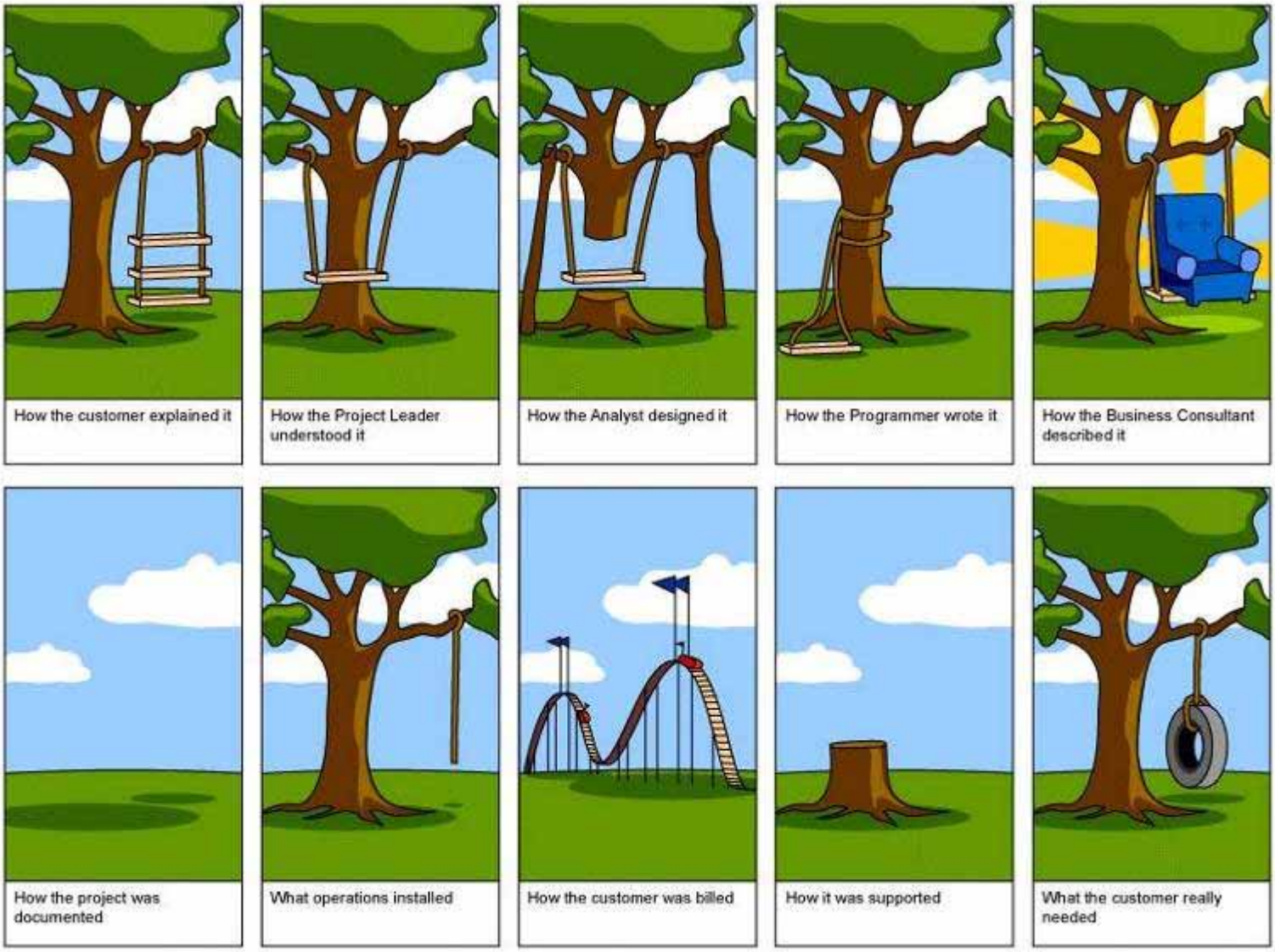
Agile Software Development meets SF

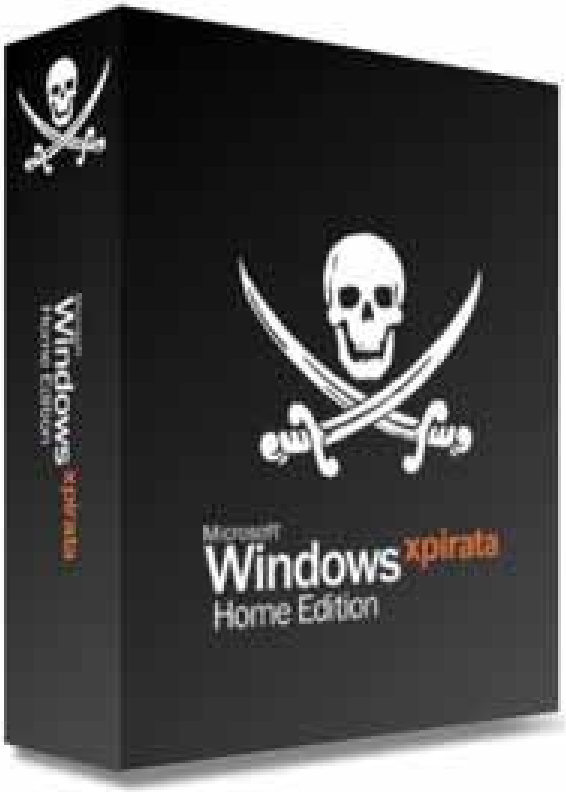
Karlstad Group Meeting, Vienna 2008

Hans-Peter Korn

How Software Development (often) works:



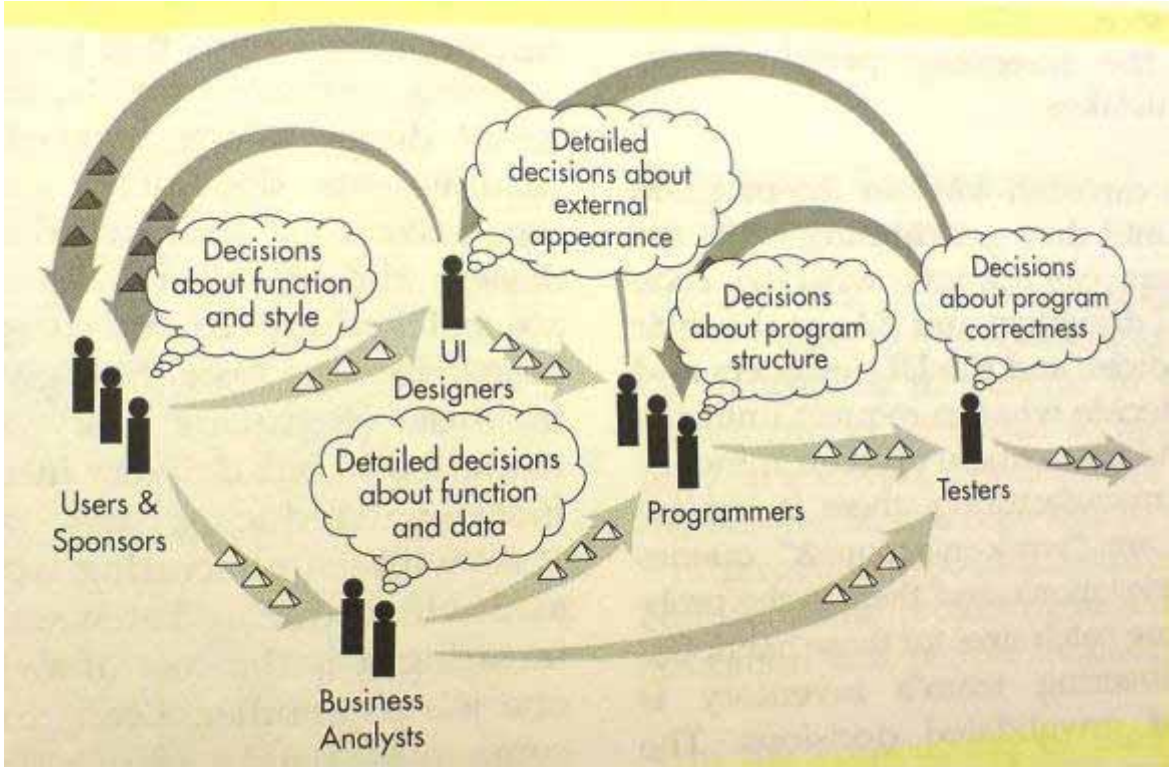
Hard work with Software:



