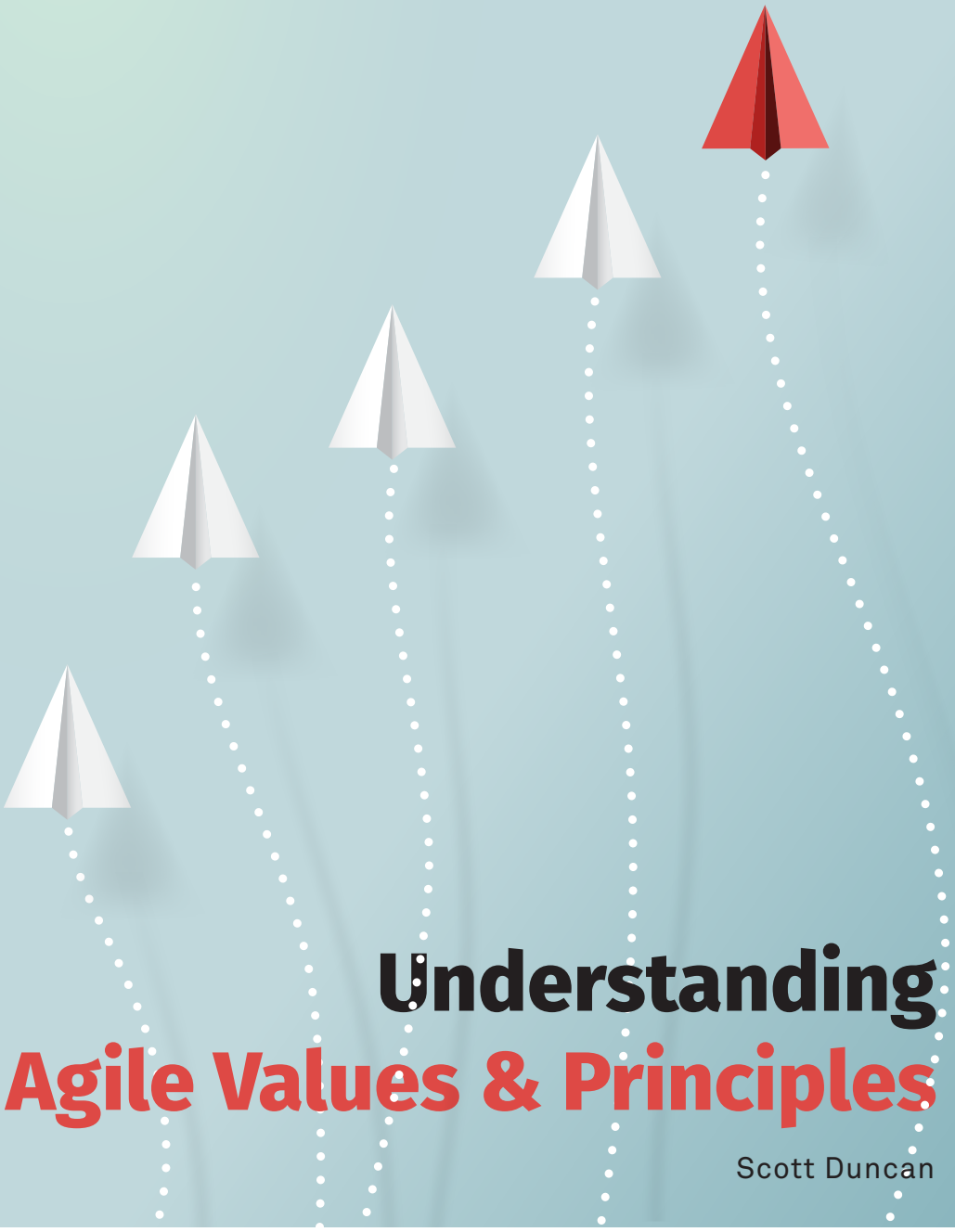
AN EXAMINATION OF THE AGILE MANIFESTO



Understanding Agile Values & Principles

Scott Duncan

InfoQ
ENTERPRISE SOFTWARE DEVELOPMENT SERIES

Understanding Agile Values & Principles

Published by C4Media, publisher of InfoQ.com.

ISBN: 978-0-359-52387-0

Production Editor: Ana Ciobotaru
Copy Editor: Lawrence Nyveen
Design: Dragos Balasoiu

Contents

Foreword

Agile was founded on a set of values and principles in a short document known as the Agile Manifesto. To implement these agile values and principles, teams follow certain practices, such as writing user stories or doing pair programming.

In the early days of agile, I remember participating in many raging debates about whether principles or practices should come first. That is, if a team followed a set of practices long enough and well enough, would team members come to learn the principles underlying those practices? Or was it necessary for teams to start with a solid understanding of the principles of agile before implementing its practices?

These debates often raged at the early agile conferences and Scrum gatherings. I met Scott at one of these conferences: the Agile Development Conference in Salt Lake City in 2004.

In the ensuing 15 years, I have been consistently impressed with Scott's mastery of agile. When he weighs in on the question of principles versus practices, as he does in this book, we all benefit.

In *Understanding Agile Values and Principles*, Scott analyzes each of the 12 principles and four value statements of the Agile Manifesto. Along the way he explains the intent and importance of each. But he also describes how teams may struggle to put a principle or value into practice. And he offers practical, ready-to-use advice on living the principles and values of the Agile Manifesto.

The Agile Manifesto fits on one page. But, as short and concise as it is, much is left open to debate and interpretation. Those debates will never — and probably should never — stop. It is through discussing what it means to be agile that we uncover better ways of developing software. Scott's thoughts and advice in this book represent a significant step in forwarding that discussion.

Mike Cohn
Co-founder of the Scrum Alliance and Agile Alliance
Author of User Stories Applied, Agile Estimating and Planning,
and Succeeding with Agile
www.mountaingoatsoftware.com
www.agilementors.com/

Other Comments on the Book

"Whenever I engage with people who are looking to practice agile, my first goal is to ensure they understand the core agile values and principles. This is the foundational "why" that underlies the practices in a framework like Scrum. Without first establishing this core, practitioners will not be well equipped to inspect and adapt their approaches to practicing Scrum. Scott has provided important commentary on and elaboration of the values and principles in the Agile Manifesto. This is a good read whether you are an agile novice or an experienced practitioner."

- **Ken Rubin** author of *Essential Scrum: A Practical Guide to the Most Popular Agile Process* and creator of the Visual AGILExicon® (https://innolution.com/)

"Scott Duncan has written an interesting and valuable book. He works through all the values and principles of the Agile Manifesto, considering each phrase and its implications for an individual or organization interested in taking an Agile approach to their work. This is in sharp contrast to almost all the other Agile advice you'll find: Scott has no particular framework or product, no particular axe to grind. The book is essentially a series of short meditations on each of the ideas in the Manifesto. He leads us through a clear and comprehensive look at the Manifesto and what it really means to those of us who try to apply it in our work and lives. Highly recommended!"

