled and tracked in hours, he would have had to use those buffers as critical chain theory says he should.

The above doesn't mean that critical chain doesn't work, only that it easy to misuse. I suspect it only works properly in a well-behaved, high-trust environment, and will cease to work as soon as it starts to be misused, and people game the system.

## Lean Development

Agile developers have been studying the lessons from lean manufacturing to understand strategies in sequencing and stacking decisions. This was discussed in "Reconstructing Software Engineering" (p. ???).

As a reminder, *inventory* in manufacturing matches *unvalidated decisions* in product design. When the customer requests a feature, we don't actually know that it is a useful feature. When the architect or senior designer creates a design decision, we don't actually know that it is a correct or useful design decision. When the programmer writes some code, we don't know that it is correct or even useful code.

The decisions stack up between the user and the business analyst, the business analyst and designer-programmer, the designer-programmer and the tester, and the tester and the deployment person (see Figures 2-2 and 2-3). In each case, the more decisions are stacked up the greater the uncertainty in delivery of the system.

Kent Beck drew a graph (Figure 5'-7) that shows the way that decisions stack up as the delivery interval gets longer. He drew it on a log scale to capture the time scales involved. To understand it, imagine a project in which a request shows up in the morning, gets designed, programmed, tested within 24 hours. Let this be the unit measure of decision delay.

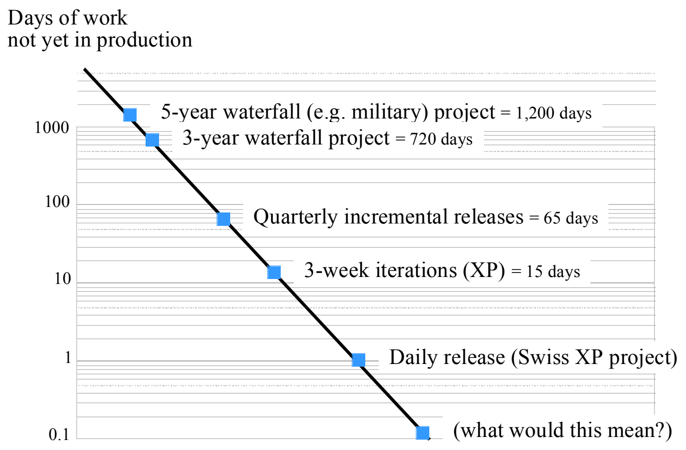


**Figure 5'-7.** Decision "inventory" shown on software projects of different types (thanks to Kent Beck).

The problem, as his drawing captures, is that the sponsors of a five-year waterfall project are paying for thousands of times more decisions and not getting validation or recovering their costs on those decisions, for five years. The appropriateness (which is to say, the quality) of those decisions decays over time, as business and technology change. This means that the value of the investment decays. The sponsors are holding the inventory costs, losing return on investment (ROI).

As the delay shrinks, from request to delivery, ROI increases because inventory costs drop, the decisions are validated (or invalidated!) sooner, and the organization can start getting business value from the decisions sooner[30]. Kent's graph shows how quickly the queue size increases as the delivery cycle lengthens.

One of the lessons from lean manufacturing is to shrink the queue size between work stations. The target is to have single-piece, or *continuous flow*. It is no accident that the first principle in the Declaration of Inter-Dependence is

> *"We increase return on investment by making continuous flow of value our focus."*

Figure 5'-7 also shows the relationship between the queue size and delay. In software development, we can't easily see the size of the decision queue. However, the size of

[30] This line of reasoning is spelled out in great detail and in ways accounting managers can understand in *Software by Numbers* (???ref???).

the queue is proportional to the delay, and we can measure the delay.

Figure 5'-8 shows how queue size and its analog, delay, can be used in managing a software project. It plots the work completed by each specialist. In this sort of a graph, a *cumulative flow graph*, the queue size corresponds to vertical size in any shaded area, and the delay to the horizontal size in the same shaded area. If you don't know how to measure the number of items in your queue, measure the length of time that decisions stay in your queues.

**Figure 5'-8.** Cumulative flow graph for a software project used to track pipeline delay and queue size (Anderson borcon). The original caption was "CFD showing lead time fall as a result of reduced WIP" [WIP = work in progress]

David Anderson describes these graphs in his book and experience reports (Anderson 2003???, Anderson 2004). He and others have built software development tracking tools that show the number of features in requirements analysis, in design, and in test, so that the team can see what their queue size and delays are. David writes that as of 2006, "managing with cumulative flow diagrams is the default management technique in Microsoft MSF methodology shipped with Visual Studio Team System and the cumulative flow diagram is a standard out of the box report with Visual Studio Team System."

Queue sizes are what most people associate with lean manufacturing, but there are two more key lessons to extract. Actually, there are certainly more lessons to learn. Read *The Machine that Changed the World* (sss yyy???) and *Toyota Production System* (xxx yyy???) to start your own investigation. Here are the two I have selected:

*Cross-training*. Staff at lean manufacturing lines are cross-trained at adjacent stations. That way, when inventory grows at a handoff point, both upstream and downstream workers can pitch in to handle that local bump in queue size. Agile development teams also try to cross-train their developers and testers, or at least sit them together.

One of the stranger obstacles to using agile software development techniques is the personnel department. In more than one organization that I have visited, personnel regulations prevented a business analyst from doing any coding or testing. That means that the business analysts are not allowed to prototype their user interfaces using Macromedia's *Dreamweaver* product, because Dreamweaver generates code from the prototyped UI, nor are they allowed to specify their business rules using FIT (Ward Cunningham's *Framework for Integrated Tests*), because that counted as writing test cases.

It is clear to the employees in these organizations that the rules slow down the company. However, even with the best of will, they don't have the politcal power to change the rules (thus illustrating Jim Highsmith's corollary to the agile catch-phrase, "People trump process." Jim follows that with: "Politics trump people").

*"We're all* us*, including customers and suppliers"*. The agile manifesto points to "customer collaboration." Lean organizations extend the "us" group to include the supplier chain and the end purchaser.

One (unfortunately, only one) agile project team told me that they when they hired an external contract supplier to write subsystem software for them, they wrote the acceptance tests as part of the requirements package (incrementally, of course). It saved the main project team energy. They had extremely low defect rates, with an average time to repair a field defect less than one day. They suspected that the subcontractor's testing procedures wouldn't be up to their standards. By taking on themselves the burden of creating the tests they needed, the subcontractor wouldn't waste their time shipping them buggy code, the defects in which they would have to first detect and then argue over with the subcontractors. This alone made it worth their time to write the tests themselves. Of course, it simplified the life of the subcontractors, since they didn't have to write the acceptance tests.

Their story is a good start, but only hints at what can happen when the entire supply chain is "us".

## New Methodology Topics

This section is for discussing the evolution of project management, testing, user-interface design, project governance, and requirements gathering.

## Agile project management

I earlier discussed the myth, "*We don't need no *%@! managers!*" (p. 50???). Even with a self-moderating team, there is still a role for someone to monitor and enhance focus, community, amicability, personal safety and communications within the team; to give the project good visibility to the sponsors and to secure directional decisions and funding from them; and to keep distractions away from the team (see Figure 5'-3???).

However, there is still a kernel of truth in the idea that the project manager's role is shifted, if not reduced, in agile development. Good agile teams develop their plans jointly, present their status visibly and publicly, and in general have a better understanding of themselves as a mutually dependent community.

Since the writing of the manifesto, some excellent books have appeared on project management in the agile context. These are all excellent books:

- *Agile Project Management* (Highsmith 2004),
- *Managing Agile Projects* (Augustine 2005), and
- *eXtreme Project Management* (DeCarlo 2004)

all highlight the human aspects of managing.

- *Agile Management for Software Engineering* (Anderson 2003) incorporates lessons from the theory of constraints (see page 62???).
- *Agile Project Management with Scrum* (Schwaber 2004) covers the Scrum view.
- Agile and extreme project management have even been entered as specific categories starting with the 2003 third edition of Wysocki's *Effective Project Management: Traditional, Adaptive, Extreme.*
- From outside the circle of agile and software project managers, Laufer's (1997) *Simultaneous Management* is based on work primarily on civil engineering projects, but with the same thoughts, recommendations and strategies as those coming from the agile world.

A number of people felt that the discussion of how to lead agile projects was being missed in the fervor surrounding the agile manifesto. This included Jim Highsmith and me as agile manifesto authors, plus over a dozen other experts in product development, project management, and line management. Over a six-month period, we worked on a follow-on to the agile manifesto that would carry over to non-software projects, and also to product (as opposed to project) management.

The result was published in January of 2005, and called the "Declaration of Inter-Dependence[31]" or DOI.

The DOI was written to fit self-managed teams as well as hierarchically managed teams. It is aimed at development teams outside and inside software development, and for product development as well as project management. Anyone reading it will immediately detect that it provides good guidelines for line management as well as project management. It contains the things we expect from an agile mind-set: paying attention to workers as human beings, using short feedback loops and situationally specific strategies.

The DOI is described in detail in Appendix C. Extended support and discussion surrounding the DOI is provided by a non-profit group called the Agile Project Leadership Network[32] (APLN).

## Testing

The newest topic in agile methodologies is the growth of automated testing and test-driven development as core practices. Here again, XP has been the leader.

Five years ago I felt that automated testing was not a critical project success factor. However, it becomes more critical as project teams shorten their iterations, work incrementally and refactor more often.

Here is an example: A person in one of my classes asked me how to get started with agile development. I suggested that he pick any project he was working on, and shift to a one-week iteration and delivery cycle just for one or two weeks. In each week, just for the trial run, he should take a new requirement, design and program it, test and integrate it, and deploy it to some user workstation.

The person said he was immediately stuck. On his project, the testing would take two days alone, so he would have only three days to do requirements, design and coding. He couldn't do a one-week iteration. It was at that point that I recognized that automated testing had become critical to agile practices.

In project visits, I find people having trouble with developing in increments, with integrating with other people's code, and with refactoring, issues that would not be problems if they had automated test suites:.

To make the creation of tests more palatable and accessible to users, customers and business analysts, Ward Cunningham created the Framework for Integrated Testing (FIT)[33]. It allows people to specify input and outputs in a spreadsheet or HTML file, and to hook those quite simply to the application under development. Using FIT or its relative, FITnesse, business analysts can write fewer

[31] http://pmdoi.org
[32] http://apln.org
[33] See http://fit.org, http://fitnesse.org

specification *documents*, replacing those with user-readable specification *tests*. This shift supports the agile manifesto value of running software over documents (see "Hexagonal Architecture Explained" (Cockburn 2004z/??) for a more detailed description of creating a system architecture that accommodates FIT).

Test-driven development (TDD) started as a way just to make sure that developers wrote unit tests at all. Its practitioners found that when they thought carefully about the tests they were writing one at a time, and how to make each test pass just after it was written, they came up with different designs than they expected. They were simpler, cleaner, and easier to change. TDD has become one of the top recommendations for developing new code.

In a bit of irony, TDD coaches find they have to fight against the "test" implications of the term – they write tests to help them design, but resistant developers only see the reference to testing. A number of alternative acronyms are being suggested to reduce the association with testing. My favorite is XXD (eXecutable eXample), pronounced "Dos Equis driven development"[34].

## User-Experience Design

Top designers of the externally visible side of a software system no longer consider themselves designers of merely the "user interface", but of the "user experience." In keeping with their choice of words, I will refer here to user experience (UX) design rather than user interface (UI) design[35].

UX designers have largely been resisting the fine-grained incremental development policies of the agile developers. Many say they need to collect most of their information before they can design a coherent user experience or interface. They say that if they are forced to develop the UX piecemeal, the result will be clumsy.

There is much to be said for this position. However, many projects don't have a timeline that supports gathering so much information before starting development.

A second group of UX designers argue that the low-precision UX model can be created in a matter of days, not months, and so incremental development can be applied. Jeff Patton argues for this view, and has published techniques to support rapid UX design with incremental development (Patton 2004, Patton 2005a, Patton 2005b). Holzblatt and XXX have created a "rapid" form of their contextual design technique to support the same development style (Holzblatt 2003?).

The UX question is complicated because the field has not sorted out how much needs to be done in advance and what can be done incrementally, on the fly. It is clear that changing the user interface with each deployment causes users discomfort. That argues for at least some up-front design. It some cases, people have shown that a broad, low-precision design suffices and some screen details can be created incrementally over time.

We can use the cooperative game model and lean manufacturing principles to help understand what is going on here and how to create a strategy. Here are the issues at play:

- Users will get upset if the design changes too often. That speaks for creating a broad user model early, even if the details are worked out incrementally.
- There is a dependency between the UX designers and the programmers. The programmers can't program a function until the UX designers understand what the function should be. Plus, there is a time lag from when the UX designers get their information and when they make their recommendations. If they have a broad user base and show their designs to the users several times to refine it, then this time lag can be quite large, often on the order of a month.

Neither of the two extremes is likely to be a good strategy. In some organizations the UX team drives the timeline, and insist on getting their design "correct and complete" before interfacing with the programmers. This hurts them in two ways. First, the project is so delayed that quite likely the user requests will have changed before the system is developed (it *was* a correct UX design, but no longer *is* a correct one). Second, the programmers are likely to raise valid issues that cause changes to the UX design.

In some organizations, would-be agile programmers drive the timeline, and insist that the UX design be done incrementally in two-week iterations, simultaneously with programming. The UX designers complain that there is not time to research their users, create a design and program it within the two-week window. Constantly encountering new user situations, they have trouble creating a consistent user model.

Balance the need for lead time with the timing of the incremental development to get a consistent big picture. Strategize about what those lead times are, how much concurrency can be employed between UX designers and programmers, and how much needs to be laid down in one

[34] Not accidentally, "Dos Equis," Spanish for "two X's," is the name of a top Mexican beer. Thanks to Kent McDonald, James Shore and Jeffry Fredrick for helping come up with this great term!

[35] This little bit of title inflation is due to the watering down of the title "user interface designer" at the turn of the millennium. Most organizations gave (still give) that title to people who have neither talent nor training in the specialty. As a reaction, true UX designers refuse to be classified as "mere" UI designers. This is a shame in all directions. One the one hand, UI design should refer to all aspects of appearance and use, not just the text or colors. At the same time, companies find people with talent in this specialty and should develop their skills in it.

single, initial design to create a consistent user experience. Where possible, use the hexagonal architecture (Cockburn ???) to decouple the system functions from the presentation of those functions.

Alert readers will notice that at this point we have left the realm of methodologies and entering the realm of project management strategies. What I hope is that the team will discuss the tension between overall consistency and incremental development, and pay attention to the lead times required between the users, the UX designers, and the programmers. Then experiment, reflect, and adjust.

## Program Governance, Burn Charts, and Systems Engineering

Agile development is sometimes rejected out of cultural reasons. One such cultural mismatch is with *systems engineering*, which deals with the design and construction of large systems, usually with mechanical, hardware and software components. Systems engineers typically accuse agile developers of being sloppy in their thinking.

There are two real difficulties in applying agile principles to typical systems engineering projects:

- Since the systems contain mechanical and hardware components, the feedback time from design to implementation to evaluation is longer than with software.
- The projects are typically large and contain multiple layers of subcontractors, and so it is extremely difficult to get real users or customers to view the growing system.

It was therefore a real treat to discover one point of near-commonality that helps bridge the culture and communications gap between the two worlds. That is the "earned-value" chart in the systems engineering world and the "burn up" chart in the agile world.

Here is a simplified explanation of creating an earned value chart, to show how it fits with the agile burn-up charts (a simple web search will turn up many tutorials on the subject).

- Write down and sequence all the tasks that need to be done.
- Estimate how long each will take. This number is the "value" of the task. It is not "value" in a market-value sense, but rather in the sense that completing that piece of work brings the project that much closer to completion.
- Using the sequence and time estimate, graph the estimated task rate on a graph with time horizontally and completion toward 100% vertically (see Figure 5'-9).
- As each task completes, *no matter how long it actually took*, credit the project with the scheduled amount of value for having completed the work. That is, if it was scheduled for 1 work-month on a 100 work-month project, but really took 2 work-months to complete, credit the team with 1% growth in "earned value" on the graph.

A similar pair of lines are created for the financial side, using expected and actual costs per task. They show the rate at which the project is burning through the budget.

The literature on earned-value charts is full of four-letter acronyms standing for the work estimated, the work completed, the difference between the two, and so on. What is important here is that the earned-value graph provides *one* way of seeing at a glance the actual speed of movement.

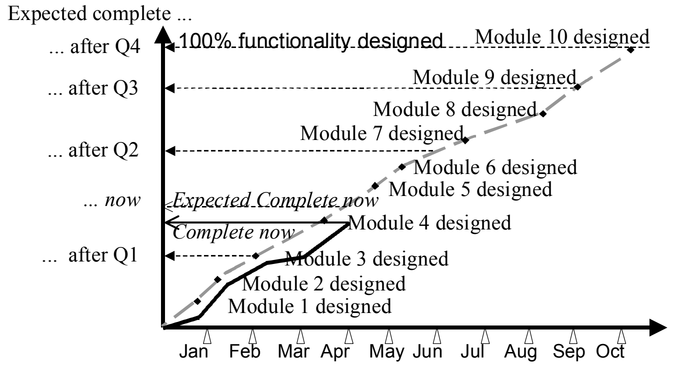


**Figure 5'-9**. Sample earned value chart (showing progress lines, not financial lines).

Figure 5'-9 shows a series of design tasks for a system consisting of ten modules. The months are marked on the horizontal axis with small triangles. The fat dashed gray line shows the expected design schedule, the shorter, solid black line shows the actual progress made to date. The vertical axis show % progress up to 100%.

I have added one thing to the normal earned-value chart in order to set up the discussion on *governance* coming up shortly. The vertical axis in Figure 5'-9 shows what should be completed at the end of each quarter, what should be completed by today's date, and what is actually completed at today's date. This projection gives an executive summary of the project status, without the details of the exact items being worked on. It shows the anticipated relative progress expected by quarter and where the team is now compared to where it should be. We shall make more use of this projection shortly.

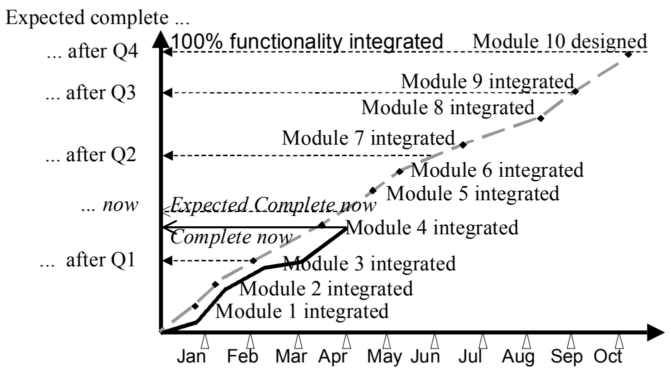


**Figure 5'-10**. Equivalent burn-up chart.

Here is the matching simplified description of how to create a *burn-up* chart.

- Write down and sequence all the system features or function that need to be developed.
- Estimate how long it will take to design, develop, integrate and test each feature or function. This number is the "value" of the task. It is not "value" in a market-value sense, but rather in the sense that completing that piece of work brings the project that much closer to completion.
- Using the sequence and time estimate, graph the estimated completion rate on a graph with time horizontally and completion toward 100% vertically (see Figure 5'-10).
- As each feature or function gets completed and integrated, *no matter how long it actually took*, credit the project with the scheduled amount of value for having completed the work. That is, if it was scheduled for 1 work-month on a 100 work-month project, but really took 2 work-months to complete, credit the team with 1% growth in "earned value" on the graph.

A similar pair of lines can be created for the financial side, using expected and actual cost per integrated feature set.

It is no accident that I copied the text from the description of the earned-value chart, nor that the two graphs look almost identical. The point I wish to make is that the earned-value chart in systems engineering can be converted to a burn-up chart in the agile world as soon as one (very significant) change is made: *No credit is given to completing requirements, design or programming!* Credit accrues only when the function or feature is integrated into the body of the system and passes testing. This is a much more reliable marker of progress than merely completing a paper design. It also requires the use of incremental development.

Several other people have written about burn charts and earned value charts. See, for example, John Rusk's "Agile Charts" (Rusk url), Glen Alleman's "Making Agile Development Work in a Government Contracting Environment" (Alleman 2003), and Mike Cohn's *Agile Estimating and Planning* (Cohn 2005).

Few people notice that the value that gets granted is not the market value of the feature or function but only its estimated completion cost. In this sense, it is not "value" at all, it is "cost". Some (rare) teams are working with their users and sponsors to assign market values to each feature or feature cluster. Doing this makes very clear the difference between something that takes a long time to develop, and something that is valuable to develop. Where they can do this, those teams are able to create true "earned-value" charts.

### *A Governance View*

"Program governance" refers to a steering group of evaluators who meet periodically to watch over an set of projects, or "program" in systems engineering jargon. When the steering committee gets together monthly or quarterly, they have a lot of data to sort through. What they need is a quick way to see the status of each project and spot what needs attention.

The vertical axis projection on the burn-up chart shows the information in an efficient way, and has the remarkable property that it can be used for both incremental and waterfall projects.

Figure 5'-11 shows how to get from the burn-up chart to a very compressed rendering of the project's predicted schedule and current status. Figure 5'-11(a) shows the vertical axis taken from Figure 5'-10.

(Two notes are in order before continuing with the description of Figure 5'-11: First, I chose a 12-month period because it is common period and it makes the quarterly marks come out tidily. Second, I chose to show *percent* complete. I could have chosen to show number of use cases, number of function points, or number of user stories instead. Which to use depends on who you are showing them to and what you would like to show.)

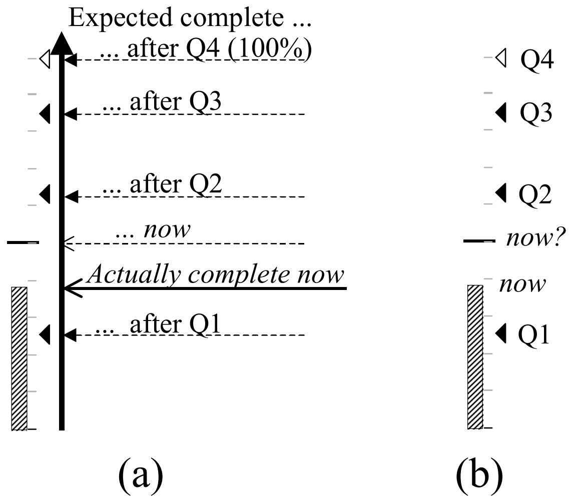


**Figure 5'-11**. The vertical axis of the burn-up chart converts to a compressed view of project status.

Just to the left of the "expected complete" arrows of Figure 5'-11(a) is a compressed rendering of the same information. The ten little tick marks placed vertically mark out 10% complete steps up to the 100% complete mark. The hollow triangle marks the amount of functionality expected to be complete by the end of the fourth and final quarter (it is, of course, at 100%). The three other black triangles mark *the amount of functionality expected to be completely integrated at the end of quarters one, two, and three, respectively.* Note that the existence of the triangle marks the time value (Q1, Q2, Q3, Q4), and the vertical placement of the triangle marks the amount of functionality completed.

The amount of functionality expected to be complete "now" is shown by the placement of a longer horizontal line. This line moves upward continually as the time flies through the year.

The amount of functionality actually completed "now" is show by the shaded thermometer bar. The bar grows upward only as new functionality gets integrated and completed.

(Second note: For projects in which a significant amount of documentation is required as part of the deliverable, credit for "done" is only given when the code runs *and* all the documentation is complete for that functionality.)

There are three particularly nice things with the triangles, line and bar in Figure 5'-11(b) besides just compactness:

- They can show a myriad of development strategies, from waterfall to agile, and anywhere in between.
- They can be overlaid on a system architecture diagram to show program status.
- They can be slightly modified to show the relative status of requirements, UI design, programming and documentation, again, for very different project strategies.

Let's look at those one at a time.