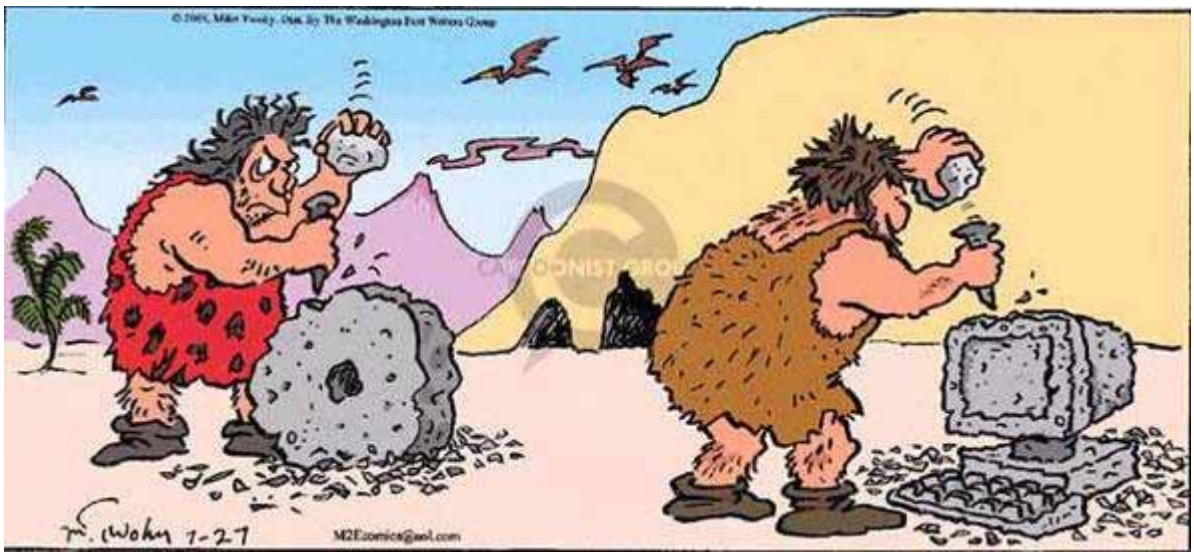
Developing (and using) IT-solutions means merging

➔ complicated but trivial technical systems

➔ with complex and not trivial social systems



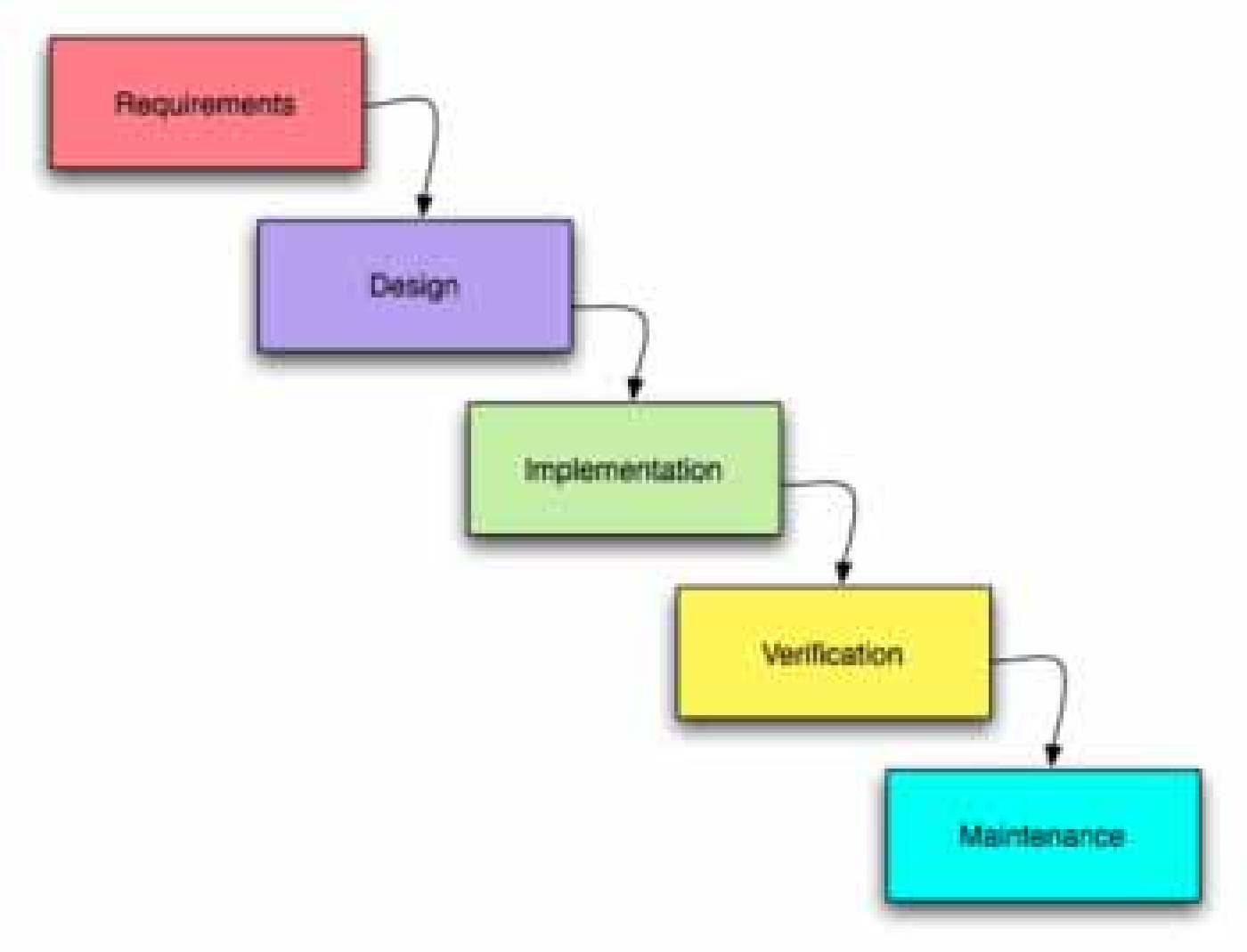
➔ ➔ forming a "hybrid" (= techno/social) system:



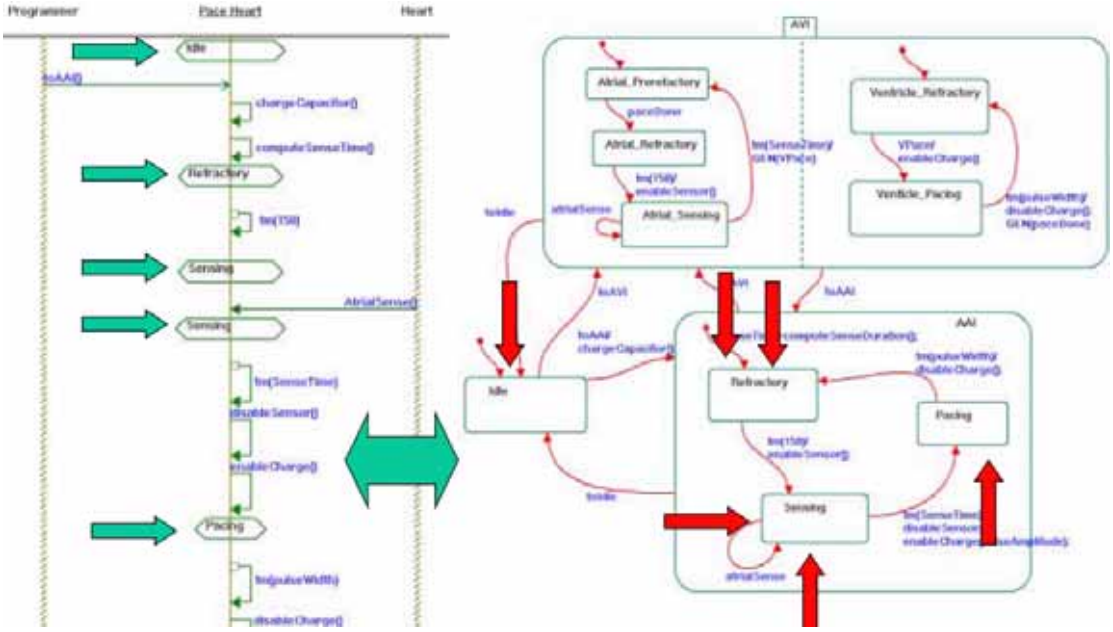
Treating such "hybrid systems" like trivial (technical) systems only is the most common less useful misunderstanding.