- **Ron Jeffries**, co-author of the Agile Manifesto and author of *The Nature of Software Development: Keep It Simple, Make It Valuable, Build It Piece by Piece* (https://ronjeffries.com/)

Preface

Many years ago I began what I planned to be a much larger book with this same title. I felt that many organizations I had been working with had started their Agile journey without a good (or any) coverage of the Agile Manifesto's Values and Principles. The book was too grand an effort and I put it aside. However, over the years, my concern for the lack of organizational knowledge of the Agile Values and Principles has not lessened. If anything, it has grown. I decided about a year ago to put up some essays on Medium.com and that turned out to be the precursor to this book.

People talk of "being" agile rather than just "doing" agile and, someday, that longer book may get done and be about the latter. This book, however, tries to be about the former with enough examples of "doing" to illustrate how one might behave if they were "being" agile.

I would not have gotten to this point, though, without many people who have influenced my thinking about Agile software development. Since I first read *eXtreme Programming Explained* in 2002, there have been enough people that I doubt I could never list all of them. And before that for about 25 years, even more people influenced how I viewed traditional software development/engineering topics.

Acknowledgements

I would like to thank a few people specifically for reading the draft of this book and for their comments about it:

First, knowing him for about 15 years, and since he (along with Ken Schwaber) was my CSM trainer, I want to thank Mike Cohn. I have had the pleasure of being at conferences with Mike, taking classes from him, working on the Scrum Alliance Board of Directors with him, and participating in his Agile Mentors Community. I also had the opportunity to read and comment upon the draft of his *Succeeding with Agile* book.

Second, knowing him almost as long as Mike, I want to thank Ron Jeffries for looking through the draft of this book. Over the years I have enjoyed presentations he has given on Agile development practices and his early writing on eXtreme Programming practices. Before I learned about Scrum, Ron's (and Kent Beck's) work on XP was my introduction to Agile practice once I encountered the Agile Manifesto.

Third, I want to thank Ken Rubin for the opportunity to read an early draft of his book *Essential Agile* and, now, for his suggestions for improvements to this book (which I'll do a better job with on the next book) and his overall perspective on it.

There are many others whose work and insights have helped me along my personal Agile journey:

Jurgen Appelo, Lyssa Adkins, and Barry Boehm & Richard Turner were others who allowed me an early look at their books;

Kent Beck, of course, through his *eXtreme Programming Explained*, pushed me in the direction that has led me to this point in my career;

Alistair Cockburn has always added to my appreciation for thinking in an Agile way every time I have read what he has written or heard him speak;

Then there have been people such as Mary & Tom Poppendieck, Robert Galen, Roman Pichler, Brian Marick, David Anderson, Ken Schwaber, Jeff Sutherland, Scott Ambler, and a host of people whose books, conference talks, blogs, and videos have offered me guidance over the years.

And, before I encountered the Agile Manifesto, during my 'phased sequential" days, especially 14 years in the Bell environment, there was the work of Barry Boehm, Tom DeMarco, David Parnas, Gerry Weinberg, and Fred Brooks to mention just a few.

Finally, my thanks to:

Ben Linders who showed the initial interest in getting this book published and has offered much help along the way;

Lawrence Nyveen for his editorial review;

Ana Ciobotaru and Dragos Balasoiu for their work on the cover of this book and other publishing details.

PART 1

An Overview

I have been coaching and training teams in agile approaches to software development for over 13 years. During this time, many frameworks and practices have been proposed and used. In my experience, one thing seems clear: as framework-specific training has increased, many of those trained seem to have limited familiarity with the Agile Manifesto's values and principles (http://agilemanifesto.org). As a result, people tend to return to what they had been doing previously when practices advocated during training have been difficult or not possible to implement. This often seems to occur without reference to the values and principles to consider how some alternative might be selected that more closely adheres to them. I believe it is the limited understanding of the values and principles that excludes them from people's thinking when choosing alternative practices.

It is for this reason that I offer my thoughts on what an understanding of the values and principles could mean to an organization. Such understanding will make it easier to pursue an agile approach successfully by using the values and principles as a reference point when considering practice alternatives. As many have pointed out, agile ideas are quite simple but not necessarily easy to implement. While based on ideas with a lineage that goes back at least to the middle of the last century, they form a set of reinforcing ideas intended to be used together, not piecemeal.

I will take each value and principle and explore what I believe it can mean, suggesting along the way a variety of possible practices and considerations that could help implement each one.

To begin, this overview will address the Agile Manifesto as a whole. The next chapter will address fundamental considerations basic to an effective group work situation, but especially important for being agile: communication, collaboration, commitment, continuous improvement, trust, and excellence. After that, each value and principle will have a chapter dedicated to addressing it.

The Agile Manifesto

What's covered below are three components of the manifesto that bear on how one understands the value statements: the initial two statements, the final statement, and the preposition. Early reactions to the manifesto often sounded as if they misunderstood or even ignored the preposition and final statement.

The initial statements

The Agile Manifesto's first paragraph is "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:". All the people at the meeting that created the manifesto had direct experience developing software. No one was a pure academic or researcher. What one finds in the manifesto, therefore, is based on practical, not theoretical, experience. Those in attendance stated they felt this experience had led them to believe four key things: the values.

The final statement

I remember reading articles back in the early 2000s in which people suggested or outright said that the manifesto authors advocated no process or tools, no documentation, no contracts, and no planning. Such overt statements ended some time ago, but you may hear this same sense expressed in statements like, "You can't possibly do <some aspect of development> without <some particular processes, tools, documentation, contracts, or plans>." Of course, the manifesto doesn't say you should do without these things. The final statement says "That is, while there is value in the items on the right, we value the items on the left more." The manifesto acknowledges value in processes, tools, documentation, contracts, and plans while saying that the authors believe there is greater importance in individuals and interactions, working software, customer collaboration, and responding to change. Understanding matters of degree are therefore important.

The preposition

Note that the word forming the connective link in each value statement is "over." It is not "instead of", "to the exclusion of", "without", or some other exclusionary form. The word "over" implies a precedence, not an exclusionary relationship, within each value statement.

If an organization has difficulty:

- with individuals interacting effectively, then more, or more rigorous adherence to, process and tools probably won't help;

- producing working software regularly, then more, or more detailed, documentation probably won't help;
- collaborating with customers positively, then more rigorous contracts probably won't help; and
- responding to change directly, then more, or more rigorous adherence to, plans probably won't help.
- Looked at another way:
- Processes and tools should help individuals interact more effectively.
- Documentation should help achieve working software more regularly.
- Contracts should be written to help collaborate with customers more positively.
- Plans and planning should help the ability to respond to change more directly.

Stated this way, if the former cannot help achieve the latter, then the former become impediments to success.

Since the manifesto

Over the years, people have proposed updating the Agile Manifesto to address things they felt the original authors missed or were now out of date. I also heard someone once claim that the manifesto was "content-free", by which he meant that it was easy to agree with without committing to anything specific. Who wouldn't want effective interaction between people, software that works, collaboration with customers, and being able to respond to change? But as I have heard (and read) at least one author of the manifesto say, it was a statement of what those 17 people in the room during those four days in 2001 believed at that time. As such, the Agile Manifesto (or any manifesto) isn't something you update.

New ideas will appear and be pursued and can offer useful guidance to what to consider in becoming and pursuing agile. In particular, two interesting ways to think about what it means to be and to pursue agile are Alistair Cockburn's Heart of Agile and Joshua Kerievsky's Modern Agile. I will come back to these two at the end of the book as well as mention comments made by Kent Beck in 2010.