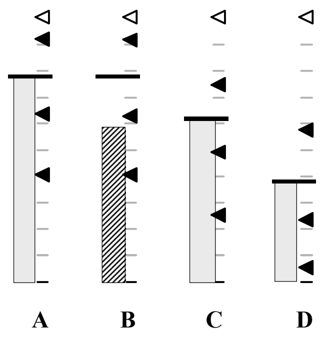


**Figure 5'-12**. Four projects with different strategies and status. [[AC:fix! separate AB from CD]]

Figure 5'-12 shows the strategy and status of each of four projects, shown nine months through a one-year plan.

Projects A and B are using the same strategy: develop and integrate about 40% of the total functionality in the first quarter, then about 63% in the second quarter, then about 90% in the third quarter, and finishing it off in the fourth quarter. Presumably this strategy is intended to allow plenty of time for feedback, testing and revision in the final quarter.

The difference between projects A and B is that A is on schedule, and B is behind (it hasn't even finished the functions scheduled to have been completed at the end of the second quarter).

Projects C and D, both on schedule, have very different strategies. Project C intends to develop very linearly, developing and integrating about 25% of the functionality each quarter. Project D is set up to run in a hybrid (semi-waterfall, semi-agile) fashion. The team expects to integrate less than 10% of the functionality in the first quarter (presumably they will be gathering requirements and setting up the architecture at that time). They will still have less than 25% of the functionality integrated by the end of the second quarter, then 60 at the end of the third quarter, and they plan to complete the last 40% of the work in the fourth quarter.

John Rusk wrote in with an idea to show both financial and progress markers at the same time. Draw two renderings of the triangle, line and bar, one for progress and one for cost (color or shade them differently, of course). Figure 5'-13 illustrates (it is a lot easier when in color on a screen, instead of black-and-white in this book).

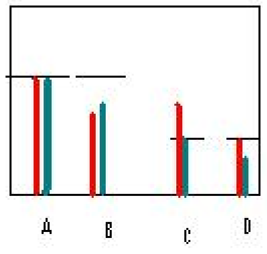


**Figure 5'-13**. Four projects with varying progress and cost status (courtesy of John Rusk).

The nice thing is that you can diagnose what is wrong from seeing the two bars at the same time. In Figure 5'-13 we see that A is on track for progress and cost. B is behind schedule due to under-resourcing (the cost is just as low as progress). C is on track for progress but the team is working overtime or extra team members have been added (cost is above prediction). D's costs are as planned but progress is slow.

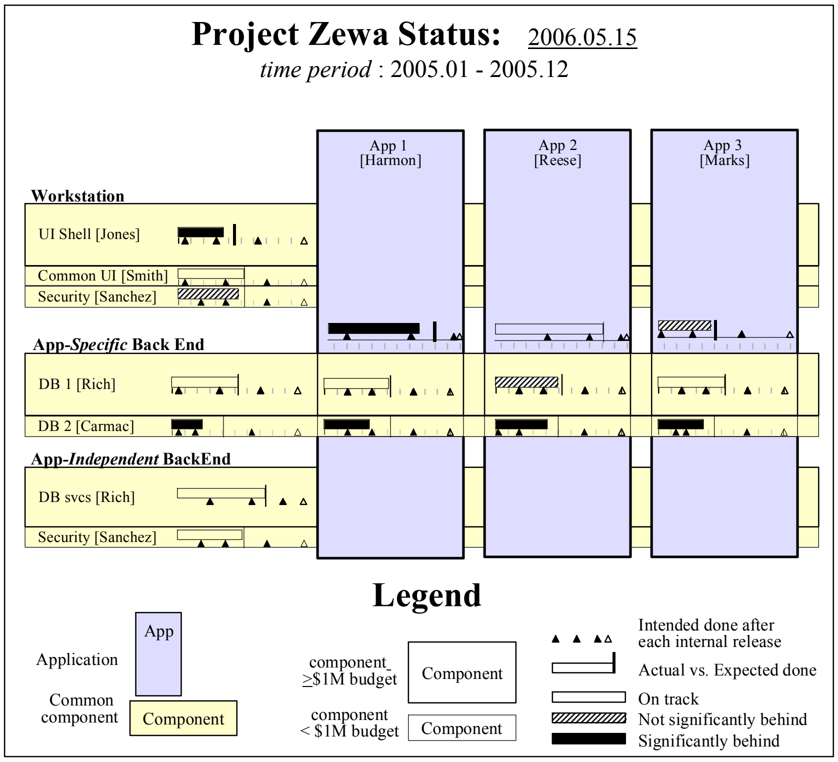


**Figure 5'-14**. Full program status for a program involving three applications sitting on a common architecture (Cockburn 2005 g).

The next thing that can be done with the governance graphic is that it can be overlaid on a system architecture diagram to give a total program snapshot. Figure 5'-14 shows such a picture (the figure is quite busy, and the interested reader is referred to the original article in which it is described in detail (Cockburn 2005 g)).

Briefly, there are three applications being developed on a common architecture consisting of a workstation portion and a back-end portion. Some of the back-end portion is application-specific and some is application independent.

There are three things to notice about the graphic:

- The thermometer bars are laid horizontally instead of vertically, for reasons of space. This does not change their reading. Time and completeness flow to the right.
- We are able to show the status of each sub-project in the program, and we can immediately see which sub-projects are on-time, a little behind, and significantly behind. This is exactly the information the governance board wants to see at a glance.
- We can show even more information than discussed so far. The technical lead for the program for which this graphic was first developed wanted the *thickness* of the components to indicate *how much money* was at stake in each. As he said, "If a small project or sub-system is behind, I won't worry about it as much as if a large one is behind." On that project, we also had system migration issues to show, and used additional coloring and marking.

When the steering committee sees a project that is behind, they will naturally want to turn to a detail page for that project. The third piece of good news is that we can use the same marking mechanism to show the accomplishments of each sub-team.

Figure 5'-15 shows sub-project detail for two projects using very differing strategies.

Project A's team is using a concurrent development approach. Their graph show that they intend to complete about a quarter of their requirements in the first quarter, almost two-thirds by the end of the second quarter, and about 90% by the end of the third quarter. Their UI team will run just behind, completing less than a quarter of the UI design work in the first quarter, then about half, and over three quarters in the next two quarters. The programming team will run just behind them, completing roughly 20%, 45, 78% respectively in those quarters.

The team of project B is using a waterfall approach. That team plans to get the requirements completed in the first quarter, the UI design completed in the second quarter, and the programming done in the next two quarters.

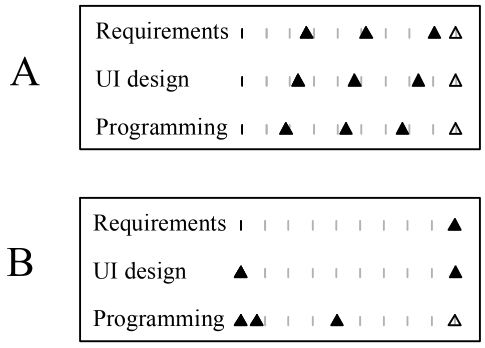


**Figure 5'-15**. Detail sheets for two projects with different strategies and status (from Cockburn 2005 g).

Both strategies can be shown on the same graph, simplifying the job of reporting status on multiple projects using differing mixes of waterfall, incremental and concurrent development.

More on the use of these projections and the use of color is given in (Cockburn 2005 g).

## Use Cases and User Stories

XP brought with it the idea of marking requirements in short phrases (*user stories*) on index cards (*story cards*) It is important to note that the short phrases on the cards are not "the requirements," they are markers that promise a conversation about what is associated with the phrase on the card. In this sense they are efficient markers and tokens in the cooperative game as described in Chapter 1.

However, unwary beginners, mistaking the early XP as defining agile practices, mistakenly thought that all agile development must be done with requirements captured as user stories on story cards.

### *User Stories Overloaded*

The problem is not that XP contains a practice of marking requirements in short phrases on index cards. That is a useful practice in many instances. The problem is that since XP was so popular, many people just starting with agile development thought they were allowed *only* to write requirements as single sentences and *only* on index cards, that anything else would un-agile (and therefore *bad* in the eyes of their peers). Just as story cards make a strong strategy in certain situations, they make a weak strategy in others.

XP was created and tested in situations with a small user base. Developers could and did simply ask their users what they wanted. With short delivery cycles and only a few users, seated close by, it was not necessary to spend much time writing down details and sorting through conflicts in the requests. The team simply wrote down the key phrases, asked the designated "customer" what to do at each choice point, programmed that up, and showed the result to the designated customer.

There are, however, many situations where those conditions are not present: shrink-wrapped products, systems with a very large or diverse set of users, or systems being deployed into large, multicultural organizations. In these situations, the very notion of asking *the* customer, or getting a quick answer to a question, don't make sense.

Here are two examples from organizations I have visited:

- A set of hospitals, some small, simple and remote, some large, urban and sophisticated, were collecting requirements for a new medical system to use by the patient bedside. Not only were the different hospitals concerned about their respective hospital rules, they had to include into the requirements the differences between emergency and regular care, inpatient and outpatient care, and between the different specialists: doctors, nurses, pharmacists, and assistants.
- A state government organization was working to change and streamline their operating processes in different offices around the state. The new software was to support both the (slowly) evolving recommended process and the variations allowed for the different locations.

In these situations, it often takes weeks to get the answer to a question about how the user interface should work, or the details of a seemingly simple business policy question.

Sadly, I have heard numerous people on stage telling the following sort of story in an experience report:

> We did very well with our XP/agile project, except that we couldn't seem to get the right Customer on the project. It often happened that we would ask the Customer a question, and it would take several weeks before she came back with an answer. Our project ran into many delays as a result. We can recommend that you get a proper Customer on your project to avoid these delays.

After hearing this refrain on a number of times, it dawned on me that the problem was not with their customer, but with their process. They had built into their it the misconception that there was one person anywhere who could give a definitive answer to business policy questions that were likely to cross multiple departments.

The solution to the problem is not to look for an omniscient customer, but to base the project's strategies around the concept that some questions will take a certain (long) time to answer. The trick is to ask those questions far enough in advance so that the delay in getting them answered doesn't delay the project. I call that "look-ahead."

(This is a good time to re-inspect Figure 2-3. Imagine in that decision-flow diagram that asking the customer a question is putting an inventory item into their queue. Some of these inventory items take a short time to complete, while others take a long time.)

There are three points of damage from using user stories and story cards in the wrong place:

- *Lack of look ahead*
- *Lack of context*
- *Lack of completeness*

The first point of damage is the one just mentioned: it relies on developers being able to get quick answers to the detailed business policy questions that arise as they code. There are organizations where sound answers can't be decided quickly. The team needs a better form of look-ahead, to detect earlier on what questions are likely to be asked, which will be easy to answer and which will take time.

The second point of damage is lack of context. A well-written user story is a very short description, often just one sentence long, of something a user wants to have the system do: "Italicize text", "Add a line to an invoice", "Compute the additional overtime pay for an hourly employee H hours at 1.5 times R rate of pay." The shortest user story I have heard of is the one word "Kaboom" on a naval artillery projects, meaning "Take a shot and tell me where it hit."

A good thing about user stories is that they can be subdivided into tinier and tinier pieces to fit within shorter and shorter iterations, and still be considered valid user stories. On one project, a user story was written: "Select text." As the team went to implement it in an early iteration, they decided to split it into two smaller user stories, "Select text with the mouse" and "Select text with the keyboard". Their intention, quite reasonable for that project, was to get something working early and grow it over time. Having one way to select text early meant they could develop other parts of the system separately from extending the text selection mechanisms.

The only rules for a user story are that it has user value, the team can draft an estimate of how long it will take to develop, and that estimate fits within their iteration period. There is nothing in the idea of a user story about how much value to the user it contains.

Being able to split user stories into very small slices generates trouble when there are hundreds or thousands of user stories. The developers and users find themselves looking at sentences that are very hard to relate to business situations. They are presented without an operations context, and without being related to each other – which ones come first, which one surround others, and so on.

At the time of this writing, there is a discussion on Yahoo's "agile-usability" discussion group about just this point. One person is asking for ideas on how to integrate hundreds of tiny user stories into a coherent whole, so that he can come up with a decent user interface design for the system. UI design requires context to be done properly; user stories don't provide that context.

For a feature in isolation, for a user story in isolation, or (for systems engineering types of contracts) for a *shall* statement in isolation, the context is information that can't be derived from other information. Only the most experienced business or usage experts can sew together the hundreds of sentences into a coherent and sound whole. This is expensive and difficult work to do.

The third point of damage is lack of completeness. As the developers ask questions, the answers generate more user stories, often more than were anticipated. On a plotted burn-up chart (as Figure 5'-16), the total number of stories keeps rising as they encounter unexpectedly complex business rules and split stories.

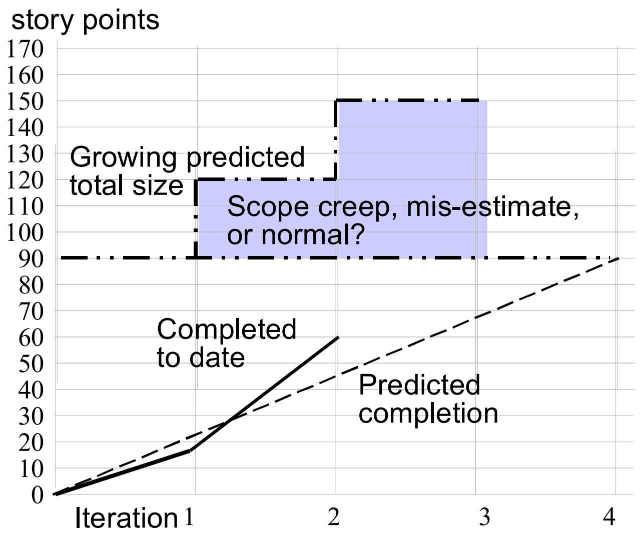


**Figure 5'-16**. Growth in estimated number of user stories needed to complete the project.

On a project that is suited to the user-story method – a small project written for a few, local users – this is not a problem. The development team simply delivers a new system every few weeks and the sponsors can tell whether they are happy with the rate of delivery. They view the ongoing work as a flow of cash for value, and steer accordingly.

On a fixed-price, fixed-scope project, however, the burn-up chart in Figure 5'-16 presents a major problem. Is that sharp growth in the size of the system due to scope creep, poor initial estimates, or simply splitting stories in a normal way? I watched an agile team attempt to complete a fixed-price, fixed-scope project bid in which this problem arose. The client-side sponsors felt they were not going to get the system they had been promised; the contractor-side executives felt that they were having to over-perform on the contract; the developers felt that there was too much scope creep; and the users felt that the project was never going to get delivered.

### *Use Cases Revisited*

Use cases solve those particular three problems However, we must first repair the bad reputation use cases have picked up from the people who write them poorly.

Most organizations produce *interaction* use cases[36] that are long (think 20 – 30 pages), contain as much screen design detail as the writer can include, and are hard to read. It is no wonder that they have a bad reputation.

That reputation applies only to interaction use cases. A good use cases is short (think 1-3 pages[37]), readable, and contains no UI design. I used to call them *intentional* use cases, but I actually prefer Larry Constantine's term, *essential* use cases (Constantine 2001). Interaction and essential use cases are worlds apart in terms of ease-of-use and economics.

On the $15M project mentioned earlier, we had over 200 use cases that served as the contract basis for the fixed-price, fixed-scope project. Even there, the use cases were essential use cases written in "casual" style[38], about two paragraphs of simple prose. They were updated at the start of each three-month delivery cycle to pick up the latest business needs and development details.

Essential use cases, two paragraphs or two pages long, can be drafted, read and updated easily. The team can write them all up front or incrementally, each to completion or one scenario at a time, at a low or high level of precision at any moment. Implementation can begin immediately or be deferred. Cost and value can be estimated on a line-at-a-time basis or on a use-case basis. In other words, you can arrange their construction and use according to your project's preferred strategies.

Well-constructed essential use cases solve the three problems mentioned earlier:

- *Completeness and look-ahead*

The extension-conditions section of the use case is a place where the usage and business experts can brainstorm of all the situations that the developers will have to deal with related to the use case. Brainstorming is done in a matter of minutes. The list of extension conditions then acts as a completeness criterion for the system's required behavior.

The project team can (and should) strategize over that list to decide which extension conditions should be developed in which delivery cycle. They can (and should) attach development estimates and perceived business value, even at "low, medium, high" levels, to understand what they are constructing for the business.

In the agile environment, a use case provides the context for what is wanted overall, but it is not the case that the team delivers all of it at once (Dave Churchill called the initial delivery "a sparse implementation of a use case, with a subset of extension conditions")

In examining the extension conditions, the team makes an assessment of how hard it will be to gather the information needed to complete the requirements associated with each condition ("low, medium, high" is often a good enough measure). Those requiring a lot of investigation or discussion should be started earlier, ahead of when the item is scheduled for development. Extensions estimated to have a quick response time can be left alone longer, possibly until the start of the iteration. In terms of Figure 2-3, the team adopts is a work-scheduling strategy that arranges for information to reach the person needing it in a timely manner.

- *Context*

Since the story line of the system's use is captured at the same time as the system's behavioral features, use cases provide the context needed for each feature in the system. *Kite* or high-level use cases give context to individual *user-goal* use cases[39], and the user-goal use cases give context to specific user actions.

### *Mapping Use Cases to User Stories*

Gerard Meszaros worked out how to fit use cases and user stories together (Meszaros 2005). The problem he solved is this:

- A user story may be a request for new functionality, extension of functionality or data, or improved usability (UI design).
- A use case does not contain data or UI design information.
- A user story must be small enough to be implemented in less than an iteration, however short the iteration may be.
- A use case may contain 10, 20, or 30 user stories, but it is not the case that each step in a use case is a user story: it may take several use case steps to capture

[36] Oddly, although I named these things back in 1994, specifically to tell people *not* to write them, that nomenclature never made it into any of my public articles. Perhaps I can fix that mistake now. In an odd twist of fate, off-shore development has caused a need for interaction use cases as the carrier of high-precision, low-bandwidth communication across cultures and continents.

[37] If I got only *one* rule to impose on an organization, it would be, "Every team delivers running, tested features to real users every three months." If I had to name a second, it would probably be: "No use case exceeds two pages."

[38] See *Writing Effective Use Cases* (Cockburn 2001) for description of briefs, casual and fully dressed use cases, levels of precision in writing, and low-, medium-, and high-level use cases.

[39] (Cockburn 1995sucwg, Cockburn 2001uc).

what is a single user story, or one use step may be complicated enough that it needs to be split into several user stories.
  - The first user story to implement is the simplest conceivable path through the main success scenario, leaving out any complications whatsoever. This is a user story that maps to multiple use case steps.
  - Most of the leftover steps and most of the extension conditions map to one user story each.
  - Some steps describe business rules or behaviors difficult to implement, either of which may need to be split into several user stories, implemented in different iterations.

This is the mismatch between user stories and use cases: some user stories are in the use cases and some are not. That makes it difficult to "convert" use cases to user stories.

Gerard's solution is simple and clever. He names four types of user story:

- *New functionality*. Implement the simplest possible path through the main success scenario of the use case. Choose it to be as narrow as possible, to fit into your iteration length.
- *New rule*. Implement part of a business rule, usually from another step or extension in the use case (occasionally not in the use case at all, as with complex business rules that live in their own documents).
- *New data*. Implement part of a data structure (which is probably not in the use case at all).
- *Improved usability*. Correct or refine UI features, whether they were part of the original plan or come from user feedback.

Gerard solved the conceptual problem of relating use cases and user stories. We now have tooling problem.

One organization constructed a special tool to help them track the mapping. People write use cases and non-behavioral requirements to define project scope. They break those down into user stories, or *chunks of work* (*COWs*) as the team refers to them (the team recognized the trouble with the term "user story" and changed their vocabulary). They attach effort estimates to the COWs and assign them to development teams for specific iterations.

The tool allows the user to type anything in the COW description and associate that with a use case or a step in the use case, and, independently, with a team and with an iteration.

The tool has three panes, with drag-and-drop between panes.

- Pane 1: A tree view of use cases and COWs (a use cases can contain other use cases, and a use case can contain COWs).
- Pane 2: A list of COWs assigned to teams.
- Pane 3: A list of COWs assigned to iterations.

With this tool, the team can write and evolve their use cases, and develop them according to any strategy they like. They can always tell what portion of a use case is implemented, to make sure that the functionality provided to a user is "fit for business use." They can see the context surrounding each COW. They can generate burn-up charts showing their rate of progress.

(Before you write me asking for a copy of the tool, it was developed for internal use only, and the above is all I am allowed to say about it, except that it took about a month to prototype and another month to get into use. With modern development environments, this sort of tool is not a large task.)

For those who do not have such a tool, there is an even simpler method:

- Print out the (1- or 2-page) use case and post it on the wall.
- Highlight with yellow highlighter the sentence or sentences being implemented in this iteration.
- Annotate to the side the data structure or business rule associated with any complex step and highlight it similarly.
- Highlight the sentences with blue (or green) highlighter as they get completed. They will now appear green.

With this method, people will always know what they are working on and what has been completed. Best of all, they will be able to tell what portion of a meaningful user activity is available and what portion is being left out.

## Persistent Questions

Over the last five years, the same sorts of questions keep arising: the limits of the agile model, agile's relationship with the Software Engineering Institute's Capability Maturity Model (SEI's CMMI), with ISO 9001 conformance, with customer relations, with domain modeling. They also ask how to introduce agile into a company, what "more agile" means, and how to tell whether their organization is agile at all.

This is the place to consider those questions.

### Sweet Spots and the Drop-Off

When applying good practices, there is a gradient and a steep drop-off. Along the gradient, doing more and better is good, but only a little bit better, but across the drop-off, doing less can be catastrophic. I suggest that getting all the key practices above the drop-off is more important than getting just one or two of them to the top of the gradient.

Recall the five sweet spots of agile development mentioned on p.???:

- Two to Eight People in One Room
- Onsite Usage Experts
- One-Month Increments
- Fully Automated Regression Tests
- Experienced Developers

It is not the case that *if* you have those sweet spots in place, *then* you can use the agile model. Rather, it is the case that *to the extent* that you can *get closer* to those sweet spots, then the *easier* it is to take advantage of the agile model, and the lighter and faster you can move.

For the first one, collocation, we see that if the people are not in the same room, but in adjacent cubicles, communication is not *as* good – this is the gradient – but it still adequate for even the Crystal Clear model to work. Once people move farther apart than the length of a school bus[40] and around a corner or two, their communication becomes so constrained that very particular dangers arise – this is the drop-off.

Similarly, it is not really necessary to have a full-time, onsite expert user in order to get good usage and usability information. I have seen projects do very well with expert users two hours a week onsite with phone calls during the rest of the time. This is the gradient. Less than once per month is on the low side of the drop-off – there just is not information and feedback from the expert, and the project is in danger of delivering the wrong system[41]. Most projects never get to see a real user expert at all.

One-week, one-month and three-month deliveries all lie on the gradient. One-year deliveries are off the drop-off.

Continuous, daily, even every-other-day integrations all lie on the gradient. Weekly integration and manual testing are off the drop-off.

Having all competent and experienced developers is really great. One competent and experienced developer for every three not so outstanding or young and learning developers is still on the gradient. Having only one competent and experienced developer for seven or ten of the others is off the drop-off.

Understanding the gradient and drop-off helps us understand the different agile methodologies being published. One of the goals of XP is to get as high up the gradient as possible. Thus, the original XP called for establishing the center of each sweet spot.

The goal of Crystal is project safety, with regular, *adequate* deliveries. Therefore, Crystal only calls for establishing the top of the drop-off. Every improvement from there is left to the discretion of the team.

Think about where your practices are relative to the drop-off. Consider that getting all of the practices above the drop-off is more important than getting just one or two of them to the top of the gradient.