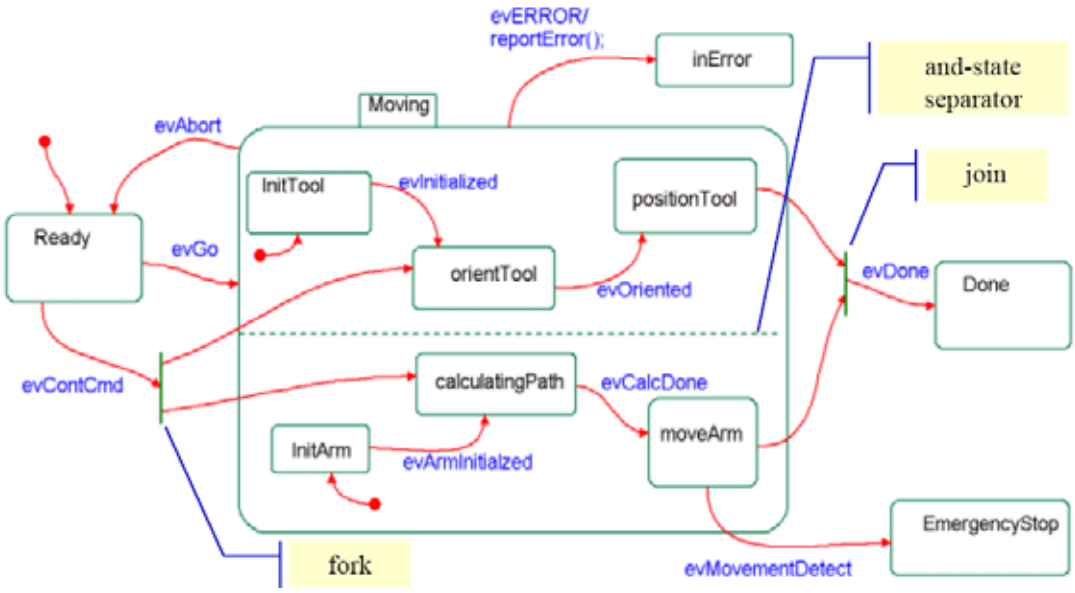
Examples for this misunderstanding:

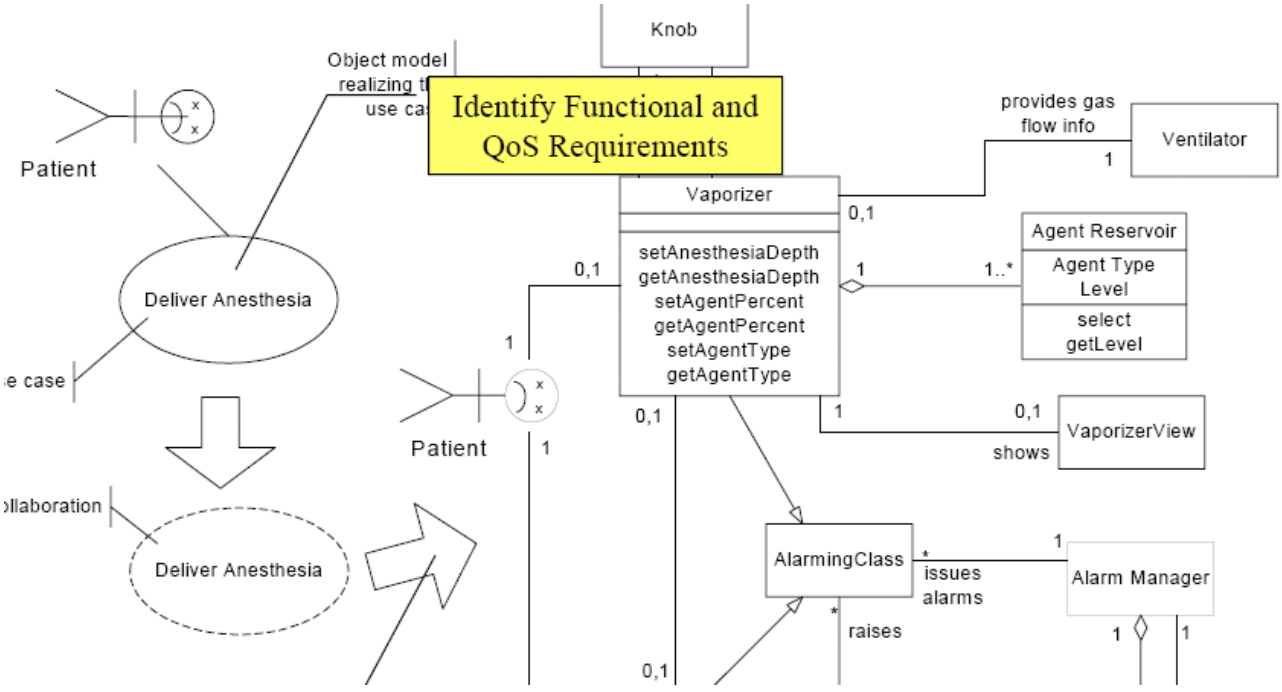
➔ **The "Waterfall-Model" states, that SW can be developed by one linear step by step process**



➔ **The requirements for and the design of the system can be and have to be documented and reviewed completely before the implementation is done**

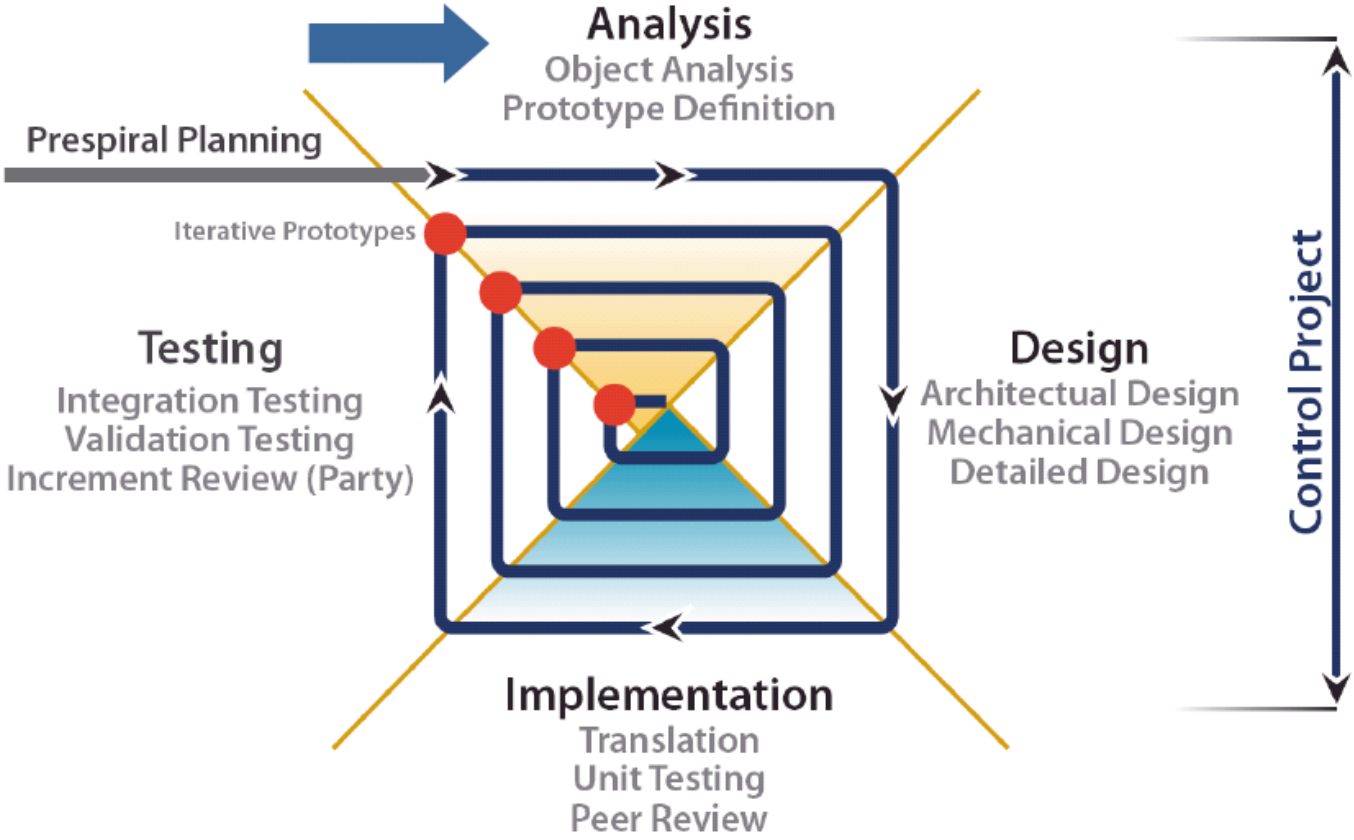






Earlier learning's:

➔ **"Bend" the waterfall to a spiral** (Barry W. Boehm, 1988)
and
➔ **try & enhance it with "iterative prototyping"**



Later learning's:

➔ **design & build the system incrementally: create first usable parts / functions asap!**

➔ **do it as simple as possible, use existing (buyable) solutions as far as possible**

➔ **pre-designed flexibility is good - AGILITY is even better!**

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others to do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, **we value the items on the left more.**

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

On February 11-13, 2001, at The Lodge at Snowbird ski resort in the Wasatch mountains of Utah, seventeen people met to talk, ski, relax, and try to find common ground and of course, to eat. What emerged was the Agile Software Development Manifesto. Representatives from Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, and others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes convened.