Not just for software

One final point before ending this introduction is that the manifesto's values and principles easily apply to any activity where people need to work in some way to produce some end product or service. If one replaces the word "software" with "product or service", I believe the values and principles retain their validity.

At an agile conference held several years after the manifesto had achieved broader visibility, I heard someone bemoan the attempt to apply agile outside the realm of software. They seemed to feel that agile, coming from the field of software and having many early practices rooted in software, would be diluted by efforts to apply it in other domains. I believe dilution of agile ideas occurs not from applying them in other domains but from people not using them as a touchstone for becoming agile regardless of frameworks and practices.

The next chapter

Hopefully, this overview will interest you in reading further. The next chapter will address the fundamental considerations of communication, collaboration, commitment, continuous improvement, trust, and excellence noted above.

PART 2

Fundamental
Considerations

In Chapter 1, I mentioned several ideas which I called "fundamental." It seems hard to expect an organization (or a group or an individual) to become (more) agile without them. As noted in Part 1, all these considerations are important for any work situation, but seem essential for being agile. Being agile, as many have said, is more important than "doing" agile. That is, understanding how to apply the values and principles is more important than specific practices. Practices, of course, must exist, and there are many ways to "do" agile. But perhaps, there is a narrower set of choices in "being" agile.

Communication

Every value in the manifesto is affected by communication, though the forms differ:

- It seems impossible to expect positive interaction between individuals without effective communication. This requires an openness and willingness to bring visibility to the relationship. It means having to risk being wrong in front of people if ideas and information are going to flow freely back and forth. It means asking more than telling and being inclusive ("we") rather than accusative ("you").
- Achieving working software certainly requires effective communication between people, but also between the software itself and those people. This requires design, development, and testing practices that expose what is happening with the software at any stage in its creation. Being able to know the stability of your system at any moment encourages experimentation, builds trust, clarifies quality, and makes change less risky.
- As positive collaboration with customers (or any stakeholders) is just a specific instance of individuals interacting, it would seem to go without saying that this requires effective communication. But it is a two-way obligation. Customers must realize that, to get the benefit from an agile approach, they must involve themselves in the approach at various points, providing feedback and clarification, and even guidance when needed.
- Finally, it is hard to imagine responding well to change without effective communication between all those who need to know about, have valuable insight into, and will be affected by such change. Discussions of scope flexibility and the impacts of change should include customers and development organizations.

Most development trouble occurs because communication is not effective.

Collaboration

Collaboration is certainly an aspect of effective communication. Many people working in teams believe they are collaborating when they are probably just cooperating. There is nothing wrong with cooperation, but collaboration goes beyond cooperating. One can cooperate without collaborating.

To "co-operate" literally means to operate together, to function in conjunction with, which can occur with limited interaction. Individuals can have their own work assignments, work to get them done, and be willing to help out others when they can. But they can essentially focus on what they feel they must accomplish individually, expecting others to do the same. People can cooperate without assuming any responsibility for any work other than their own.

To "col-laborate" literally means to work together. It involves deliberately sharing work, and responsibility for the work, involved in getting something done. This means structuring tasks so people work closely with one another to accomplish planning, estimation, commitment, development, and testing. It also means communication must be more direct and frequent, which cooperation doesn't necessarily demand.

Collaboration's connection to individual interactions and working with customers seems clear. Achieving working software and responding to change are slightly more indirect, but not much. Achieving both occurs more effectively when collaboration, not just cooperation, exists. Achieving working software can be much harder (if not confounded) when there is a hand-off mentality between skill silos.

Likewise, response to change may be less effective without a collaborative stance. People, without intending any negative impact on others, can sub-optimize the change in a way that improves (or maintains) their own situation while sometimes making other people's situations worse.

For example, you've probably been in a situation where departments in a company have been told to cut their budget by some percentage for the coming year. In one instance I experienced, IT support rightly pointed out that a lot of money was being spent supporting many versions of desktop machines used both for actual development and for basic administrative and office activities. Clerical support staff were using powerful engineering workstations, Macs, and Windows PCs to simply maintain management file systems, budget records, presentations, and memos.

IT Support noted that, unless required for development, one standard desktop environment should be used. Windows 3.4 and relatively inexpensive PCs became the approved environment when development was not a consideration. This also reduced costs since support personnel for such an environment were more plentiful and less expensive than for the other environments. After converting many people, including all management support staff, a serious, undocumented bug was exposed that took a couple weeks to track down (by technical staff other than IT Support) and another (the same technical staff) to correct the consequences of once identified. Hundreds were affected, including management whose files had temporarily "disappeared," the cost of which was never formally recorded, but IT Support could show their budget was reduced by the required percentage.

Commitment

Communication and collaboration are more easily improved when people see commitment from one another to achieve the work before them. In turn, that commitment is more easily accepted when effective communication and collaboration exist. It's a bit of a circle of course, though not a vicious one. One requirement, however, is that the people doing the work take on and make their own commitments. These should not be imposed on them. More will be said about this when self-organizing teams are addressed in a later chapter. However, people should, for example:

- make their own estimates of the work to be done;
- define their own work tasks and who will/can do them;
- agree on shared working agreements, including quality standards and practices; and
- manage and track their own progress toward completing the work.

People doing these things together, closely communicating, will more easily feel that the commitments are truly theirs, motivating them to achieve, even exceed, them. People want the opportunity to take pride in their work and commit to it. (Deming pointed this out in his work with the Japanese after WWII and documented it in 1982 in his book *Quality, Productivity, and Competitive Position*, later renamed *Out of the Crisis* in 1986.)

Continuous improvement

Perhaps nothing is more important in being agile than the pursuit of continuous improvement. Indeed, the basic agile lifecycle instantiates this through the retrospective held at the completion of every iteration of work. Unfortunately, it is sometimes a skipped step in that lifecycle, trailing planning and daily meetings in frequency according to VersionOne surveys.

A variety of things may contribute to this, but a key one is that while people come up with ideas for improvement, nothing ever changes or happens to those ideas. Unfortunately, after making lists of possible changes, people do not treat the ideas as serious work items. The ideas do not get prioritized, estimated, or committed to actual work effort in the next iteration. Thus, the next retrospective occurs, and the same ideas are raised again.

Another reason is that people don't feel there is time to conduct the retrospective given pressure to get going on the next set of customer requirements. Clearly, the time and effort to carry out improvement should show benefit but it cannot if people aren't given, or don't feel they can afford, the time to try.

Finally, I've heard people say, "Everything went fine this iteration so there is nothing to fix." The retrospective is about improvement, not just "fixing" things. Unless a team never generates any defects, is always completely accurate in its estimates, and is completely productive, there is always something that can be improved. It doesn't have to be a huge change. It's about continuous improvement as a habit (i.e., what in lean terms is called *kaizen*). All the values and principles of the Agile Manifesto can probably be more effectively implemented in one way or another in any organization.

Trust

It's has been said that trust is earned in pennies and lost in dollars, i.e., hard to come by but easy to lose. Probably nothing destroys a group's effectiveness more than lack of trust within the group or by those around it. Unfortunately, typical project management practices are often based on or conducted in some aspect of a lack of trust. But without trust, open communication and collaboration just won't happen. People will not pursue new ideas or attempt new ways of working if they do not feel they can expect support from others when they do.