## Fixed-Price, Fixed-Scope Contracts

My debut as a lead consultant was on an 18-month, $15M fixed-price, fixed-scope contract, with eventually 45 people on the development team. I found on that project that all the ideas we later wrote into the agile manifesto were needed to pull off the contract**:** incremental development, reflection, frequent integration, collocation, close user involvement (we succeeded without automated tests, which these days I would try harder to get in place).

Most fixed-price, fixed-scope contracts are priced aggressively, and typically require large amounts of overtime. The problem is not rapidly changing requirements, but development inefficiency. Fortunately, the agile ideas increase development efficiency, and there is almost always wiggle room in the development process that the team can exploit to increase their efficiency (See "Process: the 4th Dimension" (Cockburn 2004 pt4d)).

Thus, I came to the writing of the agile manifesto through the door marked "efficiency" instead of the door marked "changing requirements."

If you are working on fixed-scope contracts, remember two things:

- At the end of each iteration or delivery, if you need to keep the scope constant, *simply don't change the requirements*! There is nothing in the agile manifesto that says you have to change requirements at the end of the iteration; only that agile development facilitates requirements changes in those situations where it is appropriate.
- The contract puts a bounding box on the time, scope and money you can apply. Within those limits, you can collocate the team, work incrementally, pay attention to community and communication, automate testing, integrate often, reflect and improve, and involve users.

## Agile, CMMI, ISO9001

Rich Turner highlighted for me the key difference between the CMMI and the agile methodologies: The agile methodologies are focused at the *project* level, whereas the CMMI is focused at the *organizational* level.

[40] See the Bus-Length Communication Principle, p. 30???
[41] See the research study described in (Cockburn 1998, pp.??-??) on "user links" as critical to project success.

There are additional, philosophical differences that can be debated, but one must first take into account that their targets are different.

An agile initiative addresses the question: How do we make *this* software on *this* project come out well? A CMMI initiative addresses the questions: How well is the organization doing at being what it wants to be? Do the different groups have in common what they are supposed to have in common? Do they train newcomers to behave in the ways the group want? And so on.

Thus, it is not a meaningful question to ask whether XP is a CMMI level 3 methodology, because XP does not contain any rules about how to train an XP coach, how to detect whether people are pair programming or refactoring (or why to bother detecting that). Yet those are the things the CMMI wants the organization to think about.

Since the writing of the agile manifesto, an increasing number of agile practitioners have been asked to help install agile processes across an entire organization. Interestingly, we find ourselves asking the same questions the CMMI assessment process asks: Is your team really doing daily stand-up meetings, and how can you tell? What is the training for a new Scrum master? Where are your reflection workshop outputs posted? Where is your information radiator showing the results of the latest build?

It may be time for agile practitioners to take a close look at the lessons learned from the CMMI as relates to creating common processes across an organization.

Taking into account that the targets are different, we find that

- Getting an agile development shop assessed at CMMI level 2 or 3 may not be so difficult, especially as the SEI is starting to train assessors to work with agile processes.
- Deep philosophical differences remain between the CMMI approach and the agile approach.

There are two very fundamental, philosophical differences between CMMI and agile that generate tension between the two worlds.

The first difference is this:

- The CMM(I) is based on a statistical process assumption rejected by the authors of the agile manifesto.

Process engineering literature categorizes processes as either *empirical* or *theoretical* (also called *defined*). In the process engineering literature, a *theoretical* or *defined* process is one that is well enough understand to be automated (quite a different definition of *defined* than the CMMI uses!). An *empirical* one needs human examination and intervention. The textbook by Ogunnaike (1992) has this to say:

> It is typical to adopt the defined (theoretical) modeling approach when the underlying mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the defined approach, the empirical approach is the appropriate choice.

For a defined or theoretical process to work two things must be true. First, prediction must be possible; the system must be predictable in its response to variations. Secondly, one must know and be able to measure all the parameters needed to make the prediction.

I don't think either is satisfied in software development. First, we don't yet know the relevant parameters to measure. The purpose of the agile movement and this book has been and still is, to some extent, to get people (including researchers) to pay attention to new parameters, such as quality of community, amicability, personal safety, and factors affecting speed of communication ("erg-seconds per meme").

Secondly, even when we get far enough to name the relevant parameters, I think it quite likely that team-based creative activities such as software development are chaotically sensitive, meaning that a tiny perturbation can cause an arbitrarily large effect. As one example, I once watched in amazement as a high-level employee quit the organization in anger over what seemed to some of us as a seemingly small off-hand comment. I'm sure she had been inflicted with this comment and ones similar before, but on this day, the same comment was just one too much, and she quit.

The CMMI ladder is built on the assumption that both conditions are satisfied[42]. This assumption is embedded in levels 4 and 5. At level 4, the organization captures numerical metrics about the processes. At level 5 those metrics are used to optimize the process. The assumption is that gaining statistical control of the software development process is the key to doing better.

The second philosophical difference between CMMI and agile is this:

- The CMMI places optimization of the process at the final level, level 5. Agile teams (teams using Crystal, in particular) start optimizing the process immediately.

At the very moment of starting their agile work, the team will be challenged to ask themselves how they can do better.

In asking how to do better, the statistical process assumption shows up again. In what follows, I will speak for

[42] Ilja Preuß mentions that the book *Measuring and Managing Performance in Organizations* [[ref???]] has interesting material about what dysfunctions arise if you work to such an assumption where it is wrong.

Crystal in particular, in order not to speak incorrectly for anyone else.

In the reflection workshop or post-iteration retrospective, the process and the project's condition are evaluated by people's emotional responses in addition to numerical metrics.

My working assumption is that a human, as a device, is particularly good at taking in a lot of low-grade analog information along various channels, and summarizing it in a feeling or emotional statement. Thus, when a person says they feel "uneasy" about the state of the project, or "uncomfortable" with the last iteration's communication patterns, they are, in fact, saying a great deal, even when they can't provide numerical values or details. It is the very fact of feeling uncomfortable that the team must attend to.

This view, which seemed out of fashion at the time I first formulated it, has been getting attention in recent years. Several books have been written around it, most notably *Blink* (Gladwell 2005???),??The fire book??? and *Sketches of Thought* (Goel 1995).

From sharing their feelings about what is going on, the team is in a position to start improving their working habits long before they get statistical data, and even before they are consistent in their habits. This is a fundamentally different to the CMMI model.

There is one final, significant point to be made about combining agile and CMMI: It appears to be easier to incorporate agile practices into an organization already assessed at CMMI level 3 or higher than to move an agile project team into CMMI.

A CMMI level-3, level-4, or level-5 organization has already demonstrated the ability to adopt a process organization-wide to train the people and the leaders in their roles. The agile elements they will add are unlikely to endanger that rating in any way:

- *Collocation*. The team may not be collocated initially, but if they choose to collocate, that will not affect any other part of their process.
- *Daily stand-ups*. Getting the team together daily to talk about what they are doing should not affect any of the team's other rituals or deliverables.
- *Incremental development*. Many CMMI-certified shops do not do incremental development, but working in increments should not break any part of their process.
- *User viewings*. Having one or more users visit and give feedback to the development team each week is a good practice, and will not affect any part of their documented process.
- *Continuous builds, automated regression unit and acceptance tests*. These are probably recommended in their process, and won't be damaged for being turned up.
- *Reflection workshops.* Getting together each month to discuss how to do better will not endanger any process assessment. The tricky part comes if the team decides it needs to change some part of the their process to do better, and that change puts them out of step with the greater organization's process..

In other words, a CMMI organization should, theoretically, have no trouble in adopting the key recommendations of the agile approach. Most organizations striving for CMMI assessement that I have talked to are unwilling to do those things because they don't see them as competitive drivers.

The agile organizations I have visited generally have no interest in CMMI certification at all, viewing the CMMI ladder as a simple waste of money. That is, they don't need the CMMI certification to get contracts, but it will cost time and money to get that certification.

The way I understand these two views is that organizations that profit from having a CMMI level-3 assessment for certain government contracts are not competitively hampered by not having the levels of productivity and responsiveness offered by the agile approach. Organizations operating in a high-flux, competitive environment and needing the agile approach for organizational survival, can't afford to or don't see the value to spend the money or time to create the process superstructure required to get the CMMI level certifications.

The difference here is based, not on philosophy, but in priorities created by the business operating environments.

The CMMI question transfers to the equivalent question about ISO9001 certification. ISO9001 certification is easier, in one sense: there is only one step on the ladder. The statistical assumption and the late optimization objections therefore don't apply.

One company, Thales Research and Technology (TRT UK), as part of its Small Systems Software Engineering activity, ran a trial project using the Crystal Clear methodology, and had an ISO9001 auditor evaluate what it would take to get that project's process to pass an ISO9001 audit. The report is too long to reprint here; it is presented in *Crystal Clear* (Cockburn 2004cc), pp. ???-???. The interested reader is encouraged to examine that report. The sort of comment we see from the auditor, though is of this nature:

> In order to be fully compliant with this clause of ISO 9001, whiteboard records [ed. note: they used printing whiteboards]] would need to indicate who was involved with the viewings and workshops. In addition, the Sponsor and/or User should be present at some or all in order to ensure reviewer independence. Actions

should be clearly identified to enable tracking at the next workshop. The whiteboard records would need to be kept for the mandatory records of review required by ISO 9001. (Cockburn 2004cc, p.320???).

In other words, there was not a problem in using design and reflection sessions at whiteboards, as long as everyone present signed their names on the whiteboard and dated it (and then either photographed or printed it).

When the team updates their working conventions, as called for in Crytal, they would presumably similarly print, sign and date the new list. They would then use that as the new process to evaluate against.

**Another View of Agile and CMMI, by Paul McMahon, Principal, PEM Systems**

While in some cases I agree it is "easier to incorporate agile practices into an organization assessed at CMMI level 3 or higher than to move an agile project team into CMMI," my experiences indicate that this is not always true.

I am helping a number of organizations with high CMMI maturity (Levels 3, 4, 5) become more agile. But I am also helping small organizations, which started out agile, use the CMMI model for process improvement. Last year I participated on a formal appraisal for a small company that is very SCRUM-oriented and they achieved a CMMI level 3 rating. If you use the CMMI model as it was intended I don't believe you have to "move the team into CMMI".

The CMMI model is a reference framework for process improvement. It isn't a set of required practices. In the case I mentioned, we didn't change the behavior inside the small company, except in isolated instances where change was clearly beneficial. For the most part we demonstrated where what they do (largely Scrum) meets the intent of the Level 2 and Level 3 reference practices. To accomplish this requires a good understanding of the CMMI model, including the use of what is referred to as "alternate practices."

As an example, the agile daily standup meetings can be viewed as an "alternate practice" within the CMMI model. These daily meetings along with the follow up actions by the ScrumMaster can achieve the intent of a number of the specific practices within the Project Monitoring and Control Process Area.

User viewings meet part of the need for stakeholder involvement which is an important aspect of the Project Planning and Requirements Management Process Areas and the Generic Goals within the CMMI model.

With respect to reflections workshops, the CMMI model actually recognizes that change is a good thing. It expects the team to capture lessons learned and improve the process. So reflections workshops support lessons learned feedback and improvement as required within the generic goals of the model. The one subtle difference is that if you are going for Level 3, then improvement recommendations that come out of the team should not be limited to the team, but should also be communicated to the organization level so other teams in the organization can benefit from what you have learned.

Attaining CMMI Level 3 doesn't have to mean more detailed process definitions. It means that the processes that work for an organization are shared across that organization and adhered to. Nothing says they can't be "agile processes". Institutionalization is an important part of the model, which means the processes don't break down in times of crisis and aren't ignored when new people are brought in. In my view it is often easier to institutionalize agile practices because agile teams usually believe strongly in how they do their job.

As another example, the agile company I have been helping has an open communication culture where lead engineers aren't afraid to walk into a senior manager's office as soon as they know they have a risk, or just need help. The team members are also encouraged to raise risks at the daily standup meetings, and they often do.

At first, based on the Risk Management Process Area in the CMMI model, we created a form to capture risks more formally, but the form was not well received in the organization. So instead we defined a risk process that captured exactly what the team does, and then trained new personnel in the organization's expected risk management behavior. We did require that they capture their risks in a periodic briefing to Senior Management, but we continued to encourage the informal and immediate communication that was already working well.

Our lead appraiser had no trouble at all with processes like Risk Management that clearly were institutionalized across the organization, and met the intent of the process areas—as long as the "direct artifacts" and "affirmations" substantiated the effectiveness of the process (direct artifacts and affirmations are discussed below).

My experience has been that when we try to "formalize" effective processes too often this results in negative side-effects because we unintentionally change what works. As an example, if we required formal written meeting minutes on the agile daily standup meetings, people would be less likely to speak openly, based on my experience. I have observed this type of behavior change on numerous occasions.

Another way to look at what we did was to capture what the successful people in the organization were already doing—in this case mostly Scrum—and then share it across the organization. This is an effective way to use the CMMI model and agile methods together. These examples demonstrate that agile practices can actually help achieve CMMI goals, if the model is used correctly as a framework.

Be aware that one of the keys to this working is to get a CMMI appraisal lead who understands how the CMMI model is suppose to be used, and understands agile methods. For example, you need to have a lead appraiser who isn't afraid to use "alternate practices" as they were intended by the model. Some lead appraisers discourage the use of alternate practices out of fear it will be perceived as "trying to get out" of a required practice. This can be a legitimate concern, but it shouldn't be used as a reason to change what is already working in a successful agile organization.

Although it is true that some "organizations striving for CMMI assessment" are "unwilling" to employ agile practices, my experience indicate that many organizations involved with DoD work are actually very willing and interested.

I have worked with multiple large U.S. Defense contractors who came to me, already having achieved a CMMI Level 3, 4 or 5 rating, but wanting to infuse more agile practices into their work processes. Their motivation has come from multiple sources including customer interest, developer interest, and a belief that to continue to be successful – and maybe to survive – increased agility will be required.

A question I am hearing frequently from large DoD contractors is, "How can we infuse more agile operations into our existing processes?" In one case, we created an agile developer's guide to help his projects tailor their traditional company process assets employing agile practices.

The DoD acquisition community has also expressed interest through questions like, "How can we get our people trained in agile methods so we can be more effective at evaluating the agile implementations of our contractors?" In this case, we are training acquisition community personnel and providing evaluations and recommendations on DoD projects that are already moving toward increased agility.

It is certainly true that many "Organizations operating in a high-flux, competitive environment and needing the agile approach for organizational survival, can't afford to (or don't see the value to) spend the money or time to create the process superstructure."

But, at the same time, my experience indicates you don't necessarily need a "process superstructure" to achieve a high CMMI Process Maturity Rating. Let me explain why I believe this, and why so many organizations still go the "process superstructure" route.

To attain a staged CMMI Level 3 involves 18 process areas and 140 expected specific practices. Today, many CMMI-based process improvement initiatives focus on the stepwise procedural definitions which can be implied (although erroneously in my view) from the specific practices. This approach often leads organizations to produce explicit objective evidence for each expected specific practice. These efforts frequently lead to team frustration and reduced performance. Comments such as, "our processes don't reflect what our people really do," and, "our processes force us to do non-value added work," are not uncommon.

I believe this approach goes wrong based on the way objective evidence is handled. Objective evidence (OE) is critical to the appraisal method, but more than one type of OE is allowed. To achieve an expected specific practice an organization's adherence to the practice must be verified by an appraisal team through what is referred to as *direct* and *indirect* artifacts.

Direct artifacts relate to the real intent of the practice. Examples are requirements, design, code and test artifacts (Note here that the model doesn't dictate the form these direct artifacts must take, nor the order in which they are produced). Indirect artifacts are a consequence of performing the practice, but they are not the reason for the practice, nor are they required by the practice. Examples are meeting minutes and status reports.

But this is where I have seen many process improvement initiatives go off track. Too often I have found organizations being driven to perform unnatural and non-value added behaviors to produce unnecessary indirect OE, rather than maintaining focus on the real target, quality products produced for their customers (direct OE).

My point is that indirect artifacts (e.g. meeting minutes, status reports, and so on) are not required by the model. I am not saying that meeting minutes and status reports are not important, but I am saying you can use what are called "affirmations" during an appraisal instead. This means you interview people and they tell you what they do.

For example, the appraisal team can interview an agile team and the project team members tell the appraisal team what they do. As an example, a response from a team member might be, "We hold daily standup meetings where the team members report status and the team lead listens and then removes obstacles."

The appraisal team takes notes on what they hear (affirmations). These notes are then used as evidence that the organization is meeting the intent of many specific practices in the model. We don't have to force unnatural and non-value added behavior, and extra time-consuming work to create unnecessary artifacts to get ready for a CMMI appraisal. The direct artifacts and the affirmations are sufficient.

So why then do many large DoD contractors spend huge amounts of money to get ready for a CMMI appraisal? One reason I have observed is many contractors don't trust what their people will say in those interviews. They therefore force the creation of huge amounts of indirect evidence (e.g. meeting minutes, status reports etc.) to "reduce the risk" that someone might say the wrong thing and it might lead to an unsuccessful appraisal.

There is a lot of negative talk today concerning the effectiveness of the CMMI model. One of my government clients told me that he doesn't care anymore about the CMMI rating because he can't see the difference in the performance of an organization that says its level 5 from one that says its level 2. My view is that the problem isn't with the model, but the way companies apply it focusing on the rating, rather than its real intent. In my opinion agile methods can actually help us use the CMMI model as it was intended—for real process improvement.

Our philosophy, with the small agile company that had a business goal to achieve CMMI Level 3, was not to drive the team to perform any unnatural, or non-value added acts in preparation for the appraisal. The team's affirmations and the direct artifacts became the real proof – the products they produced, their satisfied customers, and their business success: the business is currently growing 30% a year.

## When To Stop Modeling (reprise)

If you are not doing any modeling at all on your project, there is a strong possibility that you can benefit by thinking carefully about the structure of your domain (the domain model). If you think that your models need to be complete, correct and true before you start coding, then you are almost certainly doing too much.

At the time of writing the first edition of this book, most people modeled either too much – because they thought modeling was good – or not at all, typically because they didn't think about it at all. Once agile development became fashionable, overzealous would-be agilists proclaimed that modeling was bad. Once again they gave incorrect advice.

Scott Ambler, in his book *Agile Modeling* (2002) set out to correct both imbalances. He describes, for those who model too much, lighter and simpler ways to model. For those who don't bother to model at all, he encourages lightweight modeling to assist in thinking and communication.

Scott's approach lends itself well to the text I wrote on page 37??:

> Constructing models is not the purpose of the project. Constructing a model is only interesting as it helps win the game.
>
> The purpose of the game is to deliver software. Any other activity is secondary. A model, as any communication, is *sufficient*, as soon as it permits the next person to move on with her work.
>
> The work products of the team should be measured for *sufficiency with respect to communicating* with the target group. It does not matter if the models are incomplete, drawn with incorrect syntax, and actually not like the real world if they communicate sufficiently to the recipients. . . .
>
> Some successful project teams built more and fancier models than some unsuccessful teams. From this, many people draw the conclusion that more modeling is better.
>
> Some successful teams built fewer and sloppier models than some unsuccessful teams. From this, other people draw the conclusion that less modeling is better.
>
> Neither is a valid conclusion. Modeling serves as part of the team invention and part of their communication. There can be both too much and too little modeling. Scrawling on napkins is sufficient at times; much more detail is needed at other times. . . .
>
> Thinking of software development as a cooperative game that has primary and secondary goals helps you develop insight about how elaborate a model to build or whether to build a model at all.

The difficult part of this advice is that it requires you to periodically stop, think about what you've been doing, and discuss with your colleagues about whether to turn up or turn down the dial labeled *modeling*. There is no correct answer. There is only an answer that better fits your situation than some other answer.

The answer is complicated by the Shu-Ha-Ri levels of the people in the room. Some *Ri*-level people will step to the whiteboard or CASE tool and model almost as fast as they can talk. Other *Ri*-level people will do the modeling in their spoken language, in their tests (using TDD or XXD[43]), and in their code. A *Ri*-level person's advice to a *Shu*-level person is likely to be to copy the Ri-level person's style. That may or may not suit the *Shu*-level person.

My suggestion is that *Shu*-level developers need to *model* and discuss those models with the most experienced people around them, in order to learn how to *think* about their models. As part of their professional growth, they can learn to model in both UML and in tests. CRC cards and responsibility-based design (Beck 1987, Cunningham ???, Cockburn ???crc url, CRC book, Evans book, WB book2) are very good ways to get started with learning to think about models.

The tangible artifact that exits the modeling activity can take many forms. snapshots of whiteboards, UML diagrams in a CASE to drawing tool, collections of index cards taped to the wall, videotaped discussions of people drawing and talking at the whiteboard, all these are useful in different situations[44].

Always remember to pay attention to both goals of the game: deliver this system; set up for the next round of the game and the next people who will show up for that game.

For comparison, here are comments from three other agile experts on the subject of how much modeling to do[45]:

> **Jon Kern:** "I am subconsciously stomping out risk to a given level so that I can proceed with some degree of confidence. If one part of the model remains pretty lightweight because you have done it before 100 times, so be it. If another part of the model got very detailed due to inherent complexity and the need to explore the depths looking for "snakes in the grass" -- well, that just is what it is!"
>
> **David Anderson:** "I think I'd stop - have enough confidence - when three features fail to add any

[43] XXD is described on page ???.

[44] I collected a number of photos and drawings as work samples in *Crystal Clear* (Cockburn 2005cc). There isn't space to reprint all those again here. The interested reader can look through those and other work samples there.

[45] All three from http://blogs.compuware.com/cs/ blogs/ jkern/archive/ 2006/03/29/Depth_of_Initial_Modeling.aspx

significant new information to the domain model i.e. no new classes, associations, cardinality or multiplicity changes. Not significant would be methods or attribute changes. When it looks like the shape will stand up to a lot of abuse without the need for refactoring then you are ready to start building code.

"However, in recent years I've done two layers of domain modeling. The first with the marketing folks to help flush out requirements and help them understand the problem space better. I have a lower bar for "done" in this early stage.

"The second phase domain modeling done by architects and senior/lead devs has this very well defined bar that the shape must hold because we don't want to pay a refactoring penalty later. "

**Paul Oldfield:** "I have similar experiences [as Jon] - I always know when I know enough about the domain to move on, and when I need to know more. The only time I have a problem is when I can't go back for more information later. For me the key question is, do I know enough to keep me busy until the next time I get contact? If I'm fronting for a team, the question is very similar - Do I know enough to keep the team busy until next time? "

See what your team thinks is the appropriate threshold for modeling.

## The High-Tech / High-Touch Toolbox

The editors at CrossTalk magazine asked me to write an article about what the agile "toolbox" contains (Cockburn 2005tools???). This turned out to be doubly interesting, because the agile toolbox contains both social and technological tools, and both high-tech and high-touch tools.

*High-tech* tools are those that use subtle or sophisticated technology. These include performance profilers and automated build-test machines, and high-touch social. *High-touch* tools are those that address social and psychological needs of people. These include daily stand-up meetings, pair programming, group modeling sessions and reflection workshops.

People often find technology depersonalizing, so we tend to see high-tech tools as low-touch, and tend to see high-touch tools delivered in low-tech. Thus, agile developers will often express an interest in using low-tech tools for their social sessions, such as paper flipcharts and index cards ("so we can *touch* them, *move* them, and remember the discussion by the stains on the napkins"). More recently, people are using high-bandwidth communication technology to increase the *touch* factor on distance communications (and indeed, the touch factor is one of the reasons people might prefer watching videotaped discussions over reading paper documents).

Agile teams use an interesting mix of high-tech and high-touch tools, and an interesting mix of technological and social tools.

The technological tools are interesting in their diversity. There are the well-known low-tech, high-touch tools: lots of wall space, index cards, sticky notes, flip charts and white boards. There are the well-known high-tech tools for automated testing, configuration management, performance profiling, and development environment.