Now, a bigger gathering of organizational anarchists would be hard to find, so what emerged from this meeting was symbolic a Manifesto for Agile Software Development signed by all participants. The only concern with the term agile came from Martin Fowler (a Brit for those who don't know him) who allowed that most Americans didn't know how to pronounce the word 'agile'.

Alistair Cockburn's initial concerns reflected the early thoughts of many participants. "I personally didn't expect that this particular group of agilites to ever agree on anything substantive." But his post-meeting feelings were also shared, "Speaking for myself, I am delighted by the final phrasing [of the Manifesto]. I was surprised that the others appeared equally delighted by the final phrasing. So we did agree on something substantive."

Agility is the ability to change the body's position, and requires a combination of balance, coordination, speed, reflexes, and strength. (From: http://en.wikipedia.org/wiki/Agility)

Material to exercise the balance agility for children

Business agility is the ability of a business to change rapidly in response to varying economic conditions by producing high quality goods and services. (see: Nikos C. Tsourveloudi , Kimon P. Valavanis (2002). "On the Measurement of Enterprise Agility". *Journal of Intelligent and Robotic Systems* 33: 329-342.)
(From: http://en.wikipedia.org/wiki/Business_Agility)

Agile software development is a conceptual framework for software engineering that promotes **incremental** development **iterations** throughout the life-cycle of the project.
(The article: http://en.wikipedia.org/wiki/Agile_software_development offers a good first glance on agile software development. Some of the following text is from this article)

"Agile software development" evolved in the mid 1990s as part of a reaction against "heavyweight" methods, as typified by a heavily regulated, regimented, micro-managed use of the waterfall model of development. In 2001, prominent members of that community adopted the name "agile methods". Later, some of these people formed "The Agile Alliance", a non-profit organization that promotes agile development. They created the "Agile Manifesto", a canonical definition of agile development and accompanying agile principles.

Agile methods are a family of development processes, not a single approach to software development. Most of them aim to **minimize risk by developing software in short amounts of time** by incremental iterations which may last from one to four weeks. Each incremental iteration is a small entire software project including planning, requirements analysis, design, coding, testing, and documentation with an available release (without bugs) at the end of each iteration. At the end of each iteration, the team re-evaluates project priorities.

Agile methods emphasize **face-to-face communication** over written documents. Agile methods emphasize **working software as the primary measure of progress**. Agile methods therefore produce very little written documentation relative to other methods. This has resulted in criticism of agile methods as being undisciplined. An answer to this criticism is, phrased by Alistair Cockburn as one of the "Agilistas", to see software development as a **"cooperative game of communication and invention".** He grounded this view on Pelle Ehn's "Work-Oriented Development of Software Artefacts"(1988) who considered software development in the context of the philosophers Descartes, Marx, Heidegger and Wittgenstein. Considering this it turns out, that software development can be understood as a "cooperative language game". This understanding changes the character and importance of "documentation" dramatically: The documents are not longer (intermediate) result of software development, they "only" serves as "design tools" (among others, like mock-ups and screen-prototypes) to support the communication (e.g. between developers and users) to co-create a shared understanding how the IT-application should work. The working application and not the documented descriptions of requirements is the only relevant result of the design.

Seeing and DOING software design as a cooperative language game (=> Wittgenstein) **of communication and invention is one of the very important bricks of that platform which is shared with SF.**

Further key elements of Agile Software Development relating to SF:

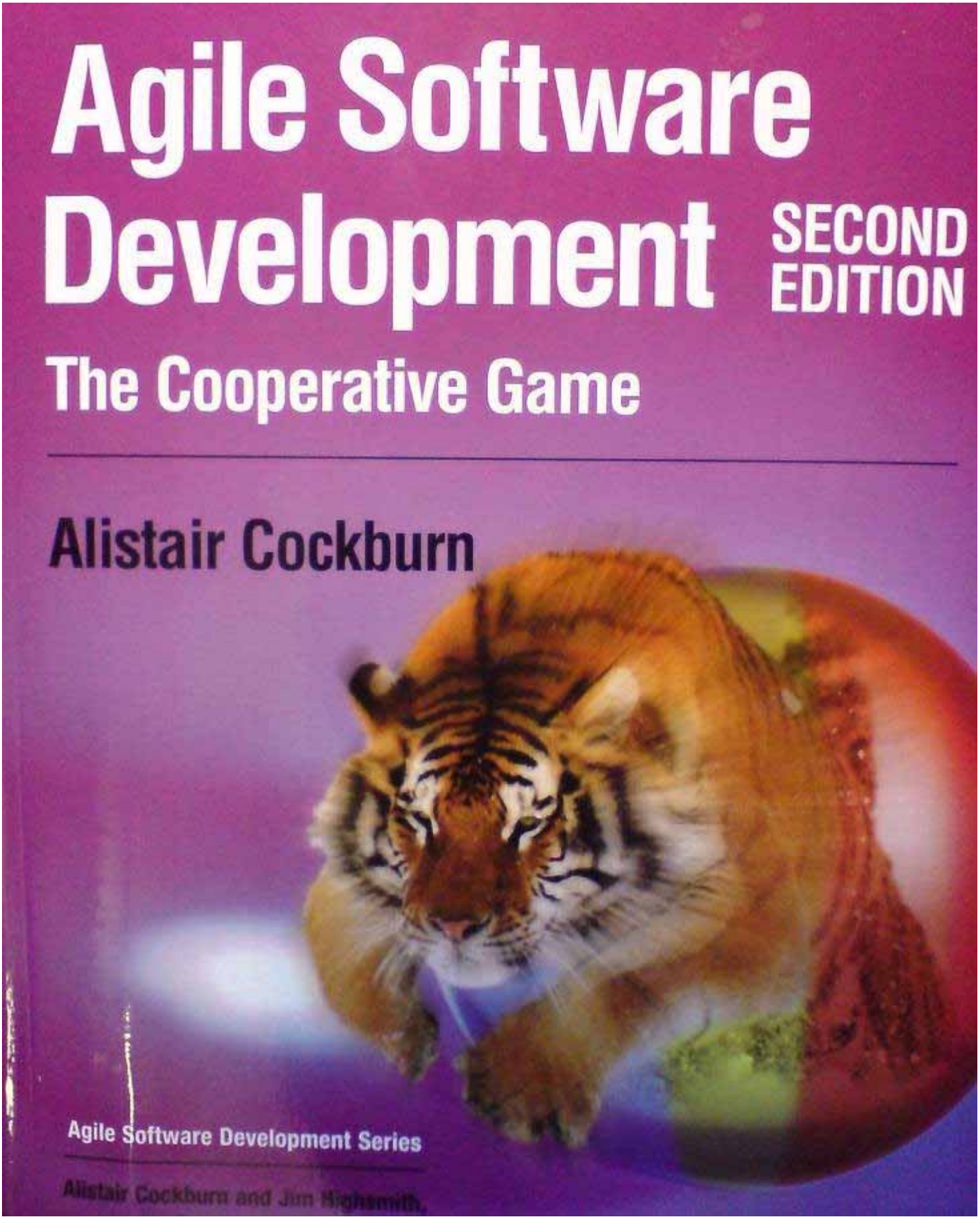
>>> Principles behind the Agile Manifesto <<<
(see: http://www.agilemanifesto.org/principles.html)

- ***Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.***
  ➔ small steps with observable effects / results
- ***Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.***
  ➔ Change is occurring all the time.
- ***Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.***
  ➔ Putting positive difference to work; Small Actions - tiny next steps that make big differences
- ***Business people and developers must work together daily throughout the project.***
  ➔ In-between - the action is in the interaction
- ***Build projects around motivated individuals.***
  ➔ Clients are always cooperating. They are showing us how they think change takes place. As we understand their thinking and act accordingly, cooperation is inevitable.
- ***Give them the environment and support they need, and trust them to get the job done.***
  ➔ People have all they need to solve problems
- ***The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.***
  ➔ the action is in the interaction; Meaning and experience are interactional constructed. We inform meaning onto our experience and it is our experience at the same time. Meaning is not imposed from without or determined from outside of ourselves. We in-form our world through interaction.
- ***Working software is the primary measure of progress.***
  ➔ Make use of what's there - not what isn't. Not heavy concepts but small changes leads to larger changing step by step.
- ***Agile processes promote sustainable development.***
  ➔ Every case is different - beware ill-fitting theory
- ***The sponsors, developers, and users should be able to maintain a constant pace indefinitely.***
  ➔ the action is in the interaction
- ***Continuous attention to technical excellence and good design enhances agility.***
  ➔ Counters - whatever helps us forward; Affirm - what's already going well?
- ***Simplicity -- the art of maximizing the amount of work not done -- is essential.***
  ➔ Radical simplicity
- ***The best architectures, requirements, and designs emerge from self-organizing teams.***
  ➔ the action is in the interaction
- ***At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.***
  ➔ Counters - whatever helps us forward; Affirm - what's already going well?