Of course, it has also been said that "In God we trust; everyone else brings data," "Trust in God but tie up your camel," and "Trust, but confirm." Trust does not exist between people automatically. People may presume trust until some problem surfaces, then begin finger-pointing among one another. It is difficult to truly trust someone with whom you have had no positive working experience, that is, have no "data" upon which that presumption exists. Effective, deliberate communication, collaboration and commitment from the very outset of people working together provide that "data," contributing strongly to creating such trust.

Excellence

Finally, there is the desire to achieve excellence. We cannot assume excellence from everyone, but we can expect them to pursue it, collaboratively, using continuous improvement. True excellence may be (or at least seem to be) a distant target, but pursuit of it is certainly reasonable. Often the problem is deciding what excellence means in a group's work and how to pursue it. Established group working agreements and quality practices along with use of the retrospective should make such decisions easier, building trust at the same time.

PART 3

Individuals and Interactions

This chapter begins a set of four chapters, each focusing on one of the Agile Manifesto's value statements.

Individuals and interactions

I think anyone who has worked with others—be it at work, on a service project, for a local community group, or even a family effort—would agree that the interactions of the individuals within the group had the biggest impact on the result. If there is effective communication, collaboration, and trust among people in such situations, they are likely able to overcome many limitations in process, environment, and tools. Those who supervise or manage such efforts should make it a priority to see that such communication, collaboration, and trust exists and that they, as supervisors or managers, protect and encourage that existence.

But a major challenge to doing this in a work environment is that so many teams are distributed. Indeed, I have not worked as a coach or trainer for any company in over a decade that did not have distributed teams. The sixth principle says that "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation." However, co-located teams are frequently not the default occurrence. While I'll address distributed teams throughout future chapters, this value is certainly an appropriate place to start, given how distribution creates challenges for communication, collaboration, and trust.

Most people think of distribution as a matter of distance (and, hence, time). Alistair Cockburn, in Chapter 3 of his book, *Agile Software Development*, notes that what matters is "is the amount of energy it takes" to pass ideas between people. Direct, face-to-face communication between people can be discouraged at amazingly short distances, e.g., perhaps 30 meters. If the energy and time it takes to stop what you are doing, get up, and go to another person is perceived as too great, people will simply not do so. They will communicate as if they were miles, even continents apart, using various forms of text-based methods. (Interestingly, people who recognize distribution across large distances and time use technology that allows them to see one another (e.g., Skype, Google Hangout, Webex, etc.), but might never think to do this more locally when disinclined to get up and go to visit another person.)

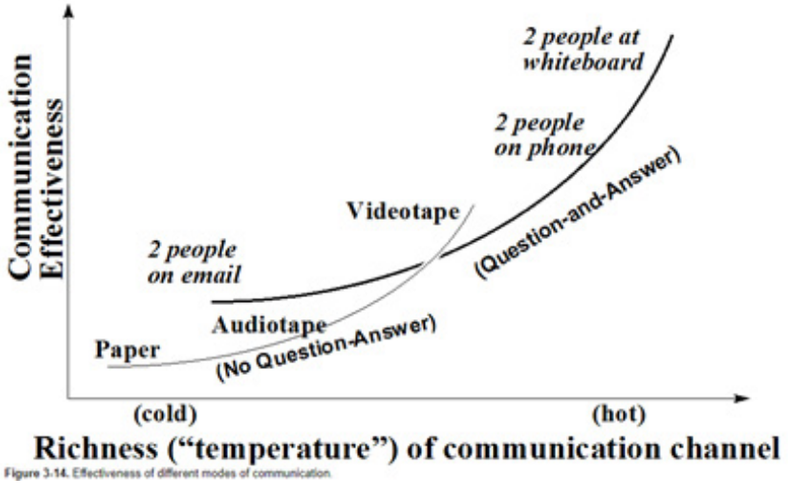


Figure 3-14. Effectiveness of different modes of communication

Cockburn notes that this figure from his book captures "the findings of researchers, such as McCarthy and Monk (1994)" in *Measuring the Quality of Computer-Mediated Communication*. There's more to say about it in Chapter 12, when the sixth principle is discussed. For now, a couple things can be pointed out.

The vertical axis indicates how effective the mode is in communicating information while the horizontal axis indicates the impact of the communication. Together, they show how little time and energy the modes take to communicate information between people as we go right and up along the axes. Note where purely text-based documentation sits ("Paper" on the diagram) compared to face to face ("2 people at whiteboard").

Processes and tools

Many tools, of course, allow people to open them up and look at the status of team work whenever they want to know something or when the tool alerts them that something has happened.

Committing to interaction

What's important is that there is the commitment to interaction, to communication, to a whole team atmosphere. Rather than finding it too hard to do this, falling back on exchanging documents, and splitting up work so people do not have to interact, look for ways people can share the work — actually collaborate not just cooperate. According to one of VersionOne's annual surveys, only about 20% of respondents say they looked for agile to help them better manage distributed teams, while 61% of them said that, in fact, that turned out to be a benefit of adopting agile. I believe this is because of a commitment to making communication work among team members and between teams. It's a challenge whether you are distributed or co-located but being agile expects this of us.

An experiment

Here's a thought experiment (or perhaps an actual exercise you can do with a group of people). Think of some very positive experience you have had working with a group of people. Write down the characteristics of that experience that make it so positive in your memory. Now categorize them as:

- people — interactions between you and individuals in the group;
- culture — the overall group/organizational assumptions, agreements, attitudes and expectations;
- process — the way(s) the group organized the work you did and carried it out;
- environment — the actual physical environment in which you worked; or
- tools — the tools (manual or electronic) you used to get the work done.

Now look at the pattern created by grouping your memories of that experience. Where do those characteristics fall in the groupings? (If you do this with a whole group, write the ideas on notecards or stickies and put them up on a board/wall with the headings listed so everyone can see the total set of groupings.) What does this tell you about why the experience was memorable?

PART 4

Working Software

Who wants software that doesn't work? But what does it mean that it works? It shouldn't have any defects, right? It should have been fully tested, correct? The agile approach to quality expects that at the end of every iteration of work, the team feels the software is of production quality. That is, if it could be put into production, the team would not worry about it failing to meet customer expectations. They are confident in the quality of what they have produced and not just based on being defect-free.

To be fully useful in production, all associated user guides, installation procedures, help-system entries, etc. should also be completed. Without them, it may not be possible to use the software even if there were no defects. The software wouldn't "work" from the end-users' perspective if they could not use it regardless of how defect-free it might be. From an agile perspective, the "software" includes that necessary user documentation. Such of documentation is considered part of what must "work," as well as the code itself.

But sticking just to the code, what must work? Let's look at a simple example of what a customer might request:

"As a banking customer, I want a list of all my accounts and balances so I can manage my finances."

Seems like a simple enough request, but let's add some acceptance criteria. These are often referred to as conditions of satisfaction (CoS).

- I want the list to be a pop-up list I can request from anywhere in the system.
- If the list exceeds 20 items, I want it to scroll.
- I want the list sorted by account type or balance.
- I want to be able to select to sort by balance then by account if I want.
- I want to be able to see due dates and required payments on debt accounts (e.g., credit cards, loans).
- I want to sort by either due date or required payment if I have those showing.
- I want to add other fields to each item such as <imagine a few others>.
- I want to filter the list by account type, balance, due date, or required payment to shorten a long list.
- I'd like to be able to search the list.