There are automation tools that had little existence before the agile wave hit, but have become indispensable for the modern agile team. Foremost among these are continuous integration engines such as CruiseControl. These run the build every twenty or thirty minutes, run the automated unit and acceptance tests, and notify the development team (in all their various locations) about the results of the tests (often by announcing whose code failed). The continuous integration engine holds the team together both socially and across time zones.

There are the distance-communication tools. These include instant messaging tools with conversation archival, microphones and web-cameras to get "real-person" interaction, speakerphones for daily stand-up meetings across time zones, technical documentation on internal wiki sites, and PC-connected whiteboards to keep records of design discussions. All the above serve to fill the gap left by the absence of frequent personal encounters between the people on distributed teams.

Most interesting, however, are the specific inclusion by agile teams of social tools. The top social tools are to collocate the team and attack problems in workshop sessions. Other social tools revolve around increasing the tolerance or amicability of people toward each other, giving them a chance to alternate high-pressure work with decompression periods, and allowing them to feel good about their work and their contributions.

The following is the list of tools from the article:

- Social roles such as coach, facilitator, and scrum master.
- Collocated teams, for fast communication and also the ability to learn about each other.
- Personal interaction, within and across specialties.
- Facilitated workshop sessions.
- Daily stand-up status meetings.
- Retrospectives and reflection activities.
- Assisted learning provided by lunch-and-learn sessions, pair programming sessions, and having a coach on the project.
- Pair programming, to provide peer pressure, as well as camaraderie, better pride in work, and energy load balancing.
- A shared kitchen.

- Toys, to allow humor and reduce stress.
- Celebrations of success and acknowledgment of defeat.
- Gold cards issued at an established rate, to allow programmers to investigate other technical topics for a day or two.
- Off-work get togethers, typically a Friday evening visit to a nearby pub, wine-and-cheese party, even volleyball, foosball, or Doom competitions.
- Posting information radiators in unusual places to attract attention (I once saw a photo of the number of open defects posted in the bathroom!)

The interesting thing is probably less that agile teams do these things than that they consider them essential *tools*.

## The Center of Agile

What would it mean to get "closer to the center" of agile development? This is a question I often hear debated. The trouble is, there is no center[46].

Most development teams are so far away from doing agile development that it is easy to give them a meaningful, "more agile" direction:

- decentralize decision making,
- integrate the code more often,
- shorten the delivery cycles,
- increase the number of tests written by the programmers,
- automate those tests,
- get real users to visit the project,
- reflect.

The problem only arises when the team is doing relatively well, and asks, "What is *more* agile?", thinking of course, that *more agile* equates with *better.*

When we wrote the agile manifesto, we were seeking the common points among people who have quite different preferences and priorities.

- For some, testing was important;
- For some, self-organization was important;
- For some, frequent deliveries was important;
- For some, handling late-breaking requirements changes was important.

These are fine things to strive for, and they are typically missing on the average team. Once a team gets a threshold amount of each in place, however, it is not clear which is *most* important, which is the "center" of the word *agile.* None of those is *most* important across all project teams. Each team must decide what is most important for *it* to address at *this* time. Eventually, there's no point in asking "where is more agile from here?"

## How Agile are You?

Agile developers dread being asked how to evaluate how "agile" a project team is. Partly this is due to fear of management-introduced metrics. Partly this comes from an understanding that there is no "center" to try to hit, and there is therefore no increasingly high numeric score to target.

SCORING AGILITY POINTS

Much to my chagrin and frustration, and over my strenuous objections, the (really very experienced) Salt Lake City agile round table group decided to use the table in this section to evaluate the projects they were involved in. After they had assessed themselves for each property in the table, they asked: "But where's the final score?"

To my further chagrin and frustration, and over my further strenuous objections, they found ways to score numbers for each property (and argued over them, to try to raise their scores). After they had added up all the numbers and declared a winner, one of them said, "But this is no good – we got a high agility score and we haven't delivered any software in two years!"

Which, as far as I'm concerned, proves my point (see also my blog on the subject[47]).

Nonetheless, a manager, the sponsoring executive, or the team itself will want to track how it, or different teams, are doing with their move into agile territory. Scott Ambler wrote a short test that provides a good starting point (Ambler 2005.12). He wrote:

"My experience is that if the team can't immediately fulfill the first two criteria then there's a 95% chance that they're not agile.

They

1. can introduce you to their stakeholders, who actively participate on the project.
2. can show—and run—their regression test suite.
3. produce working software on a regular basis.
4. write high-quality code that is maintained under CM control.
5. welcome and respond to changing requirements.
6. take responsibility for failures as well as successes.
7. automate the drudGéry out of their work."

The following is a more detailed way to examine how you are doing. It came during a visit to a company that had a number of projects trying out different aspects of the agile

[46] Rather like certain old European cities. The center of the city is the "old town." But there is no center to the old town. It is just a maze of twisty little passages, all different. Once you are there, it doesn't make any sense to ask, "How do I get closer to the center from here?"

[47] See http://blog on agility points???(cockburn url)

approach. We characterized each project by how many people were involved and the project's iteration length. Since iteration length is secondary to frequency of delivery (see "How short should an iteration be?", page ???), we included iteration length only to characterize the projects, not to evaluate them.

We filled out this table, marking *how frequently* the deliveries, the reflection workshops, the viewings, and the integrations had happened. Osmotic communication got a ‘Yes’ if the people were in the same room, otherwise we wrote down the team's spread. Personal safety[48] got 'No,' '1/2,' or ‘Yes,’ depending on a subjective rating. (If you don't have personal safety on your projects, can you post a 'No'?)

For focus, we put down "priorities" if that was achieved, "time" if that was achieved, or ‘Yes’ if both were achieved. For automated testing, we considered unit and acceptance testing separately (none had automated acceptance testing).

We marked some of the times with '**!**' if they seemed out of range. Seeing a '**!**' in one place leads you to look for a compensating mechanism in another part of the chart. For example, both SOL and EVA had only one delivery, after a year. Both received '!' marks by that delivery frequency. However, SOL has a compensating mechanism, a user viewing every week. EVA, which had no compensating mechanism, did all the agile practices except getting feedback from real users (missing both deliveries and user viewings). When that company finally produced a product, the product was rejected by the marketplace, and the company went bankrupt (this should highlight the importance of "Easy access to expert users**.**")

Use the table in Figure 5'-17 to get all of the properties above the drop-off (see "Sweet Spots and the Drop-Off," p.???), but don't try for a summary score. Instead, use the table to decide what to work on next. I would, of course, post the chart as an information radiator and update it monthly.

[48] From *Crystal Clear,* p. ???: "*Personal Safety* is being able to speak when something is bothering you, without fear of reprisal. It may involve telling the manager that the schedule is unrealistic, a colleague that her design needs improvement, or even letting a colleague know that she needs to take a shower more often. Personal Safety is important because with it, the team can discover and repair its weaknesses. Without it, people won't speak up, and the weaknesses will continue to damage the team. "

| ***Project*** | ***EBU*** | ***SOL*** | ***EVA*** | ***GS*** | ***BNI*** | ***THT*** | ***Ideal*** |
|---|---|---|---|---|---|---|---|
| # People | 25 | 25 | 16 | 6 | 3 | 2 | <30 |
| **Frequent Delivery** | 2 weeks | ! 1 year ! | ! 1 year ! | 1 month | 2 months | 4 months! | <3 months |
| **User Viewings** | 2 weeks | 1 week | ! 1 year ! | 1 month | 1/iteration | 1 month | <1 month |
| **Reflection Workshops** | 2 weeks | 1 month | 3 weeks | No | No | 1 month | <1 month |
| **Osmotic Communica-tion** | 1 floor | 1 floor | Yes | Yes | 1 floor | Yes | Yes |
| **Personal Safety** | 1/2 | Yes | 1/2 | Yes | Yes | Yes | Yes |
| **Focus (priorities, time)** | priorities | Yes | Yes | priorities | priorities | Yes | Yes |
| **Easy Access to Expert Users** | No | 1 day/week | No | No | voice, email | Yes | Yes |
| **Configuration Man-agement** | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| **Automated Testing** | No | No | unit | No | No | unit | Yes |
| **Frequent Integration** | 3/week | 3/week | daily | 1/week | monthly | 1/day | continuous |
| **Collaboration across Boundaries** | Yes | Yes | No | No | ½ | 1/2 | Yes |
| Iteration length | 2 weeks | 1 month | 3 weeks | 1 month | 2 months | 1 month | <2 months |
| Exploratory 360° | No | Yes | No | No | No | Yes | Yes |
| Early Victory | Yes | Yes | Yes | No | No | Yes | Yes |
| Walking Skeleton | Yes | Yes | Yes | No | No | Yes | Yes |
| Incremental Rearchi-tecture | Yes | Yes | Yes | No | No | Yes | Yes |
| Information Radiators | Yes | Yes | Yes | Yes | No | Yes | Yes |
| Pair Programming | No | No | Yes | No | No | No | Maybe |
| Side-by-Side Program-ming | No | No | No | No | No | No | Maybe |
| Test-First Development | No | No | Yes | No | No | No | Maybe |
| Blitz Planning | Yes | Yes | Yes | No | No | Yes | Yes |
| Daily Stand-up Meeting | Yes | Yes | Yes | Yes | Yes | No | Yes |
| Agile Interaction De-sign | No | No | No | No | No | No | Yes |
| Burn Charts | No | Yes | No | No | No | Yes | Yes |
| Dynamic Priority Lists | Yes | No | Yes | No | 1/2 | Yes | Maybe |

**Figure 5'-17.** A way to track projects' use of the agile ideas.

## Introducing Agile

The question, "How do I introduce agile software development into my organization?" has no tidy answer. Here are three ways to rephrase the question to highlight the different unanswerable questions this one contains.

- "How do I get my company to change its corporate culture?"
- "How do I get someone else to change their way of working to match mine?"
- "How do I convince my boss that his way of doing things is wrong and mine is right?"

Eric Olafson, the CEO of Tomax whose move to agile development was described in Chapter 8 (see page ???), offered an excellent insight into the problem. He said:

"You don't *transition your projects* to agile projects, you have to *transform your people* to think in agile ways."

Transitioning projects you can put on a schedule. Transforming people you can't. Many executives since then have echoed Eric's insight.

I wish I could write that in the last five years I have seen enough organizations move to agile development to formulate a good success strategy. Unfortunately, I haven't, and have, instead, become wary of large organizational initiatives. My own skeptical phrasing is:

"For any organization-wide initiative X, X gets rejected by the organizational antibodies[49]."

Agile development is just one value we can plug in for X.

The failure often comes from a backlash effect that shows up *after* one or more projects have delivered successfully – and sometimes even *because* the project delivered successfully. Apparently, simply delivering projects successfully is not sufficient to make a change stick[50]. For more discussion on organizational change, see "Business as a Cooperative Game" on page 11.

Can we say anything about introducing agile development into an organization? The book *Fearless Change* (Rising 2003???) contains some observations from change groups. Here are some starter do's and don'ts from other teams. Bear in mind that many of these ideas help you get started, but none is adequate to make the change stick.

*Don't* make an executive proclamation that all development will be done using the agile approach and then hire training for everyone in the company. This approach will burn up a lot of money quickly, and fuel the resistance movement from the beginning.

*Do* seek support at both the highest and lowest levels of the organization. The programmers, at least, have to want the change, and they need the input and support of some of the executive team. Either group without the other will become isolated and unable to proceed.

*Do* get one project team working successfully.

*Do* staff that team with at least two people who are competent, respected, and can pull off the agile approach.

*Don't* shower that team will an undue amount of support and attention. The other teams will just feel left out and get annoyed at that team.

*Do* get experts in to help. Place them *in* the teams, not outside as advisors.

*Do* consider the users and the executive sponsors part of the team ("There's only *us*").

*Do* reflect on what you are doing each month, and change your working convention to get better. If you are not changing something, trying out something new, each month for the first few months then you probably aren't getting a taste of it yet.

*Do* deliver real software to real users, quarterly, or better yet, monthly.

*Do* prepare for a backlash to come from somewhere you can't see.

Finally, one word of comfort: The higher level executives probably want what the agile approach offers, which is

"Early and regular delivery of business value."

I have not yet met an executive who finds that a troubling proposition. They usually agree that the development process needs lightening, testing needs to be beefed up, and users more involved. The increased visibility given by agile status reports are a comfort to them. They appreciate being able to make mid-course corrections. Early and regular delivery of business value helps the bottom line.

[49] Thanks to Ron Holiday for this outstanding concept ("organizational antibodies")!

[50] This is a good time to reread Gerald Weinberg's *The Psychology of Computer Programming* (1999). His writings contain many stories in this vein.

## Introducing Agile from the Top – 10 Lessons, by Eric Olafsson, CEO, Tomax

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

## Agile Outside Software Development

Agile software development calls for customer collaboration. Collaborating actively with the customer will immediately bring into discussion the company's contracts, sales, maintenance and communications relationships – in other words, the whole business. A company that takes the agile model seriously will have to make adjustments in all those areas.

Fortunately, the agile approach works well in most business environments. Let us look at a series of cases starting from the software arena and moving outward to business in general.

### Project portfolio management.

Any large organization has a portfolio of projects underway at one time. The management team should balance the effort being spent on each project on a quarterly basis, if not more often. They run into two problems: staff specialization and finding stopping points.

In most companies, one person becomes the expert on one piece of code. Every time a function is changed in that piece of code, the organization is dependent on that one person to do the work.

One of the characteristics of agile development is increased communication, hopefully with osmotic communication. In some cases, as with XP shops, programmers work in pairs, and change programming partners frequently. This means that more people learn to work in each part of the system. With rotating pair-programming partners, the organization can shift a programmer to work on a more critical project when needed, with (possibly less qualified) people able to work on the other code.

Even without pair programming, it is still a good idea to cross-train people on each other's code bases. The understudy may be less proficient, but will still be able to carry out the needed work.

The second problem with project portfolio management should be easier to deal with. With properly executed agile development, the system is put into a stable, shippable state every week, month, or quarter. At those points, the people on the project can be moved to other projects as needed. There is a danger of thrashing – moving people back and forth so often that they lose track of, or interest in, what they are working on – but there is at least the option of rebalancing the project portfolio to reflect the organization's current priorities.

With properly executed agile development, project teams will be working on the most valuable features of each system[51] in each increment. This means that when a project is trimmed or put on hold, it is always in a useable state. It might happen that the organization never restarts the project, or changes its direction. These are both valid outcomes of project portfolio management.

There is, of course, one remaining problem: the difficulty in obtaining meaningful market data to decide which features to move forward into early increments, and which to move back or drop. I have not yet met any market analysts who can provide comparative market forecasts for products down to the feature level. Real project portfolio management requires that the management team be able to compare cost and benefit numbers at the feature level, in order to decide how to best balance their use of scarce resources in development.

It probably will take a combined group of senior developers (to provide the cost numbers at the feature level), market specialists (to provide the return numbers at the feature level) and creative management facilitation to engage the group in discovery of interesting strategies for development sequences, in order to come up with good answers to the project portfolio question. Note, as before, the importance of keeping the cooperative game and "there's only us" mindset.

### Customer Relations

Companies that do custom development may have trouble with agile development because their relationship with their customers doesn't support the point of view that "there is no *them;* everyone is *us*". When the contracted company proposes that the customer visit the project weekly to provide input and steer the priorities on the project, the customer replies, "But that's what we hired *you* to do!" This is a lose-lose worldview. The customer doesn't get the best system they can, and the provider adds risk to the project.

Experienced readers will recognize that the agile model of customer relationship matches that coming from Toyota and lean development (toyota 1995?). One would therefore hope that close customer collaboration would be easy to achieve. However, my reading of the lean literature doesn't make me very optimistic on this score. Toyota's customer- and vendor relation model has not penetrated the minds of most executives even after decades of publicity and exceptional results. I can only think that every

[51] See the discussion of work-sequencing strategies (Cockburn CC 2005, pp. ??) and the use of the "iceberg list" (Cockburn bc URL 2004) for a more complete discussion of this idea.

additional push from the agile side helps. Tomax's results (see next sidebar) are heartening, because they show that customers can become used to, and then *expect*, to participate in steering their projects.

## Contracts

Companies that do work on contract are always faced with the problem of price, scope, and time (this applies to all industries). Clients often want all three to be fixed, to reduce their own risk. I see only (to date) only a few options:

*Fixed-price, fixed-scope (and possibly fixed-time)*

This option is the old one: Do your best, however you do it, to come up with a number for the project. Once inside the project, squeeze all the fat out of the process, using all the best agile techniques you can muster (see "Process, the 4th Dimension" (Cockburn 2005 Pt4D)) to deliver it as well as you can. As far as I can tell, agile techniques do not change your ability to estimate a project. Despite our most fervent wishes, estimating a project is still a matter of previous experience, feel, and guesswork. Fixed-everything projects make sense when the requirements are very stable. They are also used when the two parties don't trust each other, even though there is usually plenty of leeway for one party to twist the situation to the other's disadvantage if they want.

*Fixed-price, fixed-scope (and possibly fixed-time) but collaborate with the customers to alter scope anyway*

A risky approach, I have seen it done successfully several times, twice for commercial projects and once for a state government project. Jeff Patton (Patton 2004) describes working closely with the customers at the start of the project to identify needs, wishes and priorities. As the project proceeded, Jeff made sure they were always working on the most important items. As –they got close to the end of the time period and it was clear they would come up short on features, he made sure that the items left undone were clearly the least important ones. Those the customer either forgave or rolled into a follow-on contract.

The government contract I tracked was again a fixed-everything contract. The team worked closely with the users of the system, delivering and installing running, tested, usable software every month. After each delivery, they discussed with the users whether to continue with the contract as written, or change directions based on what the users really needed. I found it remarkably brave of the contracted company to follow the users' requests in deviation, because in a hostile sponsorship situation, they would have been at risk of non-payment. However, the user base was so happy with the usefulness of the functionality delivered that the contracted company got paid in full. This was a case of converting a fixed-price, fixed-scope, fixed-time contract into a time-and-materials contract on the fly. A dangerous but wonderful strategy if you can accomplish it.

*Time and materials*

This is simply a matter of paying for work as it gets done. It is obviously the best format to use if the requirements are volatile. In one case – I won't even call it a project – the sponsor changed the requirements wildly, even up to days before delivery. The contracted company always delivered whatever they had running at the end of the quarter, and the sponsor always had complete control as to what was on the top of the priority list. Over a period of years, it became simply a balanced flow of useful software in one direction and money in the other direction..

If the sponsor trusts the contracted company, this is the simplest contract to use. The difficulty is that it requires a certain level trust. What some contractors do is to request a trial period. Using the agile approach, they deliver usable functionality early, and so establish the trust of the sponsor for subsequent work (see the house construction example on page 95???)

*Not-to-exceed with fixed-fee (NTE/FF)*

This is the contract method used in the airport construction example on page 97???. It presupposed stable requirements. "Fixed fee" means that the contracted company is guaranteed a certain profit margin over their materials and subcontractors' fees. This protects the contracted company in case requirements are reduced or the work goes faster than estimated. "Not to exceed" puts a ceiling on the total amount paid to the contracted company, which protects the sponsor in case the work goes slower than expected. With protection for both sides, both sponsors and contracted company can look for ways to speed the work, and live with unfortunate events. The airport construction project on page 97??? used this form with *Payment on incremental acceptance*.

*Fixed price per function point or story point*

When the customer and contracted company can agree on a unit of delivery, such as function points or story points, then they can settle on a price for each unit delivered. A number of contract houses these days estimate the size of a project in function points, create a price per function point delivered, and then have an certified function point auditor assess the actual number of function points delivered in the end. The customer then pays that amount, not the originally estimated amount. This is a nice contract form, because it encourages the contracted company to work more efficiently (to increase their pay per delivered function point) and it allows the customer to change the requirements along the way.

The same can be done with XP-style story point (Cohn 2005), except that there aren't any certified story point examiners to judge the final result. Bob Martin of Object Mentor posted an interesting variant to get around this problem: a base fee per story point, plus a lower-than-usual (close-to or below cost) fee per hour. This biases the contracted company's to deliver early, but gives them some protection in case work proceeds slower than expected. Bob Martin described it this way:

> "[A]gree to pay a certain amount for each point completed, plus a certain amount for each hour worked. For example, let's say you've got a project of 1000 points. Let's also say that a team of four has established an estimated velocity of 50 points per week. This looks like about an 80 man-week job. At $100/hour this would be a $320,000 job. So lets reduce the hourly rate to $30/hour, and ask the customer for $224 per point.
>
> This sets up a very interesting dynamic. If the job really does take 80 man-weeks, then it will cost the same. If it takes 100 man-weeks then it will cost $344,000. If it takes 70 man-weeks it will cost $308,000. Notice that this is a small difference for a significant amount of time. Notice also that you, as developer feel strong motivation to be done early, since that increases your true hourly rate." (Martin 2006)

I have not seen that model in action myself, but several people have written in recommending it.

*Venture-capital financing model*

This can be used with any of the above contract forms. In this model, the sponsor gives a round of financing for a certain amount of work, and the contracted company must produce results in order to get more funding. The sponsor can cut their losses at any time if they are not getting the results they need. They can presumably alter the terms of the contract after each work period. The result of a work period need not be working software; it could be a paper study, or a requirements document, or anything the sponsor selects.

The venture-capital finance model works well with agile providers, since the agile provider is used to delivering useful, working software early and regularly.

I find it an odd irony that the venture capital financiers running start-ups that I have encountered don't take advantage of their own model to the extent agile teams do. The venture financiers let the evaluation markers occur too far apart in time. If they attached funding to monthly releases, that would oblige the start-up team to think through what it really can accomplish each month. The monthly progress would give the financiers a better sense of the start-up company's real progress.

*Incremental delivery with payment on incremental acceptance*

This can be used with either fixed-everything contracts or the NTE/FF contract. The sponsor expects incremental delivery of integrated, tested systems, constructs and runs acceptance tests at each delivery increment, and pays the contracted company after each successful incremental acceptance. This was used in the airport construction project (p. 97??) and in Solystic's French Post Office contract (p. 101???). Both project leaders told me that this payment schedule has a wonderful motivating effect on the contracted company. The requirement, of course, is that the sponsors have an understanding of incremental development and can make acceptance tests for the incremental deliveries.

Here is a sidebar about revisiting the custom contract, by Eric Olafsson, the CEO of Tomax (who also provided the sidebar on page 85???). Tomax produces customized retail systems.

A few years ago, Tomax worked to fixed-everything contracts. They were, of course, problematic, since something was certain to change even within a simple, three-month contract. Eric used the agile model to change the company's relationship with its customers. They now writes contracts so that the customer gets early results through incremental development, and can cancel or change direction after each increment. In exchange, the customer promises in the contract to make its experts available to the development team and to actively steer the project's direction.

The contract converts the customer from "them" to "us", part of the larger development team. The customer gains early returns and greater steering control. Tomax reduces its pricing risk and get better input from the customer-side stakeholders and users.

## Revisiting the Custom Contract, by Eric Olafsson, CEO, Tomax

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

about 2 pages

## Introducing Change Into An Organization

Introducing change into an organization is scary and difficult. Agile development is doubly scary because it says:

- First, change the way you do things;
- Next, keep changing, *forever*.

This puts agile development fully within the topic of introducing change into organizations, a very difficult topic, indeed.

The noted family psychotherapist Virginia Satir studied, among other things, people's reactions to change. A key observation of hers was "*if there's ever a question between comfort and familiarity, familiarity will almost always win out.*" (Satir 1999). That is, even if the familiar mode is inefficient and uncomfortable, people stay with it.

### *The Process Miniature*

A useful corollary of that observation is that things are easier the second time – they are more familiar. This shows one possible way out of the unfamiliarity trap: the *Process Miniature* (Cockburn CC 2005).

A process miniature is nothing more or less than going through the new procedure in as short a time as possible: a minute, 15 minutes, an hour, a day or a week, depending on the process.