Further reading

http://www.agilealliance.org/ and http://apln.org/ and http://pmdoi.org
http://en.wikipedia.org/wiki/Agile_software_development
http://www.extremeprogramming.org/
Alistair Cockburn: Agile Software Development - The Cooperative Game (2nd Edition), Addison-Wesley Professional; 2 edition (2006)

The following text is part of this book:



BASICS:

The ideas in this book are based on 25 years of development experience and 10 years of investigating projects directly.

The IBM Consulting Group asked me to design its first object-oriented methodology in 1991. I looked rather helplessly at the conflicting "methodology" books at the time. My boss, Kathy Ulisse, and I decided that I should debrief project teams to better understand how they really worked. What an eye-opener! The words they used had almost no overlap with the words in the books.

The interviews keep being so valuable that I still visit projects with sufficiently interesting success stories to find out what they encountered, learned, and recommend. The crucial question I ask before the interview is, "And would you like to work the same way again?" When

AGILITY

I am not the only person who is using these ideas:

- Kent Beck and Ward Cunningham worked through the late 1980s on what became called *Extreme Programming* (XP) in the late 1990s.
- Jim Highsmith studied the language and business use of complex adaptive systems in the mid-1990s and wrote about the application of that language to software development in his *Adaptive Software Development*.
- Ken Schwaber and Jeff Sutherland were constructing the Scrum method of development at about the same time, and many project leaders made similar attempts to describe similar ideas through the same years.

When a group of us met in February 2001 to discuss our differences and similarities, we found we had a surprising number of things in common. We selected the word *agile* to describe our intent and wrote the Agile Software Development Manifesto (Appendix A).

We are still formulating the principles that we share and are finding many other people who could have been at that meeting if they had known about it or if their schedules had permitted their presence.

Core to *agile* software development is the use of light-but-sufficient rules of project behavior and the use of human- and communication-oriented rules.

Agility implies maneuverability, a characteristic that is more important now than ever. Deploying software to the Web has intensified software competition further than before. Staying in business involves not only getting software out and reducing defects but tracking continually [illegible] increasingly demands. Winning in business increasingly involves winning at the software-development game. Winning at the game depends on understanding the game being played.

The best description I have found for *agility* in business comes from Goldman (1997):

> "Agility is dynamic, context-specific, aggressively change-embracing, and growth-oriented. It is not about improving efficiency, cutting costs, or battening down the business hatches to ride out fearsome competitive 'storms.' It is about succeeding and about winning: about succeeding in emerging competitive arenas, and about winning profits, market share, and customers in the very center of the competitive storms many companies now fear."

METHODOLOGIES: SPECIFIC, SIMPLE, PROOFED

We base the series on these two core ideas:

- Different projects need different processes or methodologies.
- Focusing on skills, communication, and community allows the project to be more effective and more agile than focusing on processes.

agile methodologies. Whoever is selecting a base methodology to tailor will want to find one that has already been used successfully in a similar situation. Modifying an existing methodology is easier than creating a new one and is more effective than using one that was designed for a different situation. *Crystal Clear* (Cockburn, forthcoming) is a sample methodol-

ACCEPTING THE INCOMPLETENESS OF COMMUNICATION

Knowing that perfect communications are impossible relieves you of trying to reach that perfection. Instead, you learn to manage the incompleteness of communication. Rather than try to make the requirements document or the design model comprehensible to everyone, you stop when the document is sufficient to the purpose of the intended audience. "Managing the incompleteness of communications" is core to mastering agile software development.

THINKING INEXACT THOUGHTS

We don't notice what is in front of us, and we don't have adequate names for what we do notice. But it gets worse: When we go to communicate, we don't even know exactly what it is we mean to communicate.

Pelle Ehn describes software design similarly. Recognizing that neither the users nor the designers could adequately identify, parse, and name their experiences, he asked them to *design by doing*. In the article reproduced in Appendix B he writes:

> "The language-games played in design-by-doing can be viewed both from the point of view of the users and of the designers. This kind of design becomes a language-game in which the users learn about possibilities and constraints of new computer tools that may become part of their ordinary language-games. The designers become the teachers that teach the users how to participate in this particular language-game of design. However, to set up these kinds of language-games, the designers have to learn from the users.
>
> "However, 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."

A Cooperative Game of Invention and Communication

A fruitful way to think about software development is to consider it as a cooperative game of invention and communication.

So, What Do I Do Tomorrow?

The mystery is that we can't get perfect communication. The answer to the mystery is that we don't need perfect communication. We just need to get close enough, often enough.

In the second article in the appendix, Pelle Ehn describes the difficulty he and his programmers had communicating with their computer-näive 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 416).

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.

A Game of Invention and Communication

We have seen that software development is a *group* game, which is *goal seeking, finite,* and *cooperative*. The team, which consists of the sponsor, the manager, usage specialists, domain specialists, designers, testers, and writers, works together with the goal of producing a working and useful system. In most cases, team members aim to produce the system as quickly as possible, but they may prefer to focus on ease of use, cost, defect freedom, or liability protection.

The game is finite because it is over when the goal is reached. Sometimes delivery of the system marks the termination point; sometimes the end comes a bit later. Funding for development usually changes around the time the system is delivered, and new funding defines a new game. The next game may be to improve the system, to replace the system, to build an entirely different system, or possibly to disband the group.

The game is cooperative because the people on the team help each other to reach the goal. The measure of their quality as a team is how well they cooperate and communicate during the game. This measure is used because it affects how well they reach the goal.

If it is a *goal-directed cooperative game,* what does the game consist of? What constitutes moves in the game?

The task facing the developers is this: They are working on a problem that they don't fully understand, one that lives in emotions, wishes, and thoughts and that changes as they proceed. They need to

- Understand the problem space
- Imagine some mechanism that solves the problem in a viable technology space
- Express that mental construct in an executable language, which lacks many features of expression, to a system that is unforgiving of mistakes

To work through this situation, they

- Use props and devices to pull thoughts out of themselves or to generate new ideas that might help them understand the problem or construct a solution
- Leave trails of markers for those who will come later, markers to monitor and test their progress and their understanding, and they use those markers again, themselves, when they revisit parts of their work

Software development is therefore a cooperative game *of invention and communication*. There is nothing in the game but people's ideas and the communication of those ideas to their colleagues and to the computer.

Looking back at the literature of our field, we see a few people who have articulated this before. Peter Naur did, in his 1985 article "Programming as Theory Building," and Pelle Ehn did, in "Scandinavian Design: On Participation and Skill" (1992) and in his magnificent but out-of-print book *Work-Oriented Design of Software Artifacts* (1988). Naur and Ehn did this so well that I include those two articles in near entirety in Appendix B. Robert Glass and colleagues wrote about it in "Software Tasks: Intellectual or Clerical?" (1992), and Fred Brooks saw it as such a wickedly hard assignment that he wrote the article "No Silver Bullet" (1995).

SOME PHILOSOPHY ON BACKSTAGE

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. 49), 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 this idea 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 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 it may never need to be perfect. 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 that 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 other, and the ways they carry out their jobs.

PELLE EHN, WITTGENSTEIN'S LANGUAGE GAMES

In *Work-Oriented Development of Software Artifacts* (Ehn 1988), Pelle Ehn describes a series of projects that explored ways of making software more appropriate to its final use, easier to use, and made by both programmers and end users.

The high point of the book for me is the way in which he considers software development in the context of four philosophers: Descartes, Marx, Heidegger, and Wittgenstein.

A person working in the style of Descartes thinks of an external reality worth describing and turns her efforts toward capturing that reality. She is therefore interested in the match to reality of the requirements, models, and code. This Cartesian approach filled our field's first half-century.

A person working in the style of Marx first asks, "Whom does this new system benefit? How does its deployment change the social power structure?" This is a meaningful question to consider, whether you like Marx's political theories or not.

A person working in the style of Heidegger considers the efficacy of the system as a tool. Ideally, the user should not "see" the system at all. She should see through the system to the task being performed. When I am typing a document, for example, I see the page growing text; I don't "see" the word processor. An accomplished pianist sees the music being formed, not the piano; a good carpenter sees the nail going into the wood, not the hammering tool. Heidegger's frame of evaluation helps us produce systems more fit for use.

It is only the style of Wittgenstein that opposes the style of Descartes. A person working in this style views the unfolding of the software design as the unfolding of a language game, in which new words are added to the language over time.

This immediately links to software development as a cooperative game of invention and communication. I probably owe a good deal of my construction of the cooperative game model to Ehn's writings. I had read and forgotten the following article years before working out the cooperative game idea. As I started to write this book, I reviewed this article and was shocked to see how many of my words echoed Ehn's.

Ehn is concerned with the building of shared experience through shared practice, of using practice directly as a basis for discovering needs. In other words, he is working with tacit knowledge. More than that, he highlights the place of *skill* in carrying out practices (it is interesting to read Musashi's words pointing out much the same). Although skill is a topic I have mentioned, Ehn develops it much more thoughtfully and completely.