Does the software "work" if any of the Conditions of Satisfaction don't work or exist? If we implement the basic request and verify that works,

then implement each CoS, verifying they each work as we add them, does the software "work" at each point along the way? From a code quality perspective, it could be said that it does. What if a customer will take a useful subset of all the CoS because that subset will "work" for them in the near-term. They'd prefer to get a useful increment sooner rather than later. Everything they asked for initially isn't there, but enough is to make it useful. Later, they'll get more of the things they requested.

To start, then, the team could (depending on the length of the iteration) provide the first three capabilities, then add the next two in the next iteration, then the sixth one in the following iteration, etc. At the end of each iteration, new functionality could be put into production as the customer, incrementally, gets more useful software.

Discussing what makes the software minimally viable in this way can lead to customers getting something useful rather than waiting for everything. This is part of the agile idea of ordering the work based on customer value and working to deliver high-value results soon and frequently during a release/project cycle. It puts the decision about what constitutes a viable release in the hands of the customer/business and not in the hands of the development team. The team just keeps adding the next-highest-valued functionality and the customer says when it's useful enough to be a release.

Communicating with the software

The importance of effective communication between customer and development is clear. What about effective communication with the software itself? People can communicate with the software simply by using it in some way. The software responds by performing as expected, or not. This begins with how the team verifies and validates the software through forms of inspection and testing. Then the team's product owner or customer can try the software to see how it works. Done frequently enough, this communication will reduce risk by uncovering defects or new functionality requests early, before they become too expensive, increasing customer satisfaction with the results.

The company Menlo Innovations likes to have a customer representative present with their developers throughout the process of creating the software for that customer. At the end of each week, when the team shows

what they have accomplished, they like the customer representative to demo the working software. That person was there all week discussing the functionality with the developers, seeing it developed, and helping verify it. If they cannot show how it works, something must have gone wrong in the relationship during the week and that needs to be addressed.

Implementing small pieces of functionality, verifying they are correct, and validating that they satisfy the customer's expectations is the essence of agile development. When starting to work in an agile fashion, this process can be hard because people are used to long(er) development cycles and think in terms of large(r) pieces of functionality. Getting the customer (or business) to think in smaller pieces of functionality and smaller release cycles will take time, but is essential to get working software into production sooner rather than wait a long time to get everything, with all the details completed.

I observed a team that worked very hard to break functionality down into pieces that could be completed (including user-relevant documentation) in just a few days rather than take a few weeks to get all of it done. This increased collaboration between developers, quality analysts and tech writers, making collaboration easier. It also resulted in the team spending little time estimating the work. Everything was such a small effort that they just called everything 2 points (using the Fibonacci story point scale). They spent their time breaking stories down — which was the important thing to be able to develop, test, and deliver frequently — eliminating estimation effort almost completely.

Comprehensive Documentation

User manuals, installation guides, help systems, etc. are considered part of the "software" from an Agile perspective. The "comprehensive" documentation this value mentions is internal, system forms of documentation which are often not used once a project/release is completed. The argument can certainly be made that detailed documentation is important for historical reasons, but it may not be the best way to communicate among people actively developing the current system. Such "comprehensive" documentation may contribute to the next project/release, but the immediate goal is to deliver the current requirements as quickly and inexpensively as a high level of quality and reliability allows.

Many years ago, I was at a conference and Barry Boehm spoke about the documentation "memorial libraries" he had seen in many companies, filled with shelf upon shelf of binders where the only people who ever looked at it were auditors when they needed to verify what happened when they weren't there to see it for themselves.

If you are going to generate any volume of documentation, you must accept the total cost of ownership (TCO) make the investment to keep that documentation current. As a developer, I often found that I could not trust documentation because it did not match the code. I spent a week sometimes trying to debug a problem or add functionality believing in text documentation, both outside and within the code.

I once worked where, instead of updating the original design document to match the new work, people created addenda documents which only addressed the changes being made. After a few years, if you wanted to use/rely upon the documentation, you ended up having to read 6–8 documents: the original and various addenda. Eventually, people just didn't bother which also led to not bothering to document at all after a while. The old development-centric view used to be that you needed to read the code to know what the system did. That was the only reliable "documentation."

The Agile approach does not mean to reject all documentation, of course, just to apply some simple ideas so the documentation has "sustainable" value. By "sustainable," I mean that the documentation is considered so valuable for developing high quality, valuable software, that people willingly create it and keep it updated. Agile development prefers executable work products (e.g., tests) to static ones (e.g., traditional text documents). Indeed, a strong automated test suite is, in effect, a set of executable requirements.

Going from user stories to working (i.e. tested) software using strong programming practices and meaningful naming conventions will mean you need less of the traditional documentation. There is risk when code is written and not easily and quickly verified to be correct. The interval between writing code and verifying code is that risk. In traditional development, that time can be weeks, even months. The goal is to shorten that time as much as possible. (More will be said about approaches to quality and testing in future chapters.)

I once worked as a coach for a project with three teams (growing to five over the 10 months of the project). At the outset, the teams could get one

build every four weeks (their iteration length). Such builds were done by a separate group, in a separate state, supporting all the company's projects. You can imagine what that meant to team productivity if they were to try to honor completed working software as their goal each iteration. It took several months for the management of that project to get the teams to a build a week, then a build each day, then, not too long after that, two builds a day: one ready each morning for teams on the East Coast and one done at the end of their day for the offshore members of the teams to work on. The ability (and confidence) of the teams to be more productive substantially increased at each one of these junctures. Their velocities went up 10%-15% each time just based on this.

The key thing to remember is that communication, not documentation, should be the goal. Since one size does not fit all needs, each release/project should evaluate what needs to be communicated, to whom, and what the best way to do that will be. Accepting "the way it's always been done" is likely not going to help you deliver working software frequently that meets customer needs. It may, in fact, impede it.

PART 5

Customer Collaboration

As mentioned in Chapter 2, this value can be looked upon as just a form of individuals interacting. But I think it is a different situation than how people in the development organization interact and what the content of that interaction would be. Collaboration with customers is focused on the "what" side of the work rather than the "how." That is, it focuses on what the customer wants as functionality and what they will need to see as evidence of the functionality being satisfactory to them.

Now, customers may get into the "how" quite often as they attempt to define the "what". Indeed, with traditional requirements documents, getting into details of both "what" and "how" is typical. Customers new to an agile approach could be expected to combine the two. They may consider keeping them separated as restricting their ability to define their requirements. The agile approach recommends not diving deeply into "how" too soon for a few reasons:

- It may mean spending a lot of time on something that ends up not actually getting developed because the customer changes what they want later in development.
- Even if it will get developed, people will know more about what is needed closer to the time they must make that decision. (The agile approach recommends waiting until "the last responsible moment" before a commitment is made so you know as much as possible when you must make that decision.)
- When the time to implement the functionality occurs, you may find you have unnecessarily constrained your decision making. (One can cram 10 pounds of potatoes into a five-pound bag by mashing them early on, for example, but you can't expect to get french fries later.)

All of these are also good reasons for not getting overly detailed about the "what" too soon.