- We can work through a simple example of the *planning game* (XP planning game ref???) or the *blitz planning* technique (Cockburn CC) in 15 minutes.
- In the "Extreme Hour" game[52] people run through two iterations of XP in one hour. There is a one-hour version used in showing people how Scrum works. I created a theatre piece of Crystal Orange Web (see p. ???), running through two iterations of delivering real code on their system platform in 90 minutes[53].
- In my use-case course, I run a one-hour process miniature of gathering use cases, from defining system scope to stakeholders' interests, naming all use cases, and writing defining one use cases completely, to show how all the parts of the process fit together.
- One company had every new employee go through a full development and deployment cycle in one week to learn their company's process. After having completed that initial mini-project, the new employee could be placed on any project, in any stage of development, and would have an understanding of what was going on.

Going through a process miniature creates in the mind what Grady Booch neatly but mysteriously called a *gestalt round trip* (ref?). After the gestalt round trip is completed, the process exists as an entirety in the mind, can be viewed internally, and important for us here, is a bit more familiar and less frightening.

### *Anchoring New Behaviors*

The process miniature also creates what sociologist Karl Weick calls a 'small win' (Weick 1975??). Weick found that small wins strengthen a group's strength and self-image. Good groups look for early, small wins to strengthen their sense of team accomplishment. One of these that system designers use is the *Walking Skeleton* architectural strategy ("a tiny implementation of the system that performs a small end-to-end function" - Cockburn 2005 cc p. 55???)).

One of the surprises to me was that people get mileage out of small rewards, even very small rewards. Evidently, these also count as small wins in the mind of the receiver.

I was on the receiving end of this technique one time: A colleague visiting my house saw how my assistant had created a "project wall" showing all my ongoing projects and trips. He said, "Wow, that's great! Here, I'll give you a quarter for showing me such a great idea." A few minutes later he found something else that he liked and gave me another quarter. I found myself irresistibly drawn to offering him ideas and looking to see which he thought was worth his little token offering. I managed to get six of those quarters by the end of his visit, and I still keep them in a special place as a memory of his appreciation.

You might not think that a token as small as a sticker would have any impact on people's behavior, but just the act of giving and receiving a token of appreciation or accomplishment has meaning to the receiver. Here is Kay Johanssen's story of using stickers to help create change:

**Creating Change with Stickers, by Kay Johanssen, Test Manager**

Our company's product was a variety of internet services not amenable to commercial automated testing tools, so we needed a programming-intensive approach to automating the tests. As Test Manager, I put together a team of five testers, most of whom had some coding background.

Frustratingly, we made no progress on automation. Every day, we came to work, intending to program a new automated test, but we could always see the backlog of material that needed testing *right now*, manually. Being good citizens, we performed those manual tests instead of starting programming.

I knew something had to be done, or we'd never get ahead of this mountain of manual work. Since we were running iterations with XP-like "stories," I created stories for a test framework. After a few iterations, we had

52 http://c2.com/cgi/wiki?ExtremeHour

53 http://c2.com/cgi/wiki?ExtremeHourWithActualProgramming

hacked together a framework and about three tests. That was as far as it went for a while. The mental switch required to figure out how to write effective automated tests seemed too great. And the mountain of manual testing was always there.

My first strategy was posting posted a large, visible chart with the number of tests on it. The number was three.

I waited for it to increase. We made little progress during the next couple of months. Interest seemed low. I created stories for writing tests, and the team would sometimes obligingly force a test out, or as often as not, fail to complete their automated test story because they spent extra time doing a really good job on their manual testing stories. They were looking out for the customers, and I couldn't criticize them for that. Still, something needed to be done.

One day while at the grocery store, I noticed a package of shiny stickers, I think they were colored stars. On a whim, I bought the package and took it to work. At the start of the next iteration, I announced that anyone who completed an automated test would get a sticker. That caught their interest. After they finished laughing, they signed up for the few automated test stories with alacrity. My impression was that they finally felt they were not being perceived as "selfish" if they signed up to write automated tests – they had permission to do what they wanted to do anyway. Even better, they were competing, playing the game!

Over the next few weeks, the chart of automated tests shot up, even though there were no "prizes" for "winning" and no concept of an end date or final counting. The team was energized and enjoyed a spirit of friendly competition and cooperation. I did nothing more than count tests and hand out stickers.

Some team members collected stickers on their monitor for all to see. Others pasted them on 3 by 5 cards so they could carry them to meetings to better show them off. The tester who had no coding experience got a lot of help . . . and ended up with the most stickers.

Before long, an online, automatically updated, full-color chart replaced my hand-drawn big visible chart. Test problems and framework problems got solved quickly. Soon we had a robust framework that even included a test script recorder.

The team grew confident, and set a goal for 550 tests by the end of the year, which they easily achieved.

I eventually stopped handing out stickers, but the energy and confidence continued. One year later the team had well over 1,000 automated tests.

### *A Large Changes is Many Small Ones*

If people are afraid to make small changes in their habits, they are more afraid of large changes.

Mike Collins, an expert at lean development, addressed this problem in an interesting way at the O.C. Tanner Manufacturing Company in Salt Lake City (see the next sidebar). In reading the sidebar, you many note that his method draws on Satir's idea of growing familiarity and Weick's idea of small wins.

Mike understood that if he showed up at their custom-awards manufacturing factory with a design for a ten-fold improvement in their system, he would encounter resistance from employees and management. Instead, he created a new production area with a minor set of modifications, from which they got a noticeable but not revolutionary improvement. He posted the results, reflected with the people involved about possible improvements, and created a second trial production area that was both smaller and faster.

When I first visited O.C. Tanner, they were using their third new production area. The original and the first trial production areas were still in use. The new one used about a quarter of the floor space of the original, and filled contracts in about a third the time of the original.

At the time of this writing, the entire facility has been converted to fourth generation 'mini-factories' as they now call them. Production lead times have been reduced from over 18 days to about 6 hours, with a three-fold increase in efficiency. With the smaller floor space, production workers get osmotic communication. An information radiator at the entrance to their production area lets them – and their managers – see what is being worked on as well as the productivity effects of the improved system. They expect to double their efficiency again during 2006.

What I found impressive was not so much the improvements the team had made (which Mike might have been able to accomplish from his prior experience and lean manufacturing theory), as his patience in working through a sequence of small changes and victories rather than going for one big win. I note in particular:

- Each process got *good enough* business results to pay for itself as it proceeded.
- Since the workers in the production area were invited to help set the direction of improvements, there was better buy-in from the workers.
- Both management and workers get used to having an evolving production process as a normal part of their working climate.

Mike and I both noticed that he would almost certainly have met too much resistance to succeed if he had proposed the third or fourth layout at the very beginning.

Notice in the business literature the frequent reports of a company achieving breakthrough results by making many small changes and reflecting on those changes. The latest I read was about the Outback Steakhouse soliciting ideas from employees, trying them out in selected restaurants, and then deciding which ones to franchise.

These ideas tend to come from within the organization, rather than from an outsider. This is the "we are tuning it for us" principle, something I consider a critical success factor for the adoption of new processes.

There is one other factor that assists in installing change: a threat to the company's survival from an external source. This is covered in the next section ("Programmers Read the Harvard Business Review").

**Lean Manufacturing at O.C. Tanner, by Mike Collins, Vice President - Lean Enterprise Development, O.C. Tanner**

The O.C. Tanner Company produces and ships about 9,000 employee recognition awards a day. Our work can best be described as 'mass customization,' in that we provide several million combinations of awards with an average order size of 1.3 awards per order (91% of the orders are for one award).

Five years ago, we started down the lean manufacturing track. Over those five years,

- the time from when an order is placed until we ship the first piece has dropped from 18 days to six hours,
- quality has risen from a low of 80% to 99%,
- on-time delivery has risen from a low of 60% to 99%,
- work-in-process (WIP) has dropped from almost 500,000 pieces to less than 6,000.

WIP used to be so large partly because we used large batch sizes between stations, but also because we ran production quantities at 10% or more over the actual order quantity to compensate for the quality problems. The large WIP meant we had people working full-time to move, find, expedite and otherwise manage the WIP pieces.

To measure our production improvement, we use a Quality-Efficiency-Delivery (QED) metric that averages improvements in quality, efficiency, and on-time delivery of the current year over the previous year. The improvement score was 34% in each of the last two years. This is a notable achievement given that current performance levels with both quality and on-time delivery are near 99%, with efficiency nearly 3 times better than it was 5 years ago.

Even though virtually every system, process and practice has been re-thought, the most critical aspect of the change has been a change in how the people think about their jobs and the company. Changing people and culture is much more difficult than changing systems, processes and practices. In fact, to effectively change systems, processes and practices requires that people and culture be changed first or at least in parallel.

Most of the employees had been with the company for many years and were well entrenched in the routines of their work. The company was reasonably profitable with good employee benefits and few saw any reason for change.

Initially, production consisted of 28 departments with each department working more or less independently of the others. Within each department, employees generally had only a single or perhaps two job skills. Work was passed in batches from employee to employee and from department to department. Most employees focused on their single skill job and few of the employees made any suggestions for change and even fewer ever participated in any change effort.

To make the progress realized to date has required several critical changes:

- Teams had to be implemented in which the collective creative thought of all can be harnessed.

O.C. Tanner went from 28 production departments to one department of 11 teams. Each team is in and of itself a self-directed mini-factory capable of producing every award combination.

Working as a team member is not natural for those coming out of many of today's school systems. Think about it. School is largely about competition, about getting the best grade. It means that there must be both winners and losers. If, however, a team is to be successful, the focus must change to making every team member a winner. Working as a team member requires that we greatly improve many of our skills, such as our communication skills. This change in thinking is not an easy change.

- Management had to move out of the management mindset and more into a mindset of leadership.

The job of management job is now much less of the day-to-day management of work and much more one of facilitating, teaching and supporting the efforts of the teams.

The first change to management's thinking was to learn that teams not only can, but most often will make better decisions than the manager. Most decision-making and creative improvement therefore had to move to the team members.

Because the manager often has had greater experience, he or she will see that the team on occasion is making a mistake. It is leadership that allows the mistake to be made, recognizing that we learn mostly from mistakes.

Sound judgment is of course necessary. The manager must balance the potential learning that comes through a mistake with the potential damage to the company. This change in thinking is also not an easy change.

- Team members had to learn multiple job skills, and were expected to participate in the many of the improvement efforts of the company. Most employees rotate between assignments during the day.

As the team members learn to do this, they find that they are more self-motivated, more involved and more satisfied with their work experience. The jobs are less mundane.

Almost all of the employees will tell you that they would never go back to the old system.

- Training was a key factor.

Training meant not only job skill training but also training in many other areas, such as team skills, communication skills, creative problem solving, coaching, conflict resolution, etc. Training included hands-on efforts at making improvements and solving problems, and even the understanding and use of statistics, which is very often necessary to the making of good decisions.

At first one may even question the wisdom of making these changes. O.C. Tanner is learning, however, that it is only with time and it is only by doing that an organization can begin to see the benefits of staying on the lean track.

The change rate at O.C. Tanner is accelerating. We expect efficiency to double again in the next 12 months. Leadership, instead of continually pushing the company's employees for more, is now finding itself in a position of not being able to support or even stay on top of the amount of change now taking place. It is an enviable position.

## Programmers Read the Harvard Business Review

An odd cultural shift has been the increasing number of programmers and testers who are reading the Harvard Business Review (HBR). This is a direct consequence of their watching the business consequences of their frequent deliveries and their prioritization choices. Enough technical leads have written to me to alert me to some article in the HBR that I ended up subscribing to it myself, just to keep up with what the agile development community was discussing.

As a sample example, the April 2006 issue (the time of this writing) contains two articles relevant to the topics in this book.

Ram Charan's article, "Home Depot's blueprint for Culture Change" (Charan 2006) details Home Depot's massive cultural change process as it went from a decentralized, entrepreneurial set of stores that could not or would not collaborate, to a centrally orchestrated set of stores that did collaborate. The new CEO, Robert Nardelli, was brought in from GE exactly to make that sort of cultural change.

Nardelli had one force on his side – their competitor, Lowe's, was gaining quickly, and it soon became clear that Home Depot's survival was at stake. Nardelli used that force to make many changes very quickly. While hard, it had the advantage that results also showed up quickly.

When people complained about the pace of change, Nardelli was able to answer back, "Good point. Give me five minutes. I'm going to go call Lowe's and ask them to slow down for us."

This sort of external threat helps a lot in installing change.

Key to their change process was the institution of metrics. I found it interesting to read that metrics were used to *increase* collaboration, the "we're all us" mindset:

"At the same time, the metrics made clear and reinforced the collaborative behavior and attitudes that Nardelli and Donovan wanted to encourage. . . . Companywide metrics also provided a platform for collaboration."

They used the metrics to show that they made a greater profit when working together than working separately or competing amongst themselves.

They instituted

- an annual strategic conference,
- learning forums for managers in which they ran simulations of various collaborative and competitive strategies, and
- weekly conference calls among the top executives around the nation.

Alert readers will recognize all of these as staples of the agile approach.

I noted two more key points in the article, the first concerning how long it takes to get a cultural change to stick, and the second about the importance of suggestions from inside the team:

"A year and a half after Nardelli took over as CEO, he and Donovan knew that there still was significant opposition within the organization to the changes they were making."

"Assuming the rate of change is more or less right, how do you make it stick? . . . Where possible, get people affected by a change to help define the problem and design the solution."

Note how similar these points are to those raised by Eric Olafson and Mike Collins in their sidebars.

The same issue of the HBR held an article about global localization (Rigby 2006):

"Standardized offerings discourage experimentation and are easy for competitors to copy . . . Customization encourages local experimentation and is difficult for competitors to track, let alone replicate. When well executed, location strategies can provide a durable competitive edge for retailers and product manufacturers alike."

". . . successful localization hinges on getting the balance right. Too much localization can corrupt the brand and lead to ballooning costs. Too much standardization can bring stagnation . . . "

The recommended strategy is to standardize within a related cluster of situations, and localize separately in each cluster.

This strategy matches the discussion earlier of tailoring methodologies to projects (see "Methodologies across the organization", page 24???).

Several organizations I have visited came to the conclusion that they could benefit from three base methodologies (for each major project type they encountered), and they should then tune the base methodology locally on the project. This allows them some scale advantages and some tailoring advantages.

There are two points I wish to draw from the above:

- Modern agile programmers *do* care about how their work influences the organization, and are studying business management within their professional development as programmers.
- Agile development practices are applicable to business management life in general, and vice versa, as

evidenced by writings in professional business writings.

## House construction

Many people argue that house construction isn't (or is) like software development. Since I have been making extensions to my house for several years, I now have some first-hand evidence to offer[54].

You can probably imagine how I would go about making house extensions. They should be done incrementally, using time-and-materials (trust-based) plus venture-capital (cut your losses) funding, and in a close collaboration with both the architect and construction contractors. You would be right.

In our first conversation, the architect suggested creating a "master plan" of all the desired changes, and then doing the work incrementally. I declined, because I was pretty sure that we would change too much for the initial plan to have meaning by the half-way point, and it would end up being a waste of my money. As events transpired, I was correct.

We chose to run each idea as a stand-alone project, but were careful to ask at each key moment, "If we were to extend in [some particular way] next year, how would that affect our decision now?" Surprisingly, we found no cases where the future choice affected the present decision, including putting a partial basement under the house.

Our first project was to put a basement under our house[55]. This turned out to be easier than expected, because our house was built on short stilts, to provide an insulating air cushion over the ground.

Rather than excavate the whole basement, we decided to excavate only 1/3 of the basement. This gave us the basement entrance we needed, and left open the question whether we would extend the basement or build a side wing to the house (in the next project). We learned that there would be no cost savings for excavating more than we needed now, even if we chose to expand the basement later. The XP community calls this the YAGNI principle, for "You Aren't Gonna Need It". We excavated only what we needed. Three years later we still have no plans to extend the floor space, either above or below ground. YAGNI is holding.

The architect and the contractor looked underneath the house to see how to support the house while digging underneath it. The architect launched into a dissertation on the various choices. The contractor cut in with, "Why don't we just run a giant beam right down the center and hold the whole thing up while we excavate?" Note here the application of the 11th principle of the agile manifesto:

*"Simplicity—the art of maximizing the amount of work not done—is essential." (see p. 371???)*

The architect adopted the suggestion, and after that, the architect, the contractor and the construction crew had excellent conversations that merged their best ideas. To this day they always trade construction ideas back and forth along with how those might change the design of the house itself.

A surprise hit us on the first day of construction. They knocked a hole in the concrete wall where they were going to excavate, peeked in, and discovered that the beams under that part of the house ran perpendicular to those where they had looked under the house. This ruined their plan for the support beams, and showed that once again, civil engineering is ahead of software engineering. It usually takes software teams a week or two to invalidate plans. These people were able to do it within hours.

As we progressed I quickly noticed that the contractor kept changing his "fixed-price" bid for the project as new information appeared. I finally suggested we drop the fixed-price façade, and just work to time and materials. His reply surprised me: "If you are willing to carry the extra risk, that will be fine with us." I said, "I don't see any extra risk. You'll just change the price whenever something changes anyway." He said, "That's true but most people don't recognize that." In exchange for my accepting the "extra risk," he lowered his profit margin from 13% to 10%. We each felt we had benefited from this change.

With the new contract in place, his crew was able to do any number of other small projects for us, such as changing the kitchen counters, adding closets, and fixing the fence around the yard. Neither these, nor the other usual surprises, twists and turns on the project worried either of us any more.

I noticed with some interest that he and his crew held a daily stand-up meeting[56] each morning to set their day's work goals. In this short meeting, they checked that they knew what they were working on, and had the materials to get it done.

The project came to a successful conclusion, and we gave a small bonus to the contractor and the workers. Unbeknownst to me, our prompt payments made us preferred customers, which proved useful for the next projects.

The second project was to sculpt the yard, move in stone slabs and do some general landscaping. Although this was a "minor" project, the contractor was delighted to

[54] Yes, I know this is not new house development. I know others who use the *Walking Skeleton* (Cockburn 2005) and related agile strategies even with new house development.

[55] I am told that people usually put the basement in first. Putting it in last is an unintended but amusing echo of the XP strategy to plan a project as though features can be implemented in any order.

[56] Part of the Scrum methodology (Schwaber 2005), also used in XP.

help. He sent digging and moving crews to help when we needed it, and even operated machinery when his key men were ill. This was part of us being preferred customers, a roll-forward benefit from the first project.

The third project was to add a two-story porch to the side of the house. At this point, we were glad not to have done a master plan, because this addition was completely different than anything we had originally had in mind.

By now, there was trust and good communications between the architect, the contractor and us. The small wins with reflection, the venture capital funding model, the willingness to dance with the situation, were all paying off. The contractor passed along our preferred customer rating to his suppliers, so we got extremely fast service. He also took it upon himself to see that they gave us good rates, so we saved money.

Since we contracted by time and materials, it was natural to have the subcontractors do additional projects around the house as we needed it. The final bill for all this extra work was much lower than it would have been had we contracted each one separately.

While the contractors are finishing the side porch, the architect is starting on a redesign of the front porch. His design is quite different than what he originally proposed when he was enamored of his master plan. That is because we now have all the other major changes to the house in place, and he can see how to tie them together in the front approach.

The point of this long story is to illustrate how the lessons from agile software development transfer to a completely different field. Incremental development, architectural changes, daily standups, YAGNI, time-and-material contracts, venture capital funding, customer involvement and steering, small wins, process miniature, and developing collaboration and trust – each of these proved useful.

## Airport construction

There might be those of you who think that house construction is too easy. I was delighted to meet an architect who uses the agile processes in airport design and construction.

On a flight to Boston, I sat next to Joe Wolfe, who answered, in reply to my question about his occupation, that he designs airport terminals (Terminal E in Salt Lake City and the new Delta terminal at Logan Airport). Not being able to resist, I asked how he worked. His reply astonished me, because he recounted almost point for point the Crystal Clear methodology – except he was doing it with several dozen subcontractors on airport terminal construction! I showed him my draft of Crystal Clear, including the *blitz planning* technique, and he said, "Oh yes, that's what we do."

At the beginning of the project, he collects the subcontractors in a room, and has them brainstorm the work they will have to do. They write their tasks and time estimates on sticky notes, and post them on a large wall. They sequence the tasks on the wall until they have a sensible starting working plan that shows both tasks and dependencies. Someone transcribes the contents of the wall into the computer after the session.

- I asked Joe whether he had a notion of *early integration* in his project, and how he would motivate subcontractors to do that. He said they did do that, that the subcontractors get paid at integration milestones, and the cash flow motivates them.
- In reply to my question about contracts, he introduced me to "not-to-exceed plus fixed-fee" contracts (see "Contracts", page 91???) "Not to exceed" means that the subcontractor promises not to exceed a certain final amount. "Fixed fee" means that the subcontractor is guaranteed a certain profit if the work takes less time than expected. Thus, the subcontractor's profit is protected, and the airport's spending bill is protected.
- I asked him about the agile notion of exposing bad news early – how would he get subcontractors to reveal their problems early? He said that incremental funding motivates them to integrate their work, and also to seek help when they ran into trouble. An example might be an international shortage of steel.
- I was concerned that even with those, subcontractors are not used to exposing their problems to the people hiring them, and it must take some time to get them used to it. He highlighted that he, as project coordinator, had to deliberately build a climate of trust within the team, so that they would reveal their problems.

Note the similarity of his working style with that described in the discussion of *personal safety* in the *Crystal Clear* (Cockburn 2005, p. 50?):

> Skillful leaders expose their team members (and themselves!) to these situations early, and then demonstrate with speed and authenticity that not only will damage not accrue, but that the leader and the team as a whole will act to support the person.
>
> One project leader[57] told me that when a new person joined her team, she would visit that person privately to discuss his work and progress, and wait for the inevitable moment when he had to admit he hadn't done or didn't know something.

[57] Thanks to Victoria Einarsson in Sweden.

This was the crucial moment to her, because until he revealed a weakness, she couldn't demonstrate to him that she would cover for him or get him assistance. She knew she was not going to get both reliable information and full cooperation from him until he understood properly that when he revealed a weakness or mistake, he would actually get assistance. She said that some people got the message after her first visit, while others needed several demonstrations before opening up.

I was stunned by Joe Wolfe's account of designing and building an airport, as it didn't match my preconceptions at all. As a final question, I asked him if this way of working was normal in the airport-designing industry. He said, no, it wasn't, but he couldn't see how anyone could work in any other way and get the terminal completed on time. He added that he had been working this way for several decades, and his clients were so happy that he never had to look for work.

## Book publication

Finally, one can apply agile development to book publication. I did this for the first edition of this book. The normal book publishing process takes about four months. Since we had only two months before the conference announcing the book, I asked to manage the publication process myself, using the principles in this book.

The first thing I did was to select people all living in Salt Lake City – the closest I could get to full collocation. In a normal production process at that time, the manuscript would be mailed in paper form from person to person, with pencil marks on it for changes to be made. These days, the manuscript is marked electronically and emailed, but that only partially improves the matter.

Having every specialty located in Salt Lake City is not yet osmotic communication, but at least we could all meet at someone's house to discuss the manuscript. Naturally, we used electronic markup and email instead of paper.

The second change was to work incrementally. Most of the editors I have encountered like to mark up the entire manuscript in one pass. This is largely a consequence of people working separately across large distances. Knowing we would meet periodically, we arranged to build each chapter separately. I made a dependency grid marking the work needing to be done by each person so that the illustrator could work at a different pace and on different chapters than the copyeditor and the page layout person. I allocated a certain number of rework cycles for each person. This way, each person could be busy doing their work as independently as possible from each other person.

I met with the copyeditor after she marked up each of the first several chapters, so we could synchronize the style and nature of the changes. As we formed agreement on what counted as a mistake versus what to leave as author's style, the number of marks she had to make got smaller, and the number of corrections I had to make to her corrections got smaller. This was an improvement over the usual mode, in which the copyeditor marks up the entire manuscript, and then the author has to unmark any parts that are style instead of mistake.