I took the game thinking in a different direction. I am concerned with playing a group game amicably, so that communication can take place at all. You will see that Ehn's ideas complement the rest of the ideas in this book.

Pelle Ehn expresses it much better in his own words than I can through summaries. *Work-Oriented Development of Software Artifacts* is out of print, sadly. However, this excerpt from "Scandinavian Design: On

ordinary language philosophy of Ludwig Wittgenstein. My focus is on the shift in design from language as description towards language as action.

Rethinking Systems Descriptions

descriptions.

Our experiences with the UTOPIA project caused me to re-examine my philosophical assumptions. Working with the end users of the design, the graphics workers, some design methods failed while others succeeded. *Requirement specifications and systems descriptions based on information from interviews were not very successful.* Improvements came when we made joint visits to interesting plants, trade shows, and vendors and had discussions with other users; when we dedicated considerably more time to learning from each other, designers from graphics workers and graphics workers from designers; *when we started to use design-by-doing methods and descriptions such as mock-ups and work organization games*; and when we started to understand and use traditional tools as a design ideal for computer-based systems.

Practice Is Reality

Practice as the social construction of reality is a strong candidate for replacing the picture theory of reality. In short, practice is our everyday practical activity. It is the human form of life. It precedes subject-object relations. Through practice, we produce the world, both the world of objects and our knowledge about this world. Practice is both action and reflection. But practice is also a social activity; it is produced in cooperation with others. To share practice is also to share an understanding of the world with others. However, this

Against this background, we can understand the design of computer applications as a concerned social- and historical-conditioned activity in which tools and their use are envisioned. This is an activity and form of knowledge that is both planned and creative.

Once struck by the "naive" Cartesian presumptions of a picture theory, what can be gained in design by shifting focus from the correctness of descriptions to intervention into practice? What does it imply to take the position that what a picture describes is determined by its use? Most importantly, *it sensitizes us to the crucial role of skill and participation in design*, and to the opportunity in practical design to transcend some of the limits of formalization through the use of more action-oriented design artifacts.

Language as Action

To master the professional language of chairmaking means to be able to act in an effective way together with other people who know chairmaking. To "know" does not mean explicitly knowing the rules you have learned, but rather recognizing when something is done in a correct or incorrect way. To have a concept is to have learned to follow rules as part of a given practice. Speech acts are, as a unity of language and action, part of practice. They are not descriptions but below I will

Language-Games

To use language is to participate in language-games. In discussing how we in

Language-games, like the games we play as children, are social activities. To be able to play these games, we have to learn to follow rules, rules that are socially created but far from always explicit. *The rule-following behavior of being able to play together with others is more important to a game than the specific explicit rules.* Playing is interaction and cooperation. *To follow the rules in practice means to be able to act in a way that others in the game can understand.*

Knowledge and Design Artifacts

As designers we are involved in reforming practice, in our case typically computer-based systems and the way people use them. Hence, the language-games of design change the rules for other language-games, in particular those of the application's use. What are the conditions for this interplay and change to operate effectively?

A common assumption behind most design approaches seems to be that the users must be able to give complete and explicit descriptions of their demands. Hence, the emphasis is on methods to support this elucidation by means of requirement specifications or system descriptions (Jackson, 1983; Yourdon, 1982).

In a Wittgensteinian approach, the focus is not on the "correctness" of systems descriptions in design, on how well they mirror the desires in the mind of the users, or on how correctly they describe existing and future systems and their use. Systems descriptions are design artifacts. In a Wittgensteinian approach, the crucial question is how we use them, that is, what role they play in the design process.

The rejection of an emphasis on the "correctness" of descriptions is especially important. In this, we are advised by the author of perhaps the strongest arguments for a picture theory and the Cartesian approach to design—the young Wittgenstein in Tractatus Logico-Philosophicus (1923). The reason for this rejection is the fundamental role of practical knowledge and creative rule following in language-games.

Nevertheless, we know that systems descriptions are useful in the language-game of design. The new orientation suggested in a Wittgensteinian approach is that we see such descriptions as a special kind of artifact that we use as "typical examples" or "paradigm cases." *They are not models in the sense of Cartesian mirror images of reality* (Nordenstam, 1984). *In the language-game of design, we use these tools as reminders for our reflection on future computer applications and their use. By using such design artifacts, we bring earlier experiences to mind, and they bend our way of thinking of the past and the future.* I think that this is why we should understand them as representations (Kaasboll, forthcoming). And this is how they inform our practice. *If they are good design artifacts, they will support good moves within a specific design language-game.*

The meaning of a design artifact is its use in a design language-game, not how it "mirrors reality." Its ability to support such use

They could be experienced through the practical use of a prototype or mockup. This experience could be further reflected upon in the language-game of design

Design by Doing: New "Rules of the Game"

What do we as designers have to do to qualify as participants in the language-games of the users? What do users have to learn to qualify as participants in the language-game of design? And what means can we develop in design to facilitate these learning processes?

If designers and users share the same form of life, it should be possible to overcome the gap between the different language-games. It should, at least in between the two language-games.

What kind of design tools could support this interplay between language-games? I think that what we in the UTOPIA project called design-by-doing methods—prototyping, mockups, and scenarios—are good candidates. Even joint visits to workplaces, especially ones similar to the ones being designed for, served as a kind of design tool through which designers and users bridged their language-games.

The language-games played in design-by-doing can be viewed both from the

However, 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. Wittgenstein (1953) notes that "when children play at trains their game is connected with their knowledge of trains. It would nevertheless be possible for the children of a tribe unacquainted with trains to learn this game from others, and to play it without knowing that it was copied from anything. One might say that the game did not make the same sense as to us." As long as the language-game of design is not a nonsense activity to any participant but a shared activity for better understanding and good design, mutual understanding may be desired but not really required.

understanding. Not only could, for example, the typographer working at the mockup tell that the screen should be bigger to show a full page spread—something important in page makeup—he could also show what he meant by "cropping a picture" by actually doing it as he said it. It was thus possible for him to express his practical understanding, his sensuous knowledge by familiarity. He could, while working at the mockup, express the fact that when the system is designed one way he can get a good balanced page, but not when it is designed another way.

Finding a solution that works adequately in a 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, a professor at MIT, refers to this constant reframing as having a "reflective conversation with the situation." In *The Reflective Practitioner* (Schön 1983), he catalogs, among others, chemical engineers tackling a new problem. These chemical engineering graduate students were asked to create a manufacturing process to replicate a particular patina on china that came from the use of cow bone, which would soon 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 that they could replicate them. They got completely bogged down using this approach.
- Abandoning theory as too difficult, they next experimented 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

TREATING "HYBRID" SYSTEMS IN A TRIVIAL WAY ONLY - HANDLING MISTAKES CONSTRUCTIVELY

People mistakenly think that since engineers use mathematics in their craft, their predictions about how long a project will take (or how much it will cost) will be similarly accurate. However, civil engineers fail in the same way as the average software developer when put in similar situations. 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 <http://www.revelation13.net/bigdig.html>). The 600% cost overrun was attributed to the project being larger than previous projects of this sort and to its use of new and untried technologies.

Making Mistakes

That people make mistakes is, in principle, no surprise to us. Indeed, that is exactly why *iterative* and *incremental* development were invented.

Iterative refers to a scheduling and staging strategy that lets you rework pieces of the system.

Iterative development lets the team learn about the requirements and design of the system. Grady Booch calls this sort of learning "gestalt, round-trip design" (1994), a term that emphasizes the human characteristic of learning by completing.

Iterative schedules are difficult to plan, because it is hard to guess in advance how many major learnings will take place. To get past this difficulty, some planners simply fix the schedule to contain three iterations: draft design, major design, and tested design.

Incremental refers to a scheduling and staging strategy in which pieces of the system are developed at different rates or times and integrated as they are developed.

Incremental development lets the team learn about its own development process as well as about the system being designed. After a section of the system is built, the team members examine their working conventions to find out what should be improved. They might change the team structure, the techniques, or the deliverables.

Incremental is the simpler of the two methods to learn, because cutting the project into subprojects is not as tricky as deciding when to stop improving the product. Incremental development is a critical success factor for modern projects (Cockburn 1998).

The very reason for incremental and iterative strategies is to allow for people's inevitable mistakes to be discovered relatively early and repaired in a tidy manner.

That people make mistakes should really not be any surprise to us. And yet, some managers seem genuinely surprised when the development team announces a plan to work according to an incremental or iterative process. I have heard of managers saying things like

> "What do you mean, you don't know how long it will take?"

or

> "What do you mean, you plan to do it wrong the first time? I can go out and hire someone who will promise to do it right the first time."

In other words, the manager is saying that he expects the development team not to make any major mistakes or to learn anything new on the project.

One can find people who promise to get things right the first time, but one is unlikely to find people who actually get things right the first time. People make mistakes in estimation, requirements, design, typing, proofreading, installing, testing . . . and everything else they do. There is no escape. We must accept that mistakes will be made and use processes that adjust to the fact of mistakes.