It's important that the customer sees the value in frequent, open communication about what they want and what development can show to be working as a way to reduce risk and increase satisfaction with the results.

When I have seen organizations forced to work with customers (internal or external) who will not accept this, there has been noticeable stress on both sides. More than once, I have heard the business side of an organization say they don't have time for frequent meetings and that they just "need it all done by the deadline" so don't want to take time for prioritization. It doesn't always work, but my basic discussion with them goes somewhat like this:

"Do you always get everything you want by the project deadlines?"

They look at me as if I am a bit dumb and say, "Almost never."

"But you do always get the most important things by the deadline, right?"

Now a look as if I am truly dumb, "No. Frequently I don't."

"So that's why we ask for prioritization and frequent feedback, so that if there are problems getting it all done by the deadline, you will get all the highest-valued functionality possible and it will be just the way you want it."

Sometimes the lightbulb goes off.

Contract negotiation

It's unfortunate but a lot of contracts seem to be written not to describe how the vendor and customer will work together for success but to define the penalties for when one or the other fails to meet their responsibilities. This doesn't seem like a way to establish a particularly collaborative relationship between people. Contracts also typically seem to be based on the belief that everything needs to be defined up front. A more incremental approach to defining how the work will be done and how people will work together, collaboratively, to produce the most satisfactory result should be a critical part of any contract.

From the agile perspective, the customer and development organization should explore expectations for customer participation in:

- the iterative process of requirements (e.g., user story) definition and refinement,
- prompt response to questions during the release/project to clarify functionality expectations,
- defining acceptance criteria (conditions of satisfaction) for each requirement,
- each iteration's review of the functionality developed during that iteration,
- offering feedback to confirm their acceptance of what the team has done or their desire for any changes,

- ongoing elaboration of functionality as the team moves from highest to lowest priority in requirements, and
- prioritizing requirements from highest to lowest value so the development team always works on the highest-valued remaining functionality.

If customers are not prepared to do these things or enough of them (which depends on the project), then perhaps a more "protective" contract is smart.

There are a variety of places to look for opinions about structuring agile contracts. Just type "agile development contracts" into your favorite browser. A few possibilities are:

- https://en.wikipedia.org/wiki/Agile_contracts (which speaks mostly of "agile fixed-price" contracts),
- https://www.mountaingoatsoftware.com/articles/writing-contracts-for-agile-development (which takes a user story and conditions of satisfaction approach to documenting contractual expectations),
- https://www.scaledagileframework.com/agile-contracts/ (addresses this framework's managed-investment contract), and
- http://www.agilecontracts.org/ (a 44-page extract from the book *Practices for Scaling Lean & Agile Development* by Craig Larman and Bas Vodde on contracts in an agile setting).

PART 6

Responding to Change

Maybe nothing represents the traditional dictionary definitions of "agile" more than this value. A typical definition reads "able to move quickly and easily" with synonyms being "nimble, lithe, supple, limber, acrobatic, fleet-footed, light-footed, light on one's feet, alert, sharp, acute, shrewd, astute, perceptive, quick-witted". All of these imply an ability to change with minimal effort, but there is something else implied.

Here are some thoughts from an article entitled "Mountaineering Training | Body Awareness: Balance & Agility" which seem appropriate. (Also, the first article I remember reading about the agile approach was by Jim Highsmith comparing software development to climbing.)

Body awareness is the combination of balance and agility that allows you to move comfortably and confidently through difficult and challenging terrain.

Balance in mountaineering allows you to climb through challenging conditions...while keeping your equilibrium and avoiding using excess energy or concentration to stay centered. Simply put, it's being comfortable on your feet even when you're traveling through uncomfortable terrain.

Agility is being able to move quickly and easily—to be nimble and reactive...to react to the unexpected....

...Both balance and agility are motor skills and can be improved over time.

So, it's not just moving quickly, but maintaining stability during that movement. Responding to change in a meaningful, responsible manner is not only about change but about directing that change toward greater stability in the results. From a software perspective, it means change that produces a more satisfactory, useful delivery to the customer than before. It involves applying many faculties and skills, including alertness and perception to decipher the implications of the change, in deciding to change and basing that on the best information available when such a decision is to be made. It is not a license to be ad hoc or to do something when "You should know better." (More on this in Chapter 15 on the ninth principle.)

From an implementation perspective, it means constructing a process for working that allows change to occur with reduced risk and cost. An iterative, incremental approach has long been accepted as the best way to do this. In June 2003, Victor Basili and Craig Larman published "Iterative and Incremental Development: A Brief History" in IEEE Computer, in which they pointed out that this had been accepted 13 years before the waterfall model that Winston Royce defined in his 1970 paper. Basili and Larman have a lot to say about the waterfall model, including how Royce's

son states that Royce felt a waterfall approach "would not work for all but the most straightforward projects" and how the rest of Royce's paper addressed "iterative practices".

Following a plan

There is plenty of agile-related literature about the difficulties and potential wastefulness of long-term, detailed planning when working in a complex situation where change is inevitable and the ability to predict the exact outcome desired is limited.

Consider the typical project plan-creation approach of handing out a detailed requirements document and asking people to estimate, in hours, every task needed to produce the result described. This process of tasking and estimation takes many hours of effort itself. Eventually, all that data is sent to a project manager who feeds it into a project-management tool of some sort and, at some point, a project plan exists. That plan says, for example, "Seven months from now on Wednesday at 3 o'clock, here's what will happen." Who believes that? Who can be sure the person who provided the data to produce that result will really do the task that day? Everyone knows things will change and the project manager spends considerable time each week of the project adjusting that plan.

This all assumes everyone understands what is being described and that the plan describes what the customer wants. But is it even fair to require (or assume) that people, especially the customer, can conceive of everything and how it must work before anything has been designed or shown to work? Then, as well as presuming this foreknowledge, can we also expect that once the considerable effort of doing all this definition and planning is completed (which by some estimates can consume up to 50% of the project schedule) nothing major should be expected to change or, more critically, be learned?

From sources in the history of military planning we have statements attributed to people like von Moltke: "No plan survives contact with the enemy." (He really said, "No operation extends with any certainty beyond the first encounter with the main body of the enemy." The terse version seems a bit easier to grasp at a glance, though.) We also have Eisenhower saying, "In preparing for battle, I have always found that plans are useless…." The ellipses, though, represents the remainder of Eisenhower's quote, which

is "but planning is indispensable" — which brings us to the agile approach to planning which, like development itself, is early, frequent, and iterative.

It is the act of planning with the whole team involved that matters, not the specific plan at any given time. Planning causes us to communicate, to collaborate, to build confidence and trust in how we will work to move in the currently understood direction. Every day, an agile team has the chance to see where everyone is and determine if where they are and where they are headed needs to be adjusted in any way based on new information.

Agile teams contribute to forming a release plan based on the known functionality desired, how much of that the team believes it can deliver each iteration, and how many iterations, therefore, it would take to deliver all the known functionality. But things will change: teams may get better at delivery; the customer may change what they want, having seen some of the early working software; perhaps the business feels it can release a valuable increment of software earlier than planned. And, of course, it may be that not everything originally desired can get done within the budget and schedule. But, using an agile approach, at least what does get done should be the highest-value functionality. (Refer to Chapter 5 for an example of an actual discussion with a product manager on this topic.)