On the final day, we met at the house of the page layout person. We several times found ourselves working around an awkward column break and correcting sentence structure at the same time. With all of us sitting together, sometimes I would revise a sentence to make it fit before a column break, sometimes the layout person would reduce the interline spacing by a small amount to make the extra line fit within the vertical space.

This experience with book production taught me several things.

- Agile development practices work very well with book production. We reduced production time from four months to three weeks.
- The copy editor was never comfortable with the process we used. One reason was that she never knew what state the book was in at any point in time (she was not accustomed to the high flux). However, I came to realize that one of her pleasures in life was sitting down with a paper manuscript and a pot of tea, and having a quiet morning marking up the text with her pencil. Working online, working one chapter at a time, and running out to meetings with the author did not suit her working style. (Does this sound like any programmers you know?)
- There was an extra cost to working the way we did. When she submitted her invoice, she wrote that the invoice was higher than she originally bid because the author made last minute changes that cost her time, and also because of the extra meetings she had to attend. (Does this sound like any programmers you know?) This matches the prediction of the economic model in Chapter 4 (p. ???). That model shows that concurrent development should be faster but more expensive than sequential development. We worked about four times faster at about a 10% cost increase.
- Finally, not everyone enjoys iterative and concurrent development. At one point, when the copy editor was resisting meeting me to discuss the mark-up, I started explaining the benefits of our get together. She cut me off, saying, "I know – I'm editing that chapter!" She was happy afterwards to go back to her paper manuscripts and pot of tea and put agile book production in her past (for the time being!).

## Conference Organization and Limits of the Agile Model

I can think of only two environmental factors that limit the applicability of the agile model: distributed teams and inability to use incremental development.

Distributing the team slows communication and feedback, which makes it more difficult, though not impossible, to use the agile approach. This was discussed at length in "Agile teams must be collocated" on page 39???. When working in a distributed team situation, more care must be placed on following the agile principles in general.

Any situation in which incremental development can't be done loses some of the benefits of the agile approach. Space exploration is one.

The incremental version of putting a man on the moon would be to put his foot on the moon in the first space shot, and then to add other parts of his body and equipment on subsequent space shots. Clearly that doesn't make sense. What we employ instead is use of iterated prototypes: putting a whole man in orbit, putting a man and a craft around the moon, and then, finally, ten years later, putting a whole man on the moon.

Note that all the other agile principles can be used even in space exploration.

Possibly the most difficult situation to try to apply the agile model is in organizing a conference, particularly with volunteers (as is often the case) located in different parts of the world (as is often the case). There are no iterations, there are no increments, there is no opportunity for feedback, there are no users to get feedback from.

Here is our story from organizing the Agile Development Conference of 2003 and 2004, and how we "ate our own agile dog food."

- Before we pitched the conference to the first potential hosting group in 2001, we created a vision statement for the conference that stated why a new conference should come into existence and how each part of it fit into the overall purpose (ADC2003 URL). Clear vision statements are part of aligning teams. They are core to agile projects because they allow people to move fast, in harmony, and with less bureaucracy than otherwise.
- We recognized that while incremental and iterative development aren't meaningful to the organization of a single conference, they are meaningful to the organization of an *annual* conference. We therefore ran numerous reflection and feedback sessions, starting on the last day of the 2003 conference. We collected from both attendees and organizers what they would like to keep the same or change for the next year.
- We compared the feedback comments to the original vision, and decided whether it was our vision or just our implementation of the vision that needed correction.
- Some of the people organizing the following year's conference sat around my house with our feet up (and some beer in hand), discussing the differences between software development and conference organization. We wanted to see what might be done to copy into conference organization what we knew worked in software development (notice our use of a face-to-face setting and a relaxed atmosphere to spark invention).
- We decided to collocate most of the conference organization committee. We established the idea of a "design epicenter" for the conference. The design epicenter could be in a different place than the conference. That would allow us to choose a geographically constrained region where we could find all the needed track chairs. We selected London as the design epicenter for 2004, and Silicon Valley as the one for 2005.
- The London-based track chairs met periodically at pubs to discuss ways to better integrate the different conference tracks. I attended a few of these, and even I was surprised at the number of new, ideas that showed up at each meeting. The group made more progress in a single evening together than in a month of emails. (The ideas were surprises; that they made such progress was not – it was why they met at pubs in the first place.)
- One of the better innovations was to borrow from the apprenticeship model. We decided that the following year's track chairs would work as understudies to the London group. Then, when it was their turn to act as track chairs, they would come to the topic warmed up, understanding the thoughts that had gone into the various decisions (this is borrowing from Peter Naur's *theory building* model, see Appendix A).
- A lesson from the first ADC was that even the Open Space sessions (Owen 1997) were too formal for what we intended. People still enjoyed the conversations in the hallway the most. We therefore created the position of Hallway Chair. This person was to see that the hallway itself could maximize communication and community. The things we decided as crucial in the Hallway Session were:
  - a hallway (don't move it into a room and give it a session name),
  - coffee (make sure the hotel staff never clear away the refreshments, but make sure they are available and fresh all day),

- tables where programmers could sit and show each other their special tricks and interests,
- lots of butcher paper and whiteboards for people to draw on while standing around and talking.

The Agile Development Conference somewhat typically became the victim of a hostile takeover and was merged with a competing conference. Many innovations were lost: the idea of a design epicenter, the apprenticeship model, the use of a vision statement, the reflection workshops following the conference. As predicted in the cooperative game model, with the change of team came a loss in the understanding of what made the conference special[58].

While I feel sad about the loss of ADC's innovative ideas and vision, all the above is to be expected in business, whether agile or otherwise. It is partially satisfying to see that the theory predicts properly.

I catalog these ideas here in case someone else has occasion to use them in their own conference design.

[58] The continued presence of Todd Little kept most of the conference structure together. Todd helped design the first ADC, chaired the second, and then chaired the first two years of the merged conference. If Todd had not continued, much more would have been lost.

# 6'

# *The Crystal Methodologies: Evolution*

*Crystal is my considered effort to construct a package of conventions that raises the odds of succeeding in the project and actually being used.*

*The big shift since the first edition is the completion of Crystal Clear, the version for small, collocated teams. Since Crystal Clear is described at length in its own book (Cockburn 2005cc), what belongs here is to share the insights I gained into Crystal and agile methodologies in general while writing that book. Those include, Crystal's genetic code, installing Crystal Clear, stretching Crystal Clear to Yellow, and new techniques.*

# The Crystal Genetic Code

Crystal is a family of methodologies with a common genetic code. The genetic code helps you to design a new family member when you need one, and tell whether a given methodology is a member of the family. Jens Coldewey summarized Crystal neatly:

*The Crystal methodologies are about frequent delivery, close communication, and reflective improvement.*

Crystal is my considered effort to construct a package of conventions that work well together, and that taken together increase the odds of project success and the odds of actually being used by the team. It is possible that the package is faulty, but teams that have used it have reported good results on both fronts.

The Crystal methodologies are intended to be tailored at the start of each project (and periodically within the project). I put bounds on the tailoring, identifying the places where I feel a sharp drop-off in the likelihood of success occurs ("Sweet Spots and the Drop-Off", p. 47???). My observation is that when people do things similar to Crystal, they usually don't reflect on their work habits very seriously. Without reflection, they lose the opportunity to improve their output. (They also usually don't deliver the system very often, and they don't have very good tests, but those are separate matters.)

The Crystal genetic code consists of six elements:

- The cooperative game mindset
- The methodology design priorities
- The methodology design principles
- The seven properties of highly successful projects
- Techniques selected per personal discretion, but with "interesting" starter techniques to consider
- Sample methodologies to copy from

### *The Cooperative Game Mindset*

Treating software development as a cooperative game of invention and communication frees the team to consider what *moves* make sense at any moment in time and what *strategies* create a good position for the next move. It frees the team from the illusion that there is a single formula to use across all projects.

Treating the project as part of an interconnected series of games highlights the importance of balancing the need to deliver *this* system with the need to set up for the next and adjacent games.

This book has described the cooperative game model in depth.

### *Methodology Design Priorities*

Every methodology author has some priorities in mind while selecting what to include and what to exclude. Mine are:

- Project survival (it should deliver the system)
- Process efficiency
- Process habitability, or tolerance

The first priority is the overriding one. If the project does not deliver the system, there is a basic failure.

The next two priorities compete with each other. On the one hand, there are extremely efficient ways of working that people will simply refuse to use. It still happens today that the programmers can avoid or subvert the process if they don't like it. To be successful, a methodology needs to be accepted by the team. Therefore, habitability – that that people are willing to live with these conventions – is critical.

I used to write this priority as *tolerance*, meaning how much variation would be considered acceptable from person to person and from project to project. From my perspective, the methodology needs the tolerances on the conventions to be as loose as possible. It should work across the widest possible variations in practice. However, I was told that the word *tolerance* translates with undesirable side-effects into other languages, and so we chose *habitability.*

Habitability runs against efficiency, because the team, or individuals on the team, may decline to use some more efficient technique or convention. My feeling is that possibly-less-efficient conventions that are actually used may serve the team better over time.

Balancing efficiency against habitability is not a *Shu* level skill. It requires one or more people on the team to have a *Ri*-level understanding of software development to make this balance. This matches the acceptance level I have seen for Crystal. Beginner teams look at Crystal, don't see a clear formula to use, and move on to XP (version 1) or Scrum.

Advanced teams know that methodologies out of the box are unlikely to fit their situation. They like Crystal because it selects just enough rules to get the project into the safety zone, and leaves them plenty of room to incorporate their situation-specific adjustments.

### Methodology Design Principles

The principles used to design Crystal were presented in Chapter 4 . Crystal particularly emphasizes the value of face-to-face communications, adjusting methodology to project type, and attending to project-specific bottlenecks.

These principles are visible in the differences between the Crystal family members.

### Seven Properties of Highly Successful Projects

I only recently awoke to the realization that top consultants trade notes about the *properties* of a project rather than on the procedures followed. They inquire: Is there a mission statement and a project plan? Does the team deliver frequently? Are the sponsor and various expert users in close contact with the team?

Consequently, and in a departure from the way in which a methodology is usually described, I started asking teams to target key properties for the project. "Doing Crystal" becomes a matter of achieving the properties rather than following procedures. Three motives drive this shift from procedures to properties:

- The procedures may not produce the properties. Of the two, the properties are the more important.
- Other procedures than the ones I choose may produce the properties for your particular team.
- I hope to reach into the *feeling* of the project. Most methodology descriptions miss the critical feeling that separates a successful team from an unsuccessful one.

Naming the properties provides the team with catch phrases to measure their situation by: "We haven't done any *reflective improvement* for a while." "Can we get more *easy access to expert users*?" The property names themselves help people diagnose and discuss ways to fix their current situation[59].

A Crystal team measures its condition by the team's mood and the communication patterns as much as by the rate of delivery.

I noticed this while interviewing Géry Derbier of Solystic about his experiences with his Crystal Yellow project (see "Extending Crystal Clear to Yellow" and its sidebar). I was trying to work out why he said he was using Crystal, and I realized that he had been focusing on the quality of the communications within his team, across organizational boundaries, with both his customer-side sponsors and with the external test team. He watched the delivery cycles, reflected with the team each month on how work better together, and experimented with fitting different authors' ideas into the group's working patterns.

[59] Paul Oldfield catalogs problems a team is running into, and what counteracting practices they may want to consider: http://www.aptprocess.com/whitepapers/risk/RiskToPatternTable.htm

The following list presents seven properties of highly successful projects along with test questions for each. The first three are the unifying theme across the family. The other properties increase the safety factor for the project. The properties are described in greater detail in *Crystal Clear* (Cockburn 2005cc)

*Frequent Delivery*

Have we delivered running, tested, and usable code at least twice to our user community in the last six months?

*Reflective Improvement.*

Did we get together at least once within the last three months for a half hour, hour, or half day to compare notes, reflect, discuss our group's working habits, and discover what speeds us up, what slows us down, and what we might be able to improve?

*Close / Osmotic Communication*

For Crystal Clear projects: Does it take us 30 seconds or less to get our question to the eyes or ears of the person who might have the answer? Do we overhear something relevant from a conversation among other team members at least every few days?

For other Crystal colors, replace those specific times with an inquiry into how long it does take to get a question to the right person, and the frequency of serendipitous discovery.

*Personal Safety*

Can we tell our boss we mis-estimated by more than 50 percent, or that we just received a tempting job offer? Can we disagree with him or her about the schedule in a team meeting? Can we end long debates about each other’s designs with friendly disagreement?

*Focus*

Do we all know what our top two priority items to work on are? Are we guaranteed at least two days in a row and two uninterrupted hours each day to work on them?

*Easy Access to Expert Users*

Does it take less than three days, on the average, from when we come up with a question about system usage to when an expert user answers the question? Can we get the answer in a few hours?

*Technical Environment with Automated Tests, Configuration Management, and Frequent Integration*

Can we run the system tests to completion without having to be physically present? Do all our developers check their code into the configuration management system? Do they put in a useful note about it as they check it in? Is the system integrated at least twice a week?

There is one additional property worth mentioning, one that only shows up on the very best of projects:

*collaboration across organizational boundaries*.

Collaboration across organizational boundaries is not a given result on any project. It results from working with amicability and integrity both within and outside the team. It is hard to achieve if the team does not itself have personal safety and, to a lesser extent, frequent delivery. I consider the presence of good collaboration across organizational boundaries as evidence that others of the top seven safety properties are being achieved.

### *Techniques á Discretion*

A common mistake of the beginning methodologist is to believe that latest technique he just learned is the master technique that all people must use. As Andy Hunt and Dave Thomas, the "pragmatic programmers" have gone to great lengths to point out (Hunt 2000???), there are a steadily growing number of useful techniques in the world. It is the responsible of the professional in his field to learn how to use them.

It makes no sense for me, as the author of a methodology to be used by widely varying people on teams in different countries, working on different sorts of projects with different sorts of technologies, to legislate what technique will work best for each of those people and each of those teams. That is something for each person and team to work out.

From the perspective of the Crystal family of methodologies, any technique of invention is quite all right, as long as the person can explain the idea to the rest of the team afterwards.

Thus, some people will like to model in UML, others will not. Some will like to program in pairs, others won't. Some will thrive on test-first development, others will write code first and tests after. Some teams will use the XP planning game, some my Blitz Planning technique, and others the Scrum backlog list.

What I have done in this book and the Crystal Clear book is to list techniques and strategies that you and your team may find interesting and useful.

Here is short summary of interesting strategies and techniques for you to consider:

- *Exploratory 360°:* Pre-project safety check. In a few days or a few weeks, sample the project's business value, requirements, domain model, technology plans, project plan, team makeup, and methodology.
- *Early Victory:* Small wins helps a group develop strength and confidence. Arrange for some early in the project. *Walking Skeleton* is a great start.
- *Walking Skeleton:* A tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components.
- *Incremental Rearchitecture:* Evolve the architecture in stages, keeping the system running as you go. This applies both to the *Walking Skeleton* and to later revisions.
- *Information Radiator:* A display posted in a place where people can see it as they work or walk by. It shows readers information they care about without having to ask anyone a question. Ideally it is large, visible to the casual observer, easily kept up to date, understood at a glance, and changes periodically, so that it is worth visiting.
- *Pair Programming:* Two people sit side-by-side, working on the same code. They can be two programmers, or a programmer and a user, business specialist or tester.
- *Side-by-Side Programming:* Two people sit side-by-side, but at different workstations, working on different assignments. The workstations need to be close enough that each one can read the read the other's workstation simply by turning her head (60 cm – 90 cm, or 12" – 18"). They help each other as needed.
- *Test-Driven Development:* The test, or executable example, is written before deciding how to design the code. It serves test both as specification of what to design and as a practice run at using the call sequence to the function. Also called XXD (see page 61???).
- *Blitz Planning:* An index-card based planning session in which the sponsor, business expert, expert user, and developers, together, build the project map and timeline. Unlike XP's *Planning Game,* the cards in the *Blitz Planning* technique show tasks and task dependencies.
- *Daily Stand-up Meeting:* Everyone in the team meets, standing, for a maximum of 10 minutes to announce what they each are working on and where they are getting stuck.
- *Agile Interaction Design:* A one- or two- day sticky-note and index-card based system usage modeling session, based on the ideas in *Software for Use* (Constantine 2000). Described in (Patton 2005cc).
- *Burn Charts*: A time-based graph of number features to be completed over time and features completed so far.
- *Dynamic Priority Lists:* A list of features to work on in the next iteration or the whole project, prioritized

and put into ranking order by the sponsors, and updated by them at the end of each iteration or delivery cycle.

- *Reflection Workshop:* A work pause, for an hour periodically, certainly after each delivery, to reflect on the product, progress, and process.

***Sample Methodology Designs***

Humans are better at modifying than inventing. For this reason, Crystal contains sample methodology descriptions. Think of "cutting and pasting" the following into your methodology.

It is not the intention with these descriptions that you pick them up and use them untouched, but rather that you pick them up and critique them, adding and subtracting details until what you have suits your situation. Methodology modification is, after all, a core element of Crystal.

## Crystal Clear

At the time of the first edition, the Crystal Clear book was still in draft form, and the description of it fit into one page.

The book *Crystal Clear: A Human-Powered Methodology for Small Teams* came out late in 2004. In it, I summarized Crystal Clear with a single (long) sentence:

"The lead designer and two to seven other developers

- ◦ in a large room or adjacent rooms,
- ◦ using information radiators such as whiteboards and flip charts,
- ◦ having easy access to expert users,
- ◦ distractions kept away,
- ◦ deliver running, tested, usable code to the users every month or two (quarterly at worst),
- ◦ reflecting and adjusting their working conventions periodically."

The book itself runs to about 350 pages, which of course prompts the question, "How can you spend 350 pages describing one sentence?"

The answer is exactly the difference between *Shu*- and *Ri*-level descriptions. The one-sentence version was derived from talking to experienced developers, and is sufficient description for experienced developers. The 350 pages are needed to describe

- ◦ the background behind the parts of that (long) sentence,
- ◦ the Crystal genetic code and the seven properties of highly successful projects,
- ◦ the boundary conditions for Crystal Clear,
- ◦ ways to stretch the sentence for different circumstances,
- ◦ strategies that are likely to be employed,
- ◦ less-known techniques the team might like to try,
- ◦ samples of the work products that are likely to be generated (besides code),
- ◦ and (best of all) an experience report on using Crystal Clear, with commentary by an ISO 9001 auditor.

*Crystal Clear* is an extension of this book from theory to practice, what a small project team needs at the ground level, for teams at the mixed *Shu, Ha* and *Ri* levels. *Ri*-level leaders and people on larger teams can use the book to get ideas of both work samples and new techniques to try.

**Introducing Crystal Clear, by Jonathan House, Manager of Software Development, Amirsys**

In March of 2005, I had just taken a management position with Amirsys, a medical informatics company. I was excited about joining a new company, but feeling a sense of resignation at all of the heavy lifting that was in front of me to get the new company to adopt an Agile approach to developing software.

The sense of resignation was from my previous experiences as an "agile mentor," working to bring agile software development practices into various companies I had consulted with. In each case, the new practices were well received, but never really caught on, for a variety of reasons.

As I was preparing to leave our monthly Salt Lake Agile roundtable meeting, Zhon Johanssen, a long-time XP practitioner and one of the roundtable "old-timers" approached me. Like me, he was intrigued by the clean slate that Amirsys presented, but unlike me, he wasn't carrying my sense of resignation. He had an idea – self-organizing adoption of agile practices..

Rather than me pushing the practices onto the team, he suggested that it would be interesting to see what practices the team would adopt when given specific "stimuli".

I liked the idea because it would allow me to become a "minimalist" project manager (or "maximalist," as one person corrected me – I get the maximum results from each action). My sense of resignation was gone, just like that.

Although I had experience with XP, Scrum, DSDM and FFD, I decided to use Crystal Clear as our methodology framework, partially because the lightweight approach appealed to me, and partially because of my existing relationship with it's progenitor, Alistair Cockburn.

The very next day I reached my first crisis with my new philosophy as I started to prepare to fight with the executive team about the need to adopt an incremental, iterative approach. This was always the first fight I fought in my other mentoring engagements.

I stopped. I closed the door of my office, closed the shades, turned off the light and started thinking about how to get the executive team to decide that they had to have iterations. It was then that I remembered a lesson from my high-school biology class. Put simply, the executive team had to "feel the pain".

I walked into the director's meeting armed only with a notepad and a goal of ferreting out the pain that would lead to a realization that an iterative approach would be the necessary cure.

As soon as the meeting began I realized that I wasn't going to have to work very hard to get the executive team to feel the pain. First on the agenda was a discussion of how to get the development team to be more responsive to changing business requirements. After about 10 minutes of carefully directed questioning on my part, the President of the company said "Is it possible to get the development teams to deliver just a part of the functionality that we need more frequently?"

I grudgingly agreed that it may be possible, and that I would see what I could do with the development teams to get them to deliver functional code on a more frequent basis. As I walked out of the meeting I was shocked to realize that we had become an incremental development shop without so much as a single PowerPoint slide shown.

Now that I was mandated to implement iterations, it became clear that a lack of automated tests would be a serious drag to us.

Continuing the experiment, and looking to install the right pain in key places, I asked that we increase the formality of the testing process.

Pain immediately set in. Testing became an "all hands on deck" operation that completely halted other efforts. For a full two weeks this pain went on, and not once did I mention "automated tests" or "test first development".

At the end of the iteration, I got the team together and asked them if they thought it was a good idea to talk about the previous iteration (thereby introducing the reflection session practice). After letting the team vent about the pain of the testing process for a while I said, "You know, it would be nice if we were able to automate most of the testing effort, wouldn't it".

The development team decided that it would be a good idea to write code-based tests at the same time they worked on application features. Our director immediately allocated resources to develop automated tests and reduce the 80 or so pages of manual tests.

I still hadn't given that two-hour long PowerPoint presentation about how automated tests are a good idea.

The pattern had emerged, and the best part was that it was self-reinforcing. As our development teams reached a point where a specific Agile practice was needed, I went looking for the pain. Once I found it, I made it a point to insure that everyone felt that pain. As soon as the pain was established I would lightly introduce the team to the needed practice and then get out of their way.

one of the developers mentioned that a critical user story had been missed in the last iteration planning session. I took the developer aside after the meeting and showed him the “iceberg list” (Figure 5-12 in *Crystal Clear*). I came in the next day to discover that our backlog lists had been reorganized into iceberg lists.

The new approach worked with the executive team as well.

Executive team members were introducing conflicting priorities on the goals of the iteration (this is known as “feature thrashing”). So I began convening meetings in front of the iteration chart (Figure 5-22 in *Crystal Clear*) whenever a change was requested mid-iteration, with the development team and all of the executive team stakeholders. Not only did the executive team members resolve the conflicts themselves, but they now go directly to the iteration charts on their own initiative to discuss changes to the functional goals of the iteration.

Strangely enough, with all of this success, I still wasn't happy. As a project management minimalist I felt I was doing all of the pain discovery and “inflammation.”

Our daily stand-up meetings were taking too long, and I could tell one of the developers was getting annoyed and spacing out during the meeting. Rather than me policing the meeting as I might have done before, I just showed him the one page description of the stand-up meeting in *Crystal Clear*. The next day, he objected during the meeting that it was taking too long, and started enforcing the correct, shorter format.

During another stand-up meeting, one of the developers mentioned that a critical user story had been missed in the last iteration planning session. I took the developer aside after the meeting and showed him the “iceberg list” (Figure 5-12 in *Crystal Clear*). I came in the next day to discover that our backlog lists had been reorganized into iceberg lists.

Today, Amirsys is a great place to be a project manager. We can tell the state of every project simply by walking into the developer area and looking at the walls. Developer team members handle most of the status reporting tasks themselves. There are several copies of Alistair's *Crystal Clear* book floating around the office, and we have matured to the point where team members are starting to identify “pains” on their own and look for applicable solutions.

And I still have yet to create a single power-point presentation.