Given how obvious it is that people make mistakes, the really surprising thing is that managers still refuse to use incremental and iterative strategies. I will argue that this is not as surprising as it appears, because it is anchored in two failure modes of humans: preferring to fail conservatively rather than risk succeeding differently, and having difficulty changing working habits.

Preferring to Fail Conservatively

There is evidence that people generally are risk-averse when they have something in their hands that they might lose and risk-accepting if they are in the process of losing something and may have a chance to regain it (Piattelli-Palmarini 1996).

Piattelli-Palmarini describes a number of experiments involving risks and rewards. T

WE DECIDE FOR CHANCES, NOT FOR RISKS:

Illusions of Choice

Piattelli-Palmarini cites a dual experiment. In the first, people are given $300 and then have to choose between a guaranteed $100 more or a 50/50 chance at $200 more.

People prefer to take the guaranteed $100.

In the second, people are given $500 and then have to choose between having $100 taken away from them or a 50/50 chance of having $200 taken away from them.

People prefer to risk having $200 taken from them.

(Piattelli-Palmarini, p. 58)

Mathematically, all outcomes are equal. What is interesting is the difference in the outcomes depending on how the problem is stated.

Piattelli-Palmarini sums up the aspect relevant to project managers: *We are risk-averse when we might gain.*

MAKE MORE OF THAT WHAT WORKS - BUT IN A DIFFERENT WAY FOR DIFFERENT SITUATIONS:

Improve your environment:

- Collect a few work samples: an example of some good code, a well-written class comment, use case, project plan, meeting minutes, design memo, or user interface.
- Enlist a few others to do this, and put the small collection of work samples online for everyone to copy from.
- Reduce interruptions. Create a small period each day, just two hours long, in which you don't take interruptions. See if a larger group in your office will do the same.
- Reduce the need for mechanisms that rely on people's weaknesses.
- Increase the use of mechanisms that draw on people's strengths, and let them use their talents.

Many people think I want people to *always* 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 Interdependence (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 analogy is chess, with its different opening, mid-game, and end-game strategies; the same applies to projects).

COMMUNICATE & RADIATE "OSMOTIC":

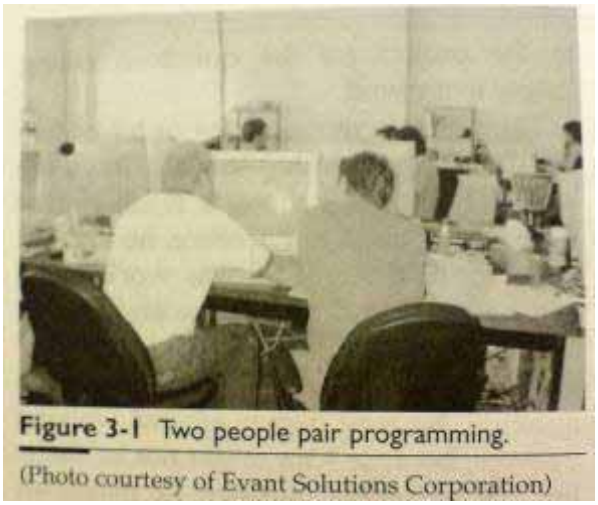


Figure 3-1 Two people pair programming.

(Photo courtesy of Evant Solutions Corporation)

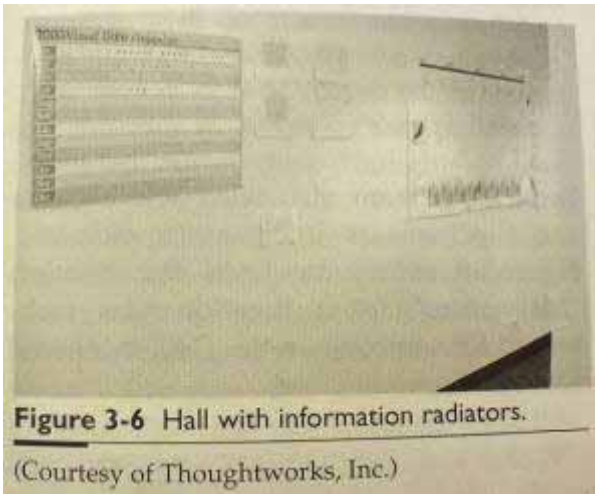


Figure 3-6 Hall with information radiators.

(Courtesy of Thoughtworks, Inc.)

Figure 3-8 Large information radiator wall showing the iteration plan, one flipchart per user story.

The wording in the posters matters. One XP team had posted "Things we did wrong last increment." Another had posted "Things to work on this increment." Imagine the difference in the projects: The first one radiated guilt into the project room and was, not surprisingly, not referred to very much by the project team. The second one radiates promise. The people on the second team referred to their poster quite frequently when talking about their project.

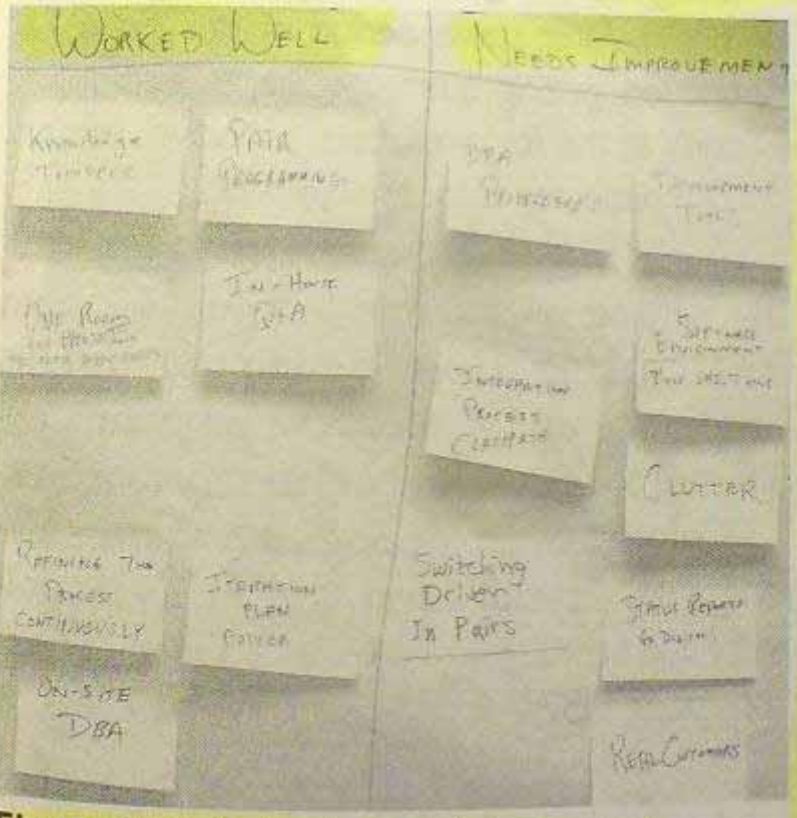


Figure 3-10 Reflection workshop output.

Programmers give mixed reviews to outside-of-work team-building exercises. Several said, roughly, "I'm not interested in whether we can barbeque together or climb walls together. I'm interested in whether we can produce software together."

What *does* build teams? Luke Hohmann offered this observation in an e-mail note:

> "The best way to build a team is by having them be successful in producing results. Small ones, big ones. It doesn't matter. This belief has empirical support; see, for instance, Brown (1990). Fuzzy team building is (IMO) almost always a waste of time and money."

Support for this is also found in Weick's description of the importance of "small wins" (Weick 2001) as well as in interviews of successful project managers.

One successful project manager told of a key moment when the project morale and "team"-ness improved. We found the following elements in the story:

- The people, who sat in different locations, met each other face to face.
- Together, they accomplished some significant result that they could not have achieved without working together.
- At some point, they placed themselves in some social jeopardy (venturing new thoughts, or admitting ignorance) and received support from the group when they might have been attacked.