PART 7

Satisfy the Customer

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

This chapter begins 12 chapters, each addressing one of the Principles in order as found at http://agilemanifesto.org/principles.html.

Our highest priority

Being agile means being dedicated, first and foremost, to satisfying the customer. This does not have to mean that "the customer is always right," but it does mean that the customer always deserves to be taken seriously. It also means that trying to satisfy the customer means both customer and development team understand what that means and what both can do to achieve that satisfaction. This brings us back to communication, collaboration, commitment, continuous improvement, trust, and excellence. Achieving this highest priority won't happen satisfactorily for everyone without pursuing these fundamental considerations.

I suppose everyone's heard the phrase (or something like it): "It's hard to remember you're supposed to be draining the swamp when you're constantly fighting the alligators." There are lots of things that can distract from remembering that your main goal is to satisfy the customer. There are other company priorities. There are organizational silo bureaucracies. There are dependencies between teams. There are technology challenges. There are the requirements themselves, which, in too much detail, can actually obscure what's truly needed.

Remember that lean ideas are some of the roots of the agile approach. A key lean idea is the elimination of waste, i.e., the elimination of anything that does not directly contribute to the creation and delivery of customer value. Doing a Value-stream mapping of your process can bring visibility to where waste resides as well as assign quantitative impact them. This is usually computed as time spent on each step of the process plus time spent (waiting) between steps.

You may believe that areas of waste are obvious, but perhaps not to everyone. A lot of the waste in an organization would not appear on a formal process diagram. Honesty in communication is critical for this. You must be able to show what is really happening, not what everybody thinks is happening. This will probably mean you have to talk about the "elephant

in the living room" and the "dead fish under the table": those things everybody ignores as they get their work done despite the "smell."

Early and continuous

Get working software into the customer's hands right away. Keep doing that every iteration and work collaboratively with the customer to get feedback that lets you know they are satisfied with what they see and/or any changes they'd like made.

You want to start early so the customer is immediately engaged with you. There's no going away for weeks or months while you do "infrastructure" work and cannot show any actual customer functionality. You'll lose the customer's commitment to be a part of the process and give you feedback. An agile iteration is expected to deliver customer-recognized value. Don't consider things agile iterations if they do not.

You want to do this continuously to maintain customer engagement and ensure no deviation from what the customer wants, even if they want changes (in fact, especially if they want changes). Doing this requires communication, collaboration, and continuous improvement while reinforcing commitment, trust, and excellence.

In this regard, active customer participation in iteration reviews/demos is crucial. (In a future chapter, I'll say more about this review/demo in some detail.)

Valuable software

Who doesn't want the software to be valuable? But there are a lot of factors that can go into what makes it valuable and what makes some features more valuable than others. Ultimately, value is in the eye of the beholder: the customer. This is what makes collaboration with the customer so important in achieving satisfaction. What makes a piece of functionality valuable to them? What will they expect to achieve by having such functionality? How will things work better for them with it than without it? Understanding their value proposition for a piece of functionality can

convey their sense of urgency/priority as well as better inform the development team how best to implement that functionality.

Despite this, sometimes other factors may affect priority more than pure customer value. For example, I once worked with a team whose products were used in nuclear power plants (among other places). They had to meet Nuclear Regulatory Commission requirements, which asked for substantially more documentation than other teams in the same company. That documentation also had to be created and approved before work was supposed to start on the functionality it described. Though the functionality was of the highest value to the customer, there was a priority for that team to create and approve the documentation ahead of the functionality. Sometimes, the priority of various forms of compliance regulations may override other, more directly functional value.

Even though you don't want to delay working on things of value to the customer, technology and infrastructure constraints may require that some things must exist before one can show fully working software that satisfies the customer. For example, if you need to make database changes to support customer functionality, you could, while such internal work is going on, develop the customer-visible functionality and get feedback sooner than later. You would simulate the database and show the customer what does work.

A team I worked with had to do just that. A customer requirement specified entering some (new form of) data and having the database respond with some corresponding data that would be displayed. The database work had to be done by a group outside the team and they weren't going to get to it for a few months. But the customer priority was to see the functionality as the next highest-priority item. The team decided to break that functionality into three separate stories: one to get the data from the customer and send it to the database, another to get the data back from the database and display it as the customer wished, and the third to integrate the other two parts when the database group got around to making the needed updates.

Now, the team could have pushed the whole thing out, but they felt that would have been forcing the customer to deal too much with their organizational issues. So, they completed the first two stories, simulated the database back end, and showed this to the customer at the next iteration review, making sure the customer understood it would be two or three months before the back-end work could occur. But the customer got to see what they needed and had some suggestions for small changes which

the team made the next iteration. Two iterations later, the database work was done and the "integration story" was accomplished successfully. The fully live functionality was then demonstrated.

It was not the purest way to work, but it upheld customer priority for what they wanted to see while not misleading the customer about what was and was not working. They, and the customer, felt this was the most valuable approach possible.

PART 8

Welcome Change

"Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."

Customer's competitive advantage

It's appropriate that this is the second principle listed. Part of satisfying customers involves doing things that offer some advantage to their business. Building the right system (i.e., what the requirements state) is the basic way to accomplish this, but what's "right" at any moment can change and that often means having to change the system in some fashion. It's important to remember how changing a system (including implementing the stated requirements) provides value to the customer. Taking requests for change seriously and not thwarting achieving better customer advantage is the point of this principle.

Welcome changing requirements

Probably no developer exactly welcomes requirements that change on them. Some discussion with a couple of the people who helped draft the Manifesto suggested that the spirit of this principle meant that the development team should not create unnecessary barriers to change by simply resisting such requests. Yes, they just spent a lot of time and budget to try to make the system as correct as possible based on prior customer feedback, but the customer wants a change. If the customer understands and accepts the consequences of such a change, the development team should work to make that change possible.

Of course, the consequences could be that such a change will:

- extend the planned deadline (if the customer still wants everything else they already expected to get within the original deadline),
- increase the budget (also if they still want everything else),
- push some previously expected functionality out beyond the current deadline and/or budget (to stay within the deadline and budget), and
- affect architecture/design so that it limits future options (which is less easily explained because of the more technical consequence it represents).

However, if the customer does understand and accepts such impact(s), the development team should make the requested changes. (It has been mentioned to me that this is no different than normal projects and the impact of changes. This is true, except that one option is not supported by an agile approach as reasonable and that is to accept all changes without impact to the cost, schedule, or other already defined scope.)

Ability to welcome change, however, can depend a great deal on the excellence in design of the existing system. If people cannot be confident about system changes because the code is not well-designed and the tests for the existing system are inadequate or expensive (and slow) to run, developers may be resistant to (or at least move more slowly in) making changes. If you don't feel you can go into the system, understand what it now does, and make changes without fear of causing new or uncovering existing defects that cannot easily be found, you will be more concerned about making changes. At the very least, you may move far more slowly because of that concern.

Change, of course, is easier when the system is easier to change. A tautology, to be sure, but good technical practices will make changes easier. A couple quick examples of short development episodes and robust testing might be helpful at this point, though.

One company I coached at was able to build and test their 40-million-line system every night because they had invested significantly in automated testing. (Indeed, I have not seen any large development effort get the productivity gains they hope to see from agile development without robust automated testing.) One team at the core of the system's function had 20,000+ tests they ran nightly so they could be confident every morning that their work should not impede other parts of the system.