## Stretching Crystal Clear to Yellow

Many people have noticed the gap between Crystal Clear (3 - 8 people) and Crystal Orange (30 - 50 people). What does Crystal Yellow look like, for 15 – 30 people?

I am indebted to Géry Derbier, of Solystic, now a French subsidiary of Northrup Grumman, for deriving a version of Crystal Yellow and reporting back on it.

Solystic won a bid to build a new post office just outside Paris to handle all the international mail going into and out of northern France. The overall project included the building itself, plus the mechanical, opto-electronics and information processing systems, and also hiring and training the staff. Géry Derbier was the project manager for the information processing software.

It was significant that the purchasing sponsor, the French Post Office, was looking for a supplier who was willing to do incremental development and get paid every six weeks in exchange for demonstrating integrated features that often. It was also significant that the external test team was located at the site of the new post office building, not at Géry's location.

Here are the highlights of Géry's team's conventions:

- *Seating*: The software development team occupied one floor, which was shaped rather like a donut. It was less than a minute's walk from anyone's office to anyone else's. The programmers sat several people to a room in several adjacent rooms. There was a "common room" where they held stand-up, status, and other meetings.
- *Time cycles*:
  - *Iterations*: Iterations were one month, and started off with a planning session and a reflection about the previous month's work.
  - *Stand-ups*: They experimented with daily stand-ups, and moved to stand-ups on Monday, Wednesday, and Friday.
  - *Integration*: They tried but never got down to daily builds. Full builds were done three times a week.
- *Programming*: The team tried and rejected pair programming, so they programmed in near proximity to each other.
- *Deliveries and user viewings*: The key representative from the Post Office visited one day each week. Integrated code was evaluated for acceptability and payment each six weeks.
- *External test collaboration*: A representative from the external test team visited and worked with the development team one day each week.

In terms of the agile evaluation framework presented at the end of the last chapter, we see for this project the following:

| | |
|---|---|
| # People | 25 |
| Frequent Delivery | 1 year (a danger signal), but early integration with acceptance tests attached to payments every six weeks (a safety factor mitigating the danger signal). |
| User Viewings | 1 week (a safety factor mitigating the danger signal) |
| Reflection Workshops | monthly |
| Osmotic Communication | 1 floor |
| Personal Safety | yes |
| Focus (priorities, time) | both |
| Easy Access to Expert Users | 1 day/week |
| Configuration Management | yes |
| Automated Testing | no |
| Frequent Integration | 3/week |
| Collaboration across Organizational Boundaries | yes |
| Iteration length | 1 month |

In terms of technique and strategy use, we see the following:

| | |
|---|---|
| Exploratory 360° | Yes |
| Early Victory | Yes |
| Walking Skeleton | Yes |
| Incremental Rearchitecture | Yes |
| Information Radiators | Yes |
| Pair Programming | No |
| Side-by-Side Programming | No |
| Test-First Development | No |
| Blitz Planning | Yes |
| Daily Stand-up Meeting | Yes |
| Agile Interaction Design | No |
| Burn Charts | Yes |
| Dynamic Priority Lists | No |

I am happy to report that the project delivered on time and the development team was recognized for their success.

I asked Géry what contributed particularly to the end success, particularly given a schedule that looked overly optimistic when I visited the project. He answered that monthly incremental development was key. Developers who might otherwise have spent weeks worrying over what was the "right" or "best" way to do something, recognized that with only one month to deliver function, getting an acceptable design out the door was better than a perfect design that didn't get delivered.

What I noticed was how much attention Géry gave to the quality of the community, morale and communication, and how much attention he gave to the quality of the team's linkage to the sponsors in the Post Office and to the external test department. He was able to build close collaboration across organizational boundaries.

Finally, we can't overlook the effect of having a sponsor who paid for integrated, running features every six weeks. Whatever Géry's bosses may have been think higher up the management chain in Solystic and Northrup Grumman, they saw a team presenting features to a customer every six weeks, and a corresponding check coming from the customer shortly thereafter. That simplifies discussion with management a whole lot.

Here is Géry's story.

## *Tailoring Crystal Yellow*, by Géry Derbier, Project Manager, Solystic

The project had several challenging aspects :

- It was business critical for the customer;
- It was the first time the customer was contracting with a integrator to provide a turn-key system;
- It was the first time Solystic was leading a complete system project, although its mother company Northrop Grumman had previous experience of this business;
- Though Solystic has a long experience in the automation of domestic mail processing, this program was one of the first business in the international field;
- It was a fixed-price, fixed-time contract to be delivered in a short time frame.

I started with pretty few ideas and guiding principles. The cooperative game model was the framework of thoughts I would always come back to in order to make sense of what I was seeing happening. The "team = product" formula, which I consider a stronger form of the Conway's law (see *Software for your head* (McCarthy 2002) was how I understood how to influence the final product quality through the group shape and characteristics. Incremental growth of both product and process requiring periodic reflection and adaptation was at the core of the method. We employed use cases to specify the system as among the qualities of uses cases was they act as a tool to link several teams together, especially development and integration teams.

During the RFP process I elaborated a development plan built on the principle of fixed-length iterations and frequent interactions with the customer team. I have later been told the customer considered this feature to be a competitive advantage. Timeboxed iterations came to me through Jim Highsmith's (2000) book *Adaptive Software Development* book and the reading of DSDM (Stapleton 1999???). At that time, I borrowed the iteration structure from Stappleton's book. During the project, we brought small variations about the form of communications with the customer during an iteration.

The overall project had a phase-and-gate structure. Six months after contract award, we had to have an approved system specification, a so-called baseline. The thirteen month milestone was the beginning of the site full system integration. Then eighteen months into the project the sorting center would open to real production with a six months period of staged ramp-up concluding with the final acceptance of the system.

We had to demonstrate running features as early as nine months after the contract was awarded. These contractual demonstrations were interim acceptance tests linked to payment by the customer. All these milestones were defined in the contract with a very high level functional description of the content.

Organizational patterns (Coplien 2005) was the body of knowledge we used to design the group structure and communications. As an example, we started with a limited the number of roles ("few roles" pattern) to include :

- project leader
- sub-team leader
- business expert
- lead designer-programmer
- other designer-programmer
- tester

We estimated the team would amount to a peak 25 persons. However we quickly assembled a team with the necessary core competencies: the project leader, the test lead, the database lead with a database designer, the application development lead and a couple of designer-programmers. The application development lead was assigned the architecture responsibility (the "Architect Also Implements" pattern). Then we slowly incorporated additional programmers.

One sub-team leader was the overall project leader. Sub-teams were application domain teams (e.g. real-time operation control, production management tools) or technical teams (we had one DBA and one tester team for the whole group). The leader and business expert roles were cumulated by one person. In the early part of the project, the team leaders were writing the uses cases. During the last third part of the project the leaders' time was mainly devoted to integration and test.

We got several adjacent 2 to 4 persons offices on the same floor of the building. The leaders were grouped in the same office. Developers would be grouped per office as much as possible by application domain or technical activity.

It was a core practice to hold regular reflection sessions. It happened every month once the team was mainly staffed, then every two months by the end of the project. These sessions were essentially informal discussions on what was happening in the project. I played the role of a facilitator. Along the discussions, when I would recognize the need for a common decision to be taken on a given subject, I would formulate a proposal and try to come to a consensus (in the smallest group I used unanimous vote). Among the subjects I led the team to explore: conventions between the developers and the internal test team, values (courage to tell things), fear, configuration management, coding standards, integration frequency, relationship with the external test team.

So, during the course of the project, we changed several working conventions and stuck to others. We tried a number of strategies and techniques, some attempts worked, another good number failed miserably. Here are some pieces of the story.

It was my plan to split the leaders group seating in the same room and seat each of them in their sub-team room once the system specification would be sufficiently completed. However, we recognized the advantage of leaving things as they were.

As the project was about automation and evolution of the customer's process, we knew we had to face progressive discovery of the customers needs until relatively far into the project. So it was clear that we would not succeed without a fair amount of concurrency of the different development activities (the “Gold Rush” strategy (Cockburn 1998)). We naturally resorted to the “Build Prototypes” pattern.

We soon recognized the need for a specific build-manager role and a rolling assignment strategy for this role.

Despite several discussions and commitments during the reflection sessions, we failed to have a disciplined protocol for internal software delivery to the test team.

The amount of unit tests was not satisfactory until we decided to have one team member responsible for monitoring the unit test coverage and having him get the other developers to invest time in writing their tests (we were far from doing TDD!).

I encouraged people to pair-program. Because they worked in near proximity, people would occasionally work by pair to analyze or debug a piece of code, but the true practice of pair-programming never get adopted. I did not insist on having it as I considered we were asking many new things from this team, iterative and incremental development not being the least change for most people in the group.

I borrowed from the Scrum methodology the way we implemented the “stand-up meeting” pattern. Given the number of people, we considered splitting the stand-up meeting into “scrum of scrums” several times during the project . However, as we did not have sufficiently automated integration tests, we decided to keep the stand-up meeting with the full team to hold the team together. We adopted small strategies to cope with the team size: The meeting would occur every two days instead of every day; during the meeting we would focus on functional aspects. Technical aspects would be raised if necessary or only quickly noticed. The meeting duration was targeted for thirty minutes.

I insisted on having two dedicated spaces for the project: a meeting room and the testing lab. The meeting room was a place with many white boards where the team could hold collective design sessions. On the white boards was an information radiator dedicated to the iteration planning and tracking: the team progress was tracked during the stand-up meeting by moving around the sticky notes on this task-board. The room was big enough to permit the stand-up meeting with the full team.

Despite all my efforts to build a collaborative culture within the team, the whole members did not work in perfect amicability. A good deal of storming occurred and unresolved tensions remained all along the project. However, a small part of the team was truly willing to ignore positional games and invest all their energy to get a successful software out of the door.

All along this project I have been a little nervous about the lightness of the process, fearing of missing something that should have been present. I retrospectively think that this lightness is precisely what helped me to be always efficiently on-guard instead of being stiffly in-guard.

# APPENDIX A'

## *The Agile Software Development Manifesto and the Declaration of Inter-Dependence*

*As the Manifesto for Agile Software Development grew in significance, people wanted to know how to apply the ideas to projects outside software development, and also to distill out the general management principles involved. A dozen and a half product, project, and management specialists from inside and outside the software industry met over a six-month period to write the more general agile leadership version of the agile manifesto. Completed in January, 2005, they called this the* Declaration of Inter-dependence*, or* DOI, *and started the Agile Project Leadership Network.*

*In the following sections, I review the writing of the agile manifesto and discuss in more detail the DOI.*

## The Agile Manifesto Revisited

The writing of the Manifesto for Agile Software Development was a watershed moment. We didn't know that at the time of writing, of course, although there were a number of prolific authors in the group. The rapidity with which people around the world undersigned the manifesto on the web site surprised me. Even more surprising was that within six months conferences already had panels on the subject. In 2006, I saw "requiring an agile approach" written into a co-source agreement between two very large companies, and the Software Engineering Institute was developing ways to help its examiners evaluate agile projects.

Many people ask what it was like to write the manifesto, and in particular, what our meeting process was. I'll offer my own personal observations of that meeting, why it was unusual, and perhaps why it worked. There isn't need or space to go into great detail, so I shall just pick out what has seemed in the light of subsequent experiences to be interesting observations.

Bob Martin called the meeting, saying, "Is it just coincidence that so many of us are saying things that sound alike?" He added that he was interested in writing a manifesto. (I, for one, was completely uninterested in writing a manifesto, so the reaction to the manifesto was perhaps more surprising to me than to him). Bob invited a wonderful set of people to the meeting, both proponents of the "lightweight process" view as well as some others. Noticing that we had people only from North America, we asked for a representative of the UK-based DSDM consortium to be present. Ari van Bennekum flew in from Holland just for the meeting. We ended up with representatives for XP, Scrum, Crystal, Adaptive, Feature Driven Development, DSDM, and generally light and pragmatic development (Andy Hunt, Dave Thomas, and Brian Marick).

After the round of "Hello Who Am I", we sat around and stared at each other for a minute, and someone said, "How do we make an agenda?" Someone suggested writing agenda items on index cards, and since the XP people in the room of course had index cards in their possession, they immediately pulled some out, started writing on them, and threw them into the center of the floor. Suddenly there was a critical mass of people casting index cards onto the floor. The rest, whatever they thought, started doing the same, until we ran out of ideas and had a pile of index cards on the floor.

Someone asked, "How do we organize those cards into an agenda?" Someone said (you will notice that by this time, I had already lost the ability to note who was suggesting what), "All those who care about the order of the agenda sort them out, and the rest can leave us to it." Since I didn't care about the sequence of topics, I took a break. When I came back, the index cards were being taped onto the wall.

Later on, someone asked, "I just thought of another agenda item. What should I do with it." Someone answered, "Find the place in the agenda you'd like it, and put it there."

I spend time on all this because two things strike me as very significant about this group so far:

- Respect. There was enormous trust in the room of each person for each other. No one tried to hijack the meeting; everyone listened very closely to every other person, giving at all times greatest credence to what that person was saying.
- Self-organization. This was self-organization of the highest caliber. Under other circumstances I have been in, one, or worse, multiple people have tried to "run" the meeting. With senior people in the room, many of whom don't know the others, it is easy a power struggle to show up in the first minutes. That never happened.

We spent time on a round of "What am I recommending and what do I stand for," by way of introduction. I noticed that each of our processes, when described in only 15 minutes, seemed strange to some of the others in the room.

We quickly decided that it was not an accident that our recommendations sounded so similar, there was something underlying that similarity, and we should discover what it was. The reason this is important to mention is that we disagreed back then on how to run projects on the day-to-day and minute-by-minute basis, and we disagree still. Notwithstanding those disagreements, there is a very strong commonality that separated then – and still separates – our views from many others.

Someone said, "I don't like the word "lightweight", it doesn't sound like we're serious." Someone else said, "*Lightweight* is a reaction *against* something. I want a word that stands *for* something." So we spent an hour searching for a tag word to capture what we were after.

I found the process of doing that word search interesting:

- First, we brainstormed 20 or 30 names and wrote them down.
- Second, we picked out a few names and discussed for each why we *didn't* like it.

The latter tactic helped us understand what we stood *for* and what we stood *against*. Writing down why we didn't like a word didn't disqualify the word, it merely helped us understand what was inside our heads.

For example, for one word, an objection was, "I'd have to be wearing pink tights to say that word. I refuse to wear pink tights." We were looking for a strong word, to match Musashi's, "Whatever guard you adopt, do not think of it as being on guard; think of it as part of the act of killing" (or for us, delivering a useful system).

Agile development is aggressively delivering what is needed, not merely avoiding paperwork. Hence my selection of the Bengal tiger for the cover of this book, as opposed to a gymnast or a dancer. My other agile mascot is this wicked-looking, deep-sea monster I found on a t-shirt:

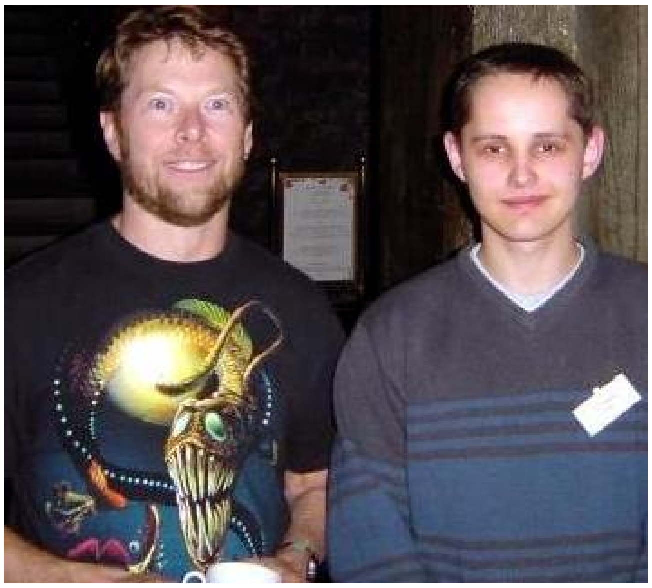


**Figure A'-1**. This t-shirt shows my favorite agile mascot (Photo with and courtesy of Jonas Kongslund).

Both the tiger and the deep-sea monster look certain to eat dinner. They are not merely avoiding paperwork or practicing flexibility.

- Third, having clarified our thoughts, we voted for our top three choice words, and ranked the results.
- The top two words were *agile* and *adaptive*. We had a round of discussion and a final vote, settling on the word *agile*. As I mentioned earlier (p. 66???), although *adaptive* has certain advantages, *agile* is the word we were looking for at the time. These days, we expect our methodologies to be both agile and adaptive.

Then someone said, "Well, now we have a word. What stands behind it?" What followed was interesting. Lunch was served, and various people sat in front of the whiteboard discussing ideas. The space in front of the whiteboard was too small for all 17 people, so people moved in and out, intermixed with eating lunch. Somewhere along the line, someone suggested contrasting *this* against *that*, and the value comparison showed up. Someone else suggested that it had to be a more or less fair value comparison, so that people could disagree meaningfully one value at a time.

When it took shape, we sat in front the whiteboard and wordsmithed it. Anyone could say why they couldn't support a particular wording, and we worked to understand their view until someone came up with a better wording. When we were done, each of us could sign for the full manifesto without reservation.

Having the name *agile* and the four value statements, we then moved to create the 12 principles. Here is where things started to break down, because it is at this level that the individual differences start to show up.

For example, does the customer or user have to be there *daily*, or is weekly good enough, or perhaps some mix of hours per week plus being available for questions? Should we write that the system gets deployed daily, weekly, monthly, quarterly? Is self-organization the crux of the approach, or is management allowed to play?

Compounding these complications, it was getting time for people to get on planes (we only budgeted two days for this whole thing, after all). Once people started to leave, we recognized that we would not be able to get closure on finer-grained recommendations. This is, indeed, what happened.

Someone pointed out that we still wanted to be able to disagree at the detail level, since we were all still exploring development methods. To use the term *precision* (p.???), we agreed with each other on precision level 0 – the single catch-phrase *agile*. We agreed on precision level 1 – the four values. We (barely) agreed on precision level 2 – the 12 principles. We agreed to allow ourselves to disagree at the next level of precision.

Finally, we reviewed why we had been able to cover so much territory so quickly. The answer was right in front of us: We had paid attention to individuals and interactions, we had worked face-to-face, we had worked collaboratively, we had focused on our product. In other words, we had just lived what we had just written.

## The Declaration of Inter-Dependence

As the manifesto grew in significance, people asked:

- What is the corresponding version for non-software product development?
- What is the corresponding set of principles and values for *management*?

Jim Highsmith notes that to understand the agile manifesto for products at large instead of just software, simply replace the word *product* for the word *software* in the manifesto, and the manifesto is still clear and correct:

Working *products* over comprehensive documentation;

Indeed, this evidenced by the myriad applications of the agile values and principles outside of software already discussed in this book. Reread, in particular, Mike Collins' sidebar (p. 40???) to see how lean manufacturing already anticipated what we were wanting to say.

On the management side, a number of people voiced an interest in exploring the extension of the agile manifesto to project management and product development outside software. We held the first meeting to explore that topic at the Agile Development Conference in 2004. That group met twice more, finally on Feb 1, 2005 writing the six points of the Declaration of Inter-Dependence, or DOI:

**We increase return on investment** by making *continuous flow of value* our focus.

**We deliver reliable results** by *engaging customers* in frequent interactions and shared ownership.

**We manage uncertainty** through *iterations, anticipation and adaptation.*

**We unleash creativity and innovation** by recognizing that *individuals are the ultimate source of value*, and creating an *environment where they can make a difference.*

**We boost performance** through *group accountability* for results and *shared responsibility for team effectiveness.*

**We improve effectiveness and reliability** through *situationally specific strategies*, processes and practices."

The signatories to the DOI were David Anderson, Sanjiv Augustine, Christopher Avery, Alistair Cockburn, Mike Cohn, Doug DeCarlo, Donna Fitzgerald, Jim Highsmith, Ole Jepsen, Lowell Lindstrom, Todd Little, Kent McDonald, Pollyanna Pixton, Preston Smith and Robert Wysocki.

That list of people is notable for the breadth of expertise, ranging from project management to product development to agile software development, to line management and to teamwork specialist.

Here below, I elaborate on my thinking about this declaration.

## Understanding the DOI

The six sentences are formed as two clauses, "We accomplish *this* by doing *that"*. In other words, you see us doing *that*, and the reason we care about *that* is because we're trying to set up *this*.

Personally, I like to read the DOI in a 2-column format, where the left column identifies the *what we're hoping to accomplish* and the right column identifies the *how we intend to accomplish that.* Consider it in this form:

| **Goal:** | **Through doing:** |
|---|---|
| *Increased ROI* | Focus on "flow of value" (not "tracking effort"), preferably in continuous (one-piece) flow. |
| *Reliable results* | Engage customers in frequent interactions and shared ownership |
| *Manage uncertainty* | Iterations, anticipation and adaptation (Think ahead, plan, iterate, deliver, reflect, adapt). |
| *Unleash creativity and innovation* | Recognize individual human beings as the ultimate soure of value; Create an environment where individual people can make a difference. |
| *Boost performance* | Group accountability for results (the whole group is singly accountable, no in-team blame); Shared responsibility for team effectiveness. |
| *Improve effectiveness and reliability* | Situationally specific strategies, processes and practices. (No one answer, folks, get used to it) |

The alert reader will notice that there is precious little about the Declaration that is unique to project management. The points apply across all management, even self-organizing and self-managing teams.

If there is one problem with the DOI, it is that the six points sound like platitudes. Perhaps that is because, unlike the Agile Manifesto, the DOI doesn't name the bad guys. It states cause and effect relations. Only when trying to apply the points do the surprises show up.

Here are some of the surprises lying in store:

- ". . . *continuous flow* of value . . ."

Lean manufacturing teaches us that a process improves as the batch sizes shrink (see "Lessons from Lean Manufacturing" on p. 37???). While developing software, the unit of inventory is the unvalidated decision. Every written requirement, architectural decision or document, design drawing or decision, and piece of written code, counts as inventory until it is integrated and tested!

Shrinking each of these to a unit piece, or continuous flow is a stretch goal to consider for software development. It implies that every decision, whether requirements, architecture, design or code, must be implemented, integrated and tested as a single-unit. I'm not good enough to do that yet, but I do understand the importance of the goal.

- "... continuous flow of *value* . . ."

Most managers manage tasks, checking task lists and task completion, when they should be managing the growth of *value*.

In software development as well as in other fields, value accrues (or fails to accrue) suddenly, at the moment of integration. All the work leading up to that moment carries no value to the buyer. This is the reason agile teams have replaced the standard earned-value chart with burn-up charts showing accumulation of running, tested features (RTF), not just completion of tasks.

- "... by engaging customers in frequent interactions and *shared ownership*. "

Shared ownership is difficult to imagine for certain consumer products, but easy to imagine for smaller customer groups. I am thinking in particular here of development for any internal user group, and custom development of any kind. See the sidebars describing Tomax's new contracts with its customers (p. ???) and Géry Derbier's relationship with the French Post Office (p. ???).

Despite that, very few organizations take the idea of sharing ownership with their customers seriously, and equally dangerous, very few custoemrs take the idea of shared ownership (and responsibility) with their contractors seriously. Many of customers don't even want shared ownership – many will say that they hired you to do "all that stuff; just deliver a useful system soon."

Surprisingly, the other surprise in the sentence is "frequent interactions." I write 'surprisingly' because the importance of frequent interactions has been highlighted, researched and reported for decades. It shouldn't be a surprise at all. However, how often does your customer, buyer, sponsor, or user base show up on the programming floor and check out what the team is really doing? The Agile Manifesto says the desired answer is "daily." The DOI says "frequent." I'll bet that in most of your organizations it is neither.

- "We *expect uncertainty* . . . "

Many managers and management advisors advocate that the point of project management is to *eliminate* uncertainty? The DOI takes the stand that uncertainty is a fact of life, that must be met on its own terms, and in this, distinguishes itself from other management advisories.