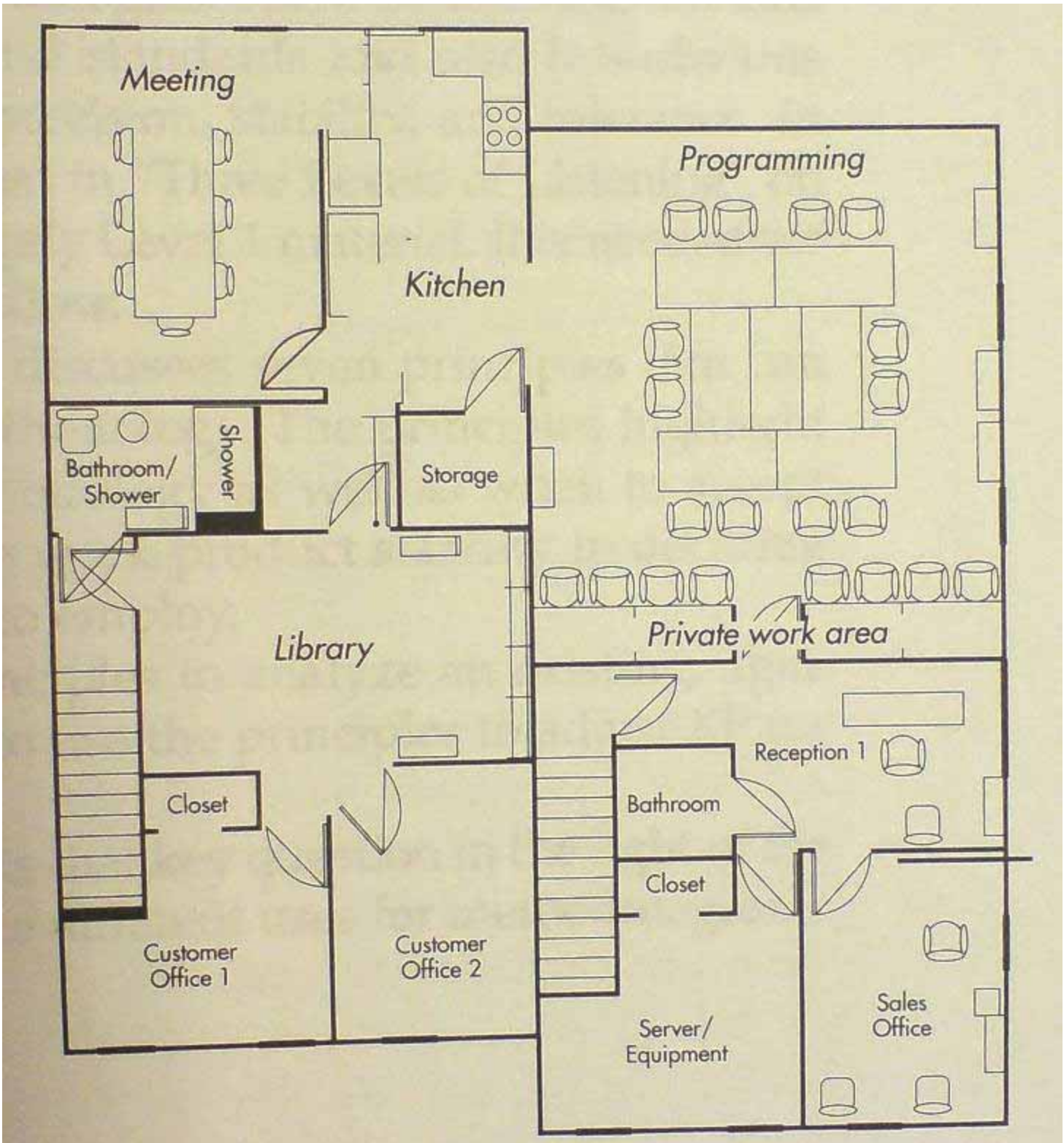


Figure 3.1-1 Completed office layout (Courtesy of Ken Auer, RoleModel Software).

SOCIAL CONSTRUCTED METHODOLOGY:

"Methodology is a social construction," Ralph Hodgson told me in 1993. Two years went by before I started to understand.

Your "methodology" is everything you regularly do to get your software out. It includes who you hire, what you hire them for, how they work together, what they produce, and how they share. It is the combined job descriptions, procedures, and conventions of everyone on your team. It is the product of your particular ecosystem and is therefore a unique construction of your organization.

Boil and condense the subject of methodology long enough and you get this one-sentence summary: "A methodology is the conventions that your group agrees to."

HOW ALL THIS WORKS - AN EXAMPLE OUT OF REAL LIFE:

XP UNDER GLASS

Extreme Programming (XP) is an *agile* methodology that illustrates the ideas in this book very well. Additionally, it is effective, well documented, and controversial. Thus, it makes a wonderful sample methodology to examine. At this point, we finally have enough vocabulary to put it under the methodology microscope.

The short story is that XP scores very high within its area of applicability. It (like all others) needs to be adjusted when applied outside its sweet spot.

XP IN A NUTSHELL

The brief review of XP is in order (Beck 2000, Jeffries 2001, XP <http://extremeprogramming.com>). Following is a summary, as brief as it would be if given as instructions over the phone or email:

Use only 3 to 10 programmers. Arrange for one or several customers to be on site to provide ongoing expertise. Everyone works in one room or adjacent rooms, preferably with the workstations clustered, monitors facing outward in a circle, half as many workstations as programmers.

Do development in three-week periods, or *iterations*. Each iteration results in running, tested code that is of direct use to the customers. The compiled system is rolled out to its end users at the end of each release period, which may be every two to five iterations.

The unit of requirements gathering is the "user story," user-visible functionality that can be developed within one iteration. The customers write the stories

for the iteration onto simple index cards. The customer(s) and programmers negotiate what will get done in the next iteration in the following way:

- The programmers estimate the time to complete each card.
- The customers prioritize, alter, and de-scope as needed so that the most valuable stories are most likely to get done in the allotted time period.

The programmers write the tasks for each story on flipcharts on the wall or a whiteboard, estimating the time they will need for each task. Over time, the customers and programmers can reprioritize or descope the tasks or stories.

Development on a story starts with the programmers discussing the story with the expert customer. Because this discussion is guaranteed to take place, the text written on the story card can be very brief, just enough to remind everyone of what the conversation is going to be about. The understanding of the requirements grow through those conversations and any pictures or documents the people decide they need.

Programmers work in pairs. They follow strict coding standards that they set up at the beginning of the project. They create unit tests for everything they write and make sure that those tests run at 100 percent every time they check in their code to the mandatory versioning and configuration-management system. They develop in tiny increments of 15 minutes to a few hours long, integrating their code several times a day. At the end of each of these integrations, they ensure that the entire code base passes all unit tests.

At any time, any two programmers sitting together may change any line of code in the system. In fact, they are supposed to. Anytime the two find a section of code that appears hard to understand or overly complex, they are to revise it, constantly simplifying and improving it. At all times, they are to keep the overall design as simple as they can and the code as clear as they can. This constant refactoring is possible because of the extensive unit test suites in place. It is also possible because the programmers rotate pair assignments every day or so, and so knowledge of the changes in the code structure passes through the group through the shifting partnerships.

While the programmers are working, the customers are doing three things: They visit with the programmers to clarify ideas, they write system acceptance tests to be run during and at the end of the iteration, and they select stories to be built for the next iteration. They may be on the project full time or not, as they decide.

The team holds a stand-up meeting every day, in which they describe what they are working on, what is working well for them, and what they might need help with. The meeting is held standing up to keep it short. At the end of each iteration, they hold another meeting in which they review what they did well and what they want to work on next time. They post this list for all to see during the next iteration.

XP prizes four values: communication, simplicity, testing, and courage. The "courage" value is intended as courage to go

ahead and make improvements to the system at any time.

One person on the team is designated the "coach" for the team. This person reviews with the team members their use of the key practices: use of pair programming and testing, pair rotation, keeping design simple, communicating, and so on.

WRAPPING UP:

I like Ron Crocker's advice: Create a simple architecture that handles all the known "big rocks" (requirements that are particularly hard to incorporate late in the project).[9] Handle the "small rocks" as they appear during the project.

AGILE

Agile implies being effective and maneuverable. An *agile* process is both light and sufficient. The lightness is a means of staying maneuverable. The sufficiency is a matter of staying in the game.

The question for using agile methodologies is not, "Can an agile methodology be used in this situation?" but "How can we remain agile in this situation?"

It seems the important questions to ask are:

1. How does this technique work?
2. Why does this technique work?
3. How is this technique related to other techniques that I am practicing?
4. What are the necessary preconditions and postconditions to effectively apply this technique in the combative situation? . . .

"As you develop a reasonable repertoire of techniques that you can perform correctly, you will need to expose yourself to as broad a range of practitioners as possible. As you watch others, you need to ask and answer at least three questions:

1. Which other practitioners do I respect and admire?
2. How is what they do different from what I do?
3. How can I change my practice (both mental model and attempts to correspond to it) to incorporate the differences that I think are most important? . . .

Seven Properties of Highly Successful Projects

- *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 takes to get a question to the right person, and the frequency of serendipitous discovery.
- *Personal Safety.* Can we tell our boss we misestimated 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?

TECHNIQUES À DISCRETION

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 help 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 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 their having to ask anyone a question. Ideally it is large, visible to the casual observer, easily kept up to date, and understandable 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 person can 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 both as a specification of what to design and as a practice run at using the call sequence to the function. Also called XXD (see page 275).
- *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 1999). Described in (Patton 2003).

AGILITY BEYOND SW-DEVELOPMENT:

THE DECLARATION OF INTERDEPENDENCE

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 is 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. 323) to see how lean manufacturing already anticipated what we wanted 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 February 1, 2005, writing the six points of the Declaration of Interdependence, 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 by 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."