- ". . . iterations, *anticipation* and adaptation."

The DOI diverges from the Agile Manifesto by calling for anticipation. The feeling among us – including both of the Agile Manifesto authors in the group – was that the agile community has been forgetting to use the information they already have at hand, and have been costing their organizations by forcing them to react to events that should have been foreseen.

To test this point of the DOI, consider in what ways and to what extent your management balances the use of iterations with anticipation, and to what extent they ever adapt.

- ". . . by recognizing that individuals are the ultimate source of value, and creating an environment where they can make a difference. "

Be truthful, now, "People are the ultimate source of value" – how many managers really take that seriously. We do. But the DOI extends the Agile Manifesto by reminding us that the environment also matters. People treated as cogs in a machine do not do as well as people who feel not only that they matter, but that they can make a difference in other people's lives.

This is in danger of becoming a platitude, and I suspect that organizations that aren't already paying attention to this point won't change just on the basis of the DOI. However, I keep reading business cases where a company disrupts its competitors, and inside the case description a lot of space is given to way in which people are not just empowered, but feel they have a chance to change the world.

- ". . .*group accountability for results* . . .."

Group accountability for results has long been known its effectiveness. One way to achieve group accountability is through cross-functional teams. These teams have been documented and recommended for several decades because they outperform the other teams (Hammer Reengr the organization ref, Toyota way ref, Holistic Diversity ref Cockburn 1998 soop).

Lean and agile manufacturing texts recommend cross-training people at adjacent work stations. The DOI goes further and suggests that people on the team develop an awareness of what the others on the team need, so that if they can't do the job, they can help in whatever way they can.

Sociologist Karl Weick, studying teams that work in life-critical environments such as aircraft carriers and firefighting, calls this *mindfulness*: people being aware of what other people are doing (Weick 2003???). *Mindfulness* is a perfect term for one of our objectives with osmotic communication, and a goal to keep when communication is no longer osmotic.

Mysteriously, this idea gets overlooked in most companies.

- ". . .*group accountability for results* and shared responsibility for team effectiveness."

Chris Avery, the teamwork specialist among the DOI authors, clarified the shared accountability phrase. He wrote,

> "Most people assume that someone else is responsible for the effectiveness -- or lack thereof -- in their team, and that someone else should do something so that the team is more effective. The DOI says that if the team isn't as effective as "I" like, then it's up to me to take responsibility for correcting the situation."

Zhon, Johanssen, a senior programmer, describes that he views improved business impact as the *programmer's* responsibility, as is improved usability and system quality. He applies that sentiment for each job in the company, and practices it himself.

- ". . .*situationally specific strategies* . . ."

Of all the points in the DOI, this one should be most expected to anyone who has read this far in the book. We have discussed at length that what works in one situation may backfire in another.

This is not obvious to those many managers who keep looking for a closed-form formula to apply. Surprisingly, it is also not obvious to otherwise enlightened people researching "best practices." The optimal strategy for a situation may require a reversal of what would normally be considered a best practice (see "Balancing Strategies ", p. 23).

The successful team stays alert to changing circumstances, and adjusts strategies and practices to suit.

Those are some of the surprises in the DOI. There is much more one can write about it, just as there is much more one can write about the agile manifesto. A longer discussion of my interpretation of the DOI, including the DOI as a "12-step process" is on my web site[60].

Finally, it is useful to know that in constructing the DOI, each co-author embedded in it their own personal framework for management. We collected those personal frameworks and checked that each person could operate their personal framework from the DOI, even if that framework wasn't explicitly named in it.

Just so you can see one, here is my personal framework. It consists of three elements:

**People** – Not only their hearts and minds, but also their eyes, ears and mouths. (Communication: the cooperative game, remember?)

**Situationally specific strategies** - All recommendations are specific to the context at hand.

**Feedback** - Close in and end-to-end, within each work group and across the total system, within the team's process and on the end product.

You may find it useful to write down your own personal framework for project management, and see how it fits or contrasts to what is in the DOI.

## Using the DOI

I wish that the principles in the DOI may become the *default* mode of management in the future, not the exception.

### *Improved Project Management*

A production manager for one of my books tried to schedule the entire four-month book production process up front. She did this because she was worried meeting a deadline.

To make sure she left nothing out, she assigned everyone's time in half-day increments . . . and then ignored small realities, such as that I would be on a small boat in the Caribbean for ten days. Naturally those were exactly the day she scheduled for me to revise the manuscript.

She phoned each person's manager each day to check that the activity was proceeding according to her schedule.

She had the 300+ page manuscript mailed from person to person in its entirety (a large batch size, see p. ??? and p. ???), so no parallel work was possible.

She allowed no time for second revisions by any person, since of course no one would make any mistakes on a project this important.

Needless to say, the project immediately ran off the rails, starting with my ten-day vacation. People made mistakes, had other work to do, and so on.. The project was behind in no time, and everybody was miserable.

She was not malicious and not even particularly incompetent. She was trying to maximize business value, cut costs, maintain quality, and so on. Her mistake was only in understanding.

I want every person, untrained or trained, to think in terms of the DOI. I want its precepts taught in kindergarten, and used by reflex by everyone on a project. "Traditional," impersonal, batch-oriented project management, is what ought to be questioned in every case. That's what I think this declaration should lead to.

When you next put 'agile' into your contract, write *using the project management Declaration of Inter-Dependence*. That way you'll not only get working soft-

[60] http:// alistair.cockburn.us/ htdocs/ crystal/ articles/ doi/ declarationofinterdependence.htm???UPDATE THIS!!!

ware, but also the benefits of the things in the right hand side of the above table.

### *Self-Managed Teams*

A final comment is appropriate for self-managed teams. We were quite aware that there are many self-directed teams with no distinguished manager position, and wanted the DOI to be applicable for them, also.

A self-managed team should have no trouble signing up for group accountability for results and shared responsibility for team effectiveness. Also flow of value (not effort), shared ownership, iterations, anticipation, adaptation, recognition of individual human beings as the ultimate source of value, and an environment where individual people can make a difference.

Actually, it seems to me that the DOI is exactly the sort of declaration a self-managed team would hang on their wall.

# APPENDIX B'

## *Naur, Ehn, Musashi: Evolution*

I have referred at length in this book to the ideas of the three people quoted in Appendix B: Peter Naur, Pelle Ehn, and Musashi.

After five years of living with their words, I can mostly suggest that you read and re-read those extracts. I continue to find value in them.

## Naur

From Peter Naur's writing, we get the idea that the team is working to create a common theory for their work. In terms of the Swamp Game (p. 10???), the team starts off not knowing what they are supposed to build, where in the swamp to build it, or what the layout of the swamp is. The theory they are building is the answer to those three questions.

Part of the communication aspect of the cooperative game is establishing a shared direction for the team and a shared view of what the results need to look like. This is called *common vision* in some writings. Naur's theory includes that, and also a common understanding of why the thing is put together the way it is.

Common vision, and common understanding of why the thing is put together the way it is, are both part of any cooperative game, and most certainly for our cooperative games of invention and communication.

Naur's discussion of theory building as a personal activity helps us to understand modes of transmitting understanding from one person to another. There is nothing that says that written documentation is the best way to convey understanding; possibly it is the worst. If we take the challenge to "convey understanding," then we can experiment with different ways until we find some that work better.

## Ehn

From Pelle Ehn's writing, we get the idea that the understanding of the task to be done may never be perfect, but may never need to be perfectl. The magic lies in the back-and-forth between developer and user, creating new understanding about the task at hand and the tools being created.

It is easy to look at Ehn's team's assignment from 1986 and think we are long past the days when people couldn't understand how the computer could help them. However, every organization working on improving their organizational process is faced with this problem. Until the system gets delivered and put into use, there is really no way that the users can tell how the presence of the new system will change the ways they work with each, and the ways they carry out their jobs.

## Musashi

I use the Musashi quites to start off my one-day workshops introducing agile development. At first, it seems strange to use samurai quotes to understand software development, but then later it becomes so obvious how much his writings have in common with modern software development.

I highlight three of his motifs:

- Waste no movement.
- Learn each tool's strength; don't become attached to any one tool.
- Reflect and adapt.

The one thing in Musashi's writing that is different and one must grapple with, is that Musashi keeps referring to killing the opponent. Who or what is the opponent?

I was shocked in one organization when someone answered, "The users!" Looking for other opinions, I asked another person, who said, "The other specialists, like the database administrator!" Another person tried to clarify, "Sometimes you have to take out one of the other people on your team in order to get your work done."

Just in case any reader is tempted to make a similar response, I wish stress that your teammate is not the opponent, nor is the user, your manager or the sponsor.

The "opponent" is the problem you are trying to solve, the obstacles to delivering the system. "Killing the opponent" is solving the problem, delivering the system. Your situation will throw enough obstacles in your way that you don't need to consider your team mates opponents.

Remember: "There's only us."

# 7 *(new chapter)*

# *Afterword*

*Your center of interest determines what you should do with the ideas in this book.*

**REFERENCES** **121**

## Agile Software Development

Designing the user experience (studying the users, designing the system metaphor, detailing the screens) has a lot of cross-dependencies with programming the functions. Since there is no simple answer, use the principles in this book and watch closely the dependencies between user experience (UX) design and program design.

- You will probably investigate the users' needs and develop the initial system metaphor as a separate activity prior to system software development, because there is probably a long lead time needed to make those decisions.
- Once the initial UX model is built, you can probably grow the screens incrementally just ahead of the software that supports it.
- The UX designers probably need at least a week or two lead time (ahead of the programmers) to get their thoughts in order for any given function. The lead time question has to be revisited frequently, since the answer is likely vary by team and by assignment. Don't get trapped into thinking that there is only one answer. Here, more than in other places, the optimal strategy varies widely.

Avoid the common misunderstandings that have attached to agile development.

- Bear in mind that *agile* means "pay attention to current realities" (as opposed to "no planning"). Always have a coarse-grained long-term plan and a fine-grained short-term plan. Learn a fast planning technique[61], and rebuild the plan frequently.
- Don't believe that "more agile" equals "better." Agility, however nifty, is only one priority that can attach to a project. Balance the desire for agility with the other priorities clamoring for attention. Learn to mix agile strategies into your other priorities.
- Don't believe that "shorter iterations" equals either "better" or "more agile," otherwise your iterations will degenerate into nothing more than planning windows that are not connected with either business value or delivering a valuable system, Find an iteration length that works for your team *and* connects with business use.
- Incorporate automated unit testing, automated system testing, and automated system builds with tests into your team's daily activities. Frequent integration with automated testing lets the team experiment, recover from mistakes, and follow evolving needs more easily.
- Demand frequent delivery of running, tested features[62]. Negotiate over how frequent "frequent" is, and how delivered "delivered" is, but be clear that demerits accrue for each layer of the software not integrated, and each stage removed from the real end users that your "delivery" consists of. Connect the frequent deliveries to business value.

[61] See, for example, the Blitz Planning technique described in (Cockburn 2005)

[62] Thanks again to Ron Jeffries for this outstanding phrase.

## Business as a Cooperative Game

Most of business is nothing more than a cooperative (and competitive!) game of invention and communication. This means that the chapters on the impossibility of communication, quirks and motivators of individuals, rules of teams, all apply directly. Consider doing the following:

- Reread the Declaration of Inter-Dependence slowly and one phrase at a time. Ask yourself what stops you from incorporating each clause (some few may prove inapplicable). Ask yourself what your core operating principles are; keep the number to three, four, or five. Look to see whether those core principles are captured there, and if not, what would you need to add.
- Improve the quality of your team by paying close attention to goodwill in the communications and the quality of the community. The speed of the team is limited by the speed at which ideas move (plus, not to forget, the talent present). *Anything* that slows the movement of ideas slows the progress of the team.
- Hold a reflection workshop once a month, no matter what business you are in. Discuss your team's operating conventions, what is slowing the team down, and what it might try to differently. Track ongoing problems, because those may need to be escalated to higher management for resolution.
- Revisit your vendor contracts. Incorporate the idea, "They are also *us*." As long as you treat your supply chain as *them*, it slows the rate of critical feedback and the movement of ideas, and slows your business.
- Try some friendly competition inside your team, making sure to hand out interesting rewards at the end. You will probably do better if you don't make the competition directly affect the team's output, because people will try to cheat the system. Ideally, when they collude and cheat, they should improve, not hurt, your total output.
- Revisit the 2D Crystal grid of projects (the one showing team size and criticality) and the sweet spots of agile development. Wherever you can reduce team size, collocate the team, or get faster feedback, you have a chance to reduce communication and coordination overhead.
- Find opportunities to get "small wins." Even just a few to start with will make the team stronger and more robust, improve morale and community, and give you chances to tune both the direction you are headed and the operating conventions the team is using to get there.

Small wins plus periodic reflection are as close to being a magic sauce as I know. Make them part of your core.

## Leadership

Everything just mentioned regarding business as a cooperative game applies directly to leadership and project management, in software development or another business, in a management-oriented organization or a self-organized group. Consider:

- Notice that the role of the leader or manager is amply described in the following list.
  - Get executive nourishment (both money and decisions).
  - Develop talent and skills.
  - Integrate results frequently.
  - Reflectively incorporate feedback on both the product and the process.
  - Build amicability and personal safety.
  - Monitor communication for frequency and quality.
  - Arrange for adequate time to focus on the work.
  - Generate clarity in both direction and priorities.

  After you strip all the paperwork and drudge out of the leader or manager's assignment, those items remain. They become the heart of the leader or manager's work.
- Learn the language of lean manufacturing, in particular the notion of a pull system and the penalties of queues of work in progress. Learn to think of development as an assembly line of *decisions*, and visualize the dependencies between the individual people to create meaningful strategies.
- Separate project management strategies from your organization's process policies to the greatest extent possible. Different situations call for different management strategies, and you don't want to be

constrained to an obviously ineffective strategy just because someone unwittingly bundled development strategies into the corporate methodology standard.

## Everyone

Always remember: "*There's only us.*"

# REFERENCES

## *References*

(Allen 1998) Allen, T.J., *Managing the Flow of Technology*, MIT Press, Boston, MA, 1984.

Austin, R., Devin, L., *Artful Making: What Managers Need to Know about How Artists Work*, Financial Times Prentice Hall, Upper Saddle River, NJ, 2003.

(Augustine 2005) Augustine, S., *Managing Agile Projects*, Prentice-Hall PTR, Upper Saddle River, NJ, 2005.

(Boehm 2004) Boehm, B., Turner., R., *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, Boston, MA, 2004.

(Booch 1996) Booch, G., *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, Menlo Park, CA, 1996.-→ email him?

(Cerasoli 2001) Cerasoli, R., Office of the Inspector General, Commonwealth of Massachusetts, "A History of Central Artery/Tunnel Project Finances 1994 – 2001: Report to the Treasurer of the Commonwealth", available online at http://www.state.ma.us/ig/publ/cat01rpt.pdf.

(Chase URL) [9] Chase, T., "Revelation 13: The Big Dig," online at http://www.revelation13.net/bigdig.html.

(Cockburn ) Cockburn, A., *Surviving Object-Oriented Projects*, Addison-Wesley, Reading, MA, 1995.

(Cockburn ) Cockburn, A., *Crystal Clear: A Human-Powered Methodology for Small Teams*, Addison-Wesley, Boston, MA, 2005.

Highsmith, J., *Agile Project Management: Creating Innovative Products*, Addison-Wesley, Boston, MA, 2004.

Holtzblatt, K., Wendell, J., Wood, S., *Rapid Contextual Design: A How-To Guide to Key Techniques for User-Centered Design*, Elsevier, San Francisco, CA, 2005.

Laufer, A., *Simultaneous Management: Managing Project in a Dynamic Environment*, Amacom, New York, NY, 1997.

Marcus, A., *Big Winners and Big Losers: The 4 Secrets of Long-Term Business Success and Failure*, Wharton School Publishing, Upper Saddle River, NJ, 2005.

(NATO 1968) Naur, P. Randell, B, *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7th to 11th October 1968, Naur, P., Randell, B., eds., 1969, online at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

Ohno, T., *Toyota Production System: Beyond Large-Scale Production*, Productivity Press, Portland, OR, 1988..

Poppendieck, M., Poppendieck, T., *Lean Software Development: An Agile Toolkit*, Addison-Wesley, Boston, Mam 2003.

Rich94: Rich, B, Janos, L., *Skunk Works: A Personal Memoir of My Years at Lockheed*, Little, Brown and Company, Boston, MA, 1994.

Reinertsen, D., *Managing the Design Factory: A Product Developer's Toolkit*, The Free Press, New York, NY, 1997.

(Schön 1983) Schön, D., *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, 1983.

Schwaber, K., Beedle, M., *Agile Software Development with Scrum*, Prentice-Hall, Upper Saddle River, NJ, 2002.

Stapleton, J., *DSDM: Business-Focussed Development, 2$^{nd}$ Edition*, Addison-Wesley, London, UK, 2003.

Weick, K., *The Social Psychology of Organizing, 2nd Edition*, McGraw-Hill, New York, NY, 1979.

Womack, J., Jones, D., Roos, D., *The Machine That Changed the World: The Story of Lean Production*, HarperPerennial, New York, NY, 1991.

(Wright 1953) Wright, O., *How We Invented the Airplane: An Illustrated History*, edited by F.C. Kelly, Dover, 1953.

Charan, R., "Home Depot's Blueprint for Culture Change", Harvard Business Review, April, 2006, pp. 60-70.

Rigby, D., Vishwanath, V., "Locatization: The Revolution in Consumer Markets", Harvard Business Review, April, 2006, pp. 82-92.

open

http://alistair.cockburn.us/crystal/talks/easq/efficiencyasaspendablequantity020.ppt, ICAM paper: Case Studies Motivating Efficiency as a "Spendable" Quantity

"Learning from Agile Software Development", parts 1 and 2, CrossTalk Magazine, Oct, 2002 (pp. 10-14) and Nov, 2002 (

CockburnEiE: [http://c2.com/cgi/wiki?ExpertInEarshot]

CockburnPrrP: [http://alistair.cockburn.us/crystal/articles/prrp/projectriskreductionpatterns.html]

CoplienOP: [http://www.bell-labs.com/cgi-user/OrgPatterns/OrgPatterns?OldOrgPatterns]

to do

cockburn process the 4th dimension

cockburn governance article

*Sketches of Thought*, Goel, V., MIT Press, Boston, MA, 1995.

ogunaike 1995

gladwell, *Blink*, ???2005

boehm, b, "Get ready for agile methods – with care," IEEE Computer, jan 2002? pp. ???

boehm, turner Balancing agility with discipline

goldratt critical chain

goldratt toc

ADC 2003 URL

Anderson book

Anderson experience report:

http://www.agilemanagement.net/Articles/Papers/BorConManagingLeanDevWithCFDs.pdf

http://www.agilemanagement.net/Articles/Papers/Feature_Driven_Development_-_towards_a_TOC__Lean__Six_Sigma_solution_v1_0.pdf

http://www.agilemanagement.net/Articles/Papers/From_Worst_to_Best_in_9_Months_Final_1_3.pdf

anderson: http://www.agilemanagement.net/Articles/Papers/Feature_Driven_Development_-_towards_a_TOC__Lean__Six_Sigma_solution_v1_0.pdf

http://www.agilemanagement.net/Articles/Papers/From_Worst_to_Best_in_9_Months_Final_1_3.pdf

Jeffries RTF URL

Larman iterative book

Brooks no silver bullet

inevitable illusions

co are iterations hazardous

Co A Governance Model For Agile and Hybrid-Agile Projects Alistair Cockburn, Humans and Technology, arc@acm.org, ©Alistair Cockburn HaT TR 2005.03, Sept. 23, 2005)

patton 2004 2005 articles

extreme hour url

walking skeleton ref --- CC

Orbanes, ?, "Everything I know about business I learned from Monopoly", Harvard Business Review, 2002.

Cumins, D., "The Development Game," Agile Development Conference, 2003, online at ...

Artful Making?

the end of s.e. and the start of cooper gaming

Mathiassen 2002 article Aalborg curriculum ?

Olson article

Goldratt The Goal ref

Anderson, D., *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results*, Prentice-Hall PTR, 2003.

Augustine, S., *Managing Agile Projects*, Prentice-Hall PTR, 2005.

Highsmith, J., *Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004.

DeCarlo, D., *eXtreme Project Management: Using Leadership, Principles, and Tools to Deliver Value in the Face of Volatility*, Jossey-Bass, 2004.

Schwaber, K., *Agile Project Management with Scrum*, Microsoft Press, 2004.

Wysocki, R., *Effective Project Management: Traditional, Adaptive, Extreme, Third Edition*, Wiley, 2003.

Ambler, S., *Agile Modeling*, ???

Cohn, M., *User Stories Applied*, Addison-Wesley, 2005???

Scott Ambler December 2005 issue of Software Development "How Agile Are You?",

http://www.sdmagazine.com/documents/s=9933/sdm0512h/0512h.html

Weick, K.

Anderson: Managing Lean Software Development with Cumulative Flow Diagrams, BorCon, 2004.

Glen Alleman's "Making Agile Development Work in a Government Contracting Environment" (Alleman 2003) [http://www.niwotridge.com/PDFs/ADC%20Final.pdf]

John Rusk's "Agile Charts" (Rusk url) [http://www.agilekiwi.com/agile_charts.htm]

Mike Cohn's *Agile Estimating and Planning* (Prentice-Hall PTR 2005)

(Constantine 2001) Constantine, L., Lockwood, L., "Structure and Style in Use Cases for User Interface Design", in van Harmelen, M. (ed.), *Object-Modeling and User Interface Design*, Addison-Wesley, 2001. Extract available online at http://www.foruse.com/articles/structurestyle2.pdf.

(Cockburn 1995sucwg) Cockburn, A., "Structuring Use Cases with Goals," online at http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm.

(ambler 2005.12) http://www.sdmagazine.com/documents/s=9933/sdm0512h/0512h.html patton 2004 unfixing the fixed price contract

Satir, V., "From Virginia Satir: Models of Perceiving the World–Attitude Toward Change", in *Couples Therapy in Managed Care: Facing the Crisis*, Brothers, B.J., ed., Haworth Press, 1999, p.4..

(Martin 2006), Martin, R., "Estimating Prices Up Front," in the comp.software.extreme-programming Google Groups discussion, Sep 26 2004 7:42 am, full URL = http://groups.google.com/group/comp.software.extreme-programming/browse_frm/thread/7a063ecc282d852a/9a203fad85f3d363?hl=en&lr=&rnum=1&prev=/groups%3Fq%3D%2522We%2520aren't%2520talking%2520about%2520throwing%2520anything%2520together.%26hl%3Den%26lr%3D%26selm%3Dhsedl0p0n9a0i5hunfu69l5ledcr03c483%25404ax.com%26rnum%3D1#9a203fad85f3d363

(cockburn 2002.10 crosstalk a) Cockburn, A., "Learning from Agile Development" parts 1 and 2, in *CrossTalk: The Journal of Defense Software Engineering*, online at http://www.stsc.hill.af.mil/crosstalk/2002/10/cockburn.html and http://www.stsc.hill.af.mil/crosstalk/2002/11/cockburn.html.

(Marcus 2006) Marcus, A., *Big Winners and Big Losers: The 4 Secrets of Long-Term Business Success and Failure*, Wharton School Publishing, 2006.

(Owen 1997) Owen, H., *Open Space Technology: A User's Guide*, 2nd edition, Berrett-Koehler Publishers, 1997.

(Coplien 2005) Coplien, J., Harrison, N., *Organizational Patterns Of Agile Software Development,* Prentice Hall, 2005.

126-80 = 46 pages left 7/03 7 pm
126-85 = 41 pages left 7/03 10 pm
135-91 = 44 pages left 7/04 6 pm
141-101 = 40 pages left 7/05 2 pm
143-112 = 31 pages left 7/05 5 pm
143-124+2 = 21 pages left 7/05 5 pm
13 pages left 7/06 6 pm