Another company had a smaller, but no less technically complex, system of 1.5 million lines. They built and tested every two hours. If a developer checked in any code and associated tests (which they had to include) within any two-hour period, at every two-hour mark, the system automatically built and ran all tests, prior and new. If that automated run did not happen for four hours, management would come around and ask why. They expected small increments of new functionality to be put into the system that frequently and proven to leave a stable system state. If it did not, pulling out the troublesome code was easy for them. (I am reminded of the character in the movie *Kelly's Heroes,* who, describing how fast his tanks can go in reverse, says "We want to know we can get out of trouble even faster than we got into it.")

Both organizations, each morning or two-hour cycle, knew the stability of the system. You can make changes far more confidently if that sort of visibility is possible. The longer the time between code creation and testing, the larger the risk and the more slowly development may occur.

Late in development

If developers don't like change, they certainly won't like it late in a release/project. A well-designed, easily tested system will mean that even late changes (assuming understanding and acceptance of consequences) should not cause great concern. The iterative nature of agile development allows for considerable opportunity to review the state of the system and make changes earlier rather than later through the opportunity to see and provide feedback on working software at the end of every iteration. This is typically not the case in a more "phased sequential" (a nice term for "waterfall") approach where the true state and adequacy of the system is not seen until late in development.

The dislike for change, and especially late change, derives from this traditional way development and change happens. Much time and budget has been spent creating and implementing the rather detailed requirements. Many design assumptions have been based on the initial understanding of those requirements. Late changes usually mean much time and budget will be spent going back and updating and correcting intermediary deliverables, not just the code itself. While this is happening, technical work may come to a halt, at least in parts of the system, and people will need to shift to other work to keep busy. Then they will be asked to shift back to where they were and make the changes. Multitasking and interruption in flow leads to loss of productivity in pure terms because of time wasted, but it also impacts people's sense of accomplishment and satisfaction in work.

Of course, under pressure to make changes fast, people may get the work done by skipping certain quality practices (e.g., code reviews and unit testing), working (a lot of) overtime, and skipping updating the documentation. Naturally, all these can be bad ideas. I've mentioned the latter already in Chapter 4 of this series. I'll get to the other two in later chapters.

PART 9

Deliver Frequently

"Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

Deliver working software frequently

Chapter 7 discussed "early and continuous delivery" and Chapter 4 discussed working software. The "frequently" part was implied in continuous delivery when I said "Get working software into the customer's hands right away. Keep doing that every iteration…." What I didn't say was how long an iteration would be. I don't know too many agile coaches and trainers who aren't encouraging clients to consider no more than two-week iterations. That was not the case when the Agile Manifesto was written.

From a couple of weeks to a couple of months

When the Agile Manifesto was written, the various "lite" approaches had cycles from one week to three months. The phrase you see here was, it seems to me, a compromise among all the people involved. With people from XP, Scrum, and like-minded approaches being about half the attendees, I can believe a phrase such as "from one to four weeks" could have made them happy. That would not have worked for everyone, though.

With a preference to the shorter timescale

This phrase, closing out the principle, represents those supporters of a shorter timeframe and perhaps the recognition by the others that this seemed to be the direction things would head in. These days, there is not a lot written or heard about the approaches with longer cycles. Feature-driven design seems to have had the last book written about it in 2002

though the Wikipedia article seems to have had numerous updates over the years. The dynamic systems development method (DSDM) remains an active community that recommends the shorter timescales, though it has rather voluminous online documentation in two sets: DSDM and Atern. (The latter is described as "a framework based on best practice and lessons learnt by DSDM Consortium members" and is the current form of the DSDM approach. Commercial books date back to 2002.)

Why the short iterations?

For some, the idea of putting out, at least for customer review, working software every few weeks (let alone every week like XP teams do) seems inconceivable. And that does mean what I think it means: "not capable of being imagined or grasped mentally; unbelievable." I would recommend that most agile teams start with two-week iterations. I believe it takes three to four iterations before teams begin to understand what it means to work together in an agile fashion. A team may as well get those over within a couple months rather than up to four months. You can always go to three or four weeks later if that seems necessary.

Mike Cohn, one of my CSM trainers along with Ken Schwaber, has said that when Scrum teams have come to him asking to extend their Sprints from four weeks to, say, six weeks, he says, "Let's try two-week sprints instead." They feel they cannot get things done in four weeks and want more time. Cohn says, in effect, let's figure out how much you can really get done in a couple of weeks. The idea is that if you can't reasonably predict what can get done in four weeks, what are the odds you can predict getting potentially more done in six weeks?

I worked with a team once that, because of the company policy, was expected to work in four-week sprints (because that's what was in the original Scrum book and what trainers were saying a decade ago). They wanted to set a shorter target for getting things done for themselves. While working in four-week sprints as far as everyone else was concerned, they established a planning approach that targeted weekly goals for having working software.

Also as I mentioned in Chapter 4, there was a team that worked to get functionality broken into pieces they could complete every two to three days. Neither of those teams ever failed to deliver on their sprint goals and

they were not sliding through on easy effort commitments. Indeed, they seemed to find it easier to get more done by defining smaller amounts of work.

Another advantage, but also a challenge, of shorter iterations is the opportunity for more frequent feedback from customers about what the team has done in the past few weeks. The problem is that customers (and internal stakeholders) are likely not used to such frequent demands on their time to interact with the team. I'll have more to say about this in the next chapter.

PART 10

Working Together

"Business people and developers must work together daily throughout the project."

Business people and developers

The term "business people" can include people with job titles such as customer, end user, product owner, project manager, product manager, executive, business analyst, or subject-matter expert. Sometimes this is generically referred to as "the customer." It's anyone who wants some amount of functionality created and would communicate their ideas for that to those who will create that functionality. They own the project's vision and roadmap, create/gather requirements, help manage the requirement backlog, provide acceptance criteria, elaborate on requirement details, and provide feedback on work done by the development team.

All the very earliest agile teams consisted primarily (if not completely) of software developers who did design, wrote code, tested, and documented results. Therefore, the term "developer" has come to represent anyone on a team who contributes actual effort to some aspect of the deliverable functionality. The Scrum Guide used to say that the only title on a development team was "developer" to emphasize avoiding siloed behavior on the team. The latest version now says, "Scrum recognizes no titles for Development Team members, regardless of the work being performed by the person." One would expect people on the team to have (had) titles such as developer, business analyst, quality analyst or tester, tech writer or documentation specialist, user-experience designer, or database analyst. These people have the responsibility to estimate user stories, assist in story refinement (what some call "backlog pruning"), participate in iteration planning, commit to and deliver on the iteration goal(s), review and demonstrate each iteration's accomplishments with the customer, and perform continuous improvement.

Work together daily

If an agile effort is to focus on "working software over comprehensive documentation", the expectation is that a lot of what is represented in traditional requirements documentation will be communicated verbally and acted upon very soon after that. If a team is working in iterations of

one to four weeks, there isn't much time to wait for answers to questions. Hopefully, answers to questions can come in, say, no more than 24 hours. Any greater delay may mean functionality does not get done during the iteration or, worse, teams will guess what might be intended/needed and spend time developing the wrong thing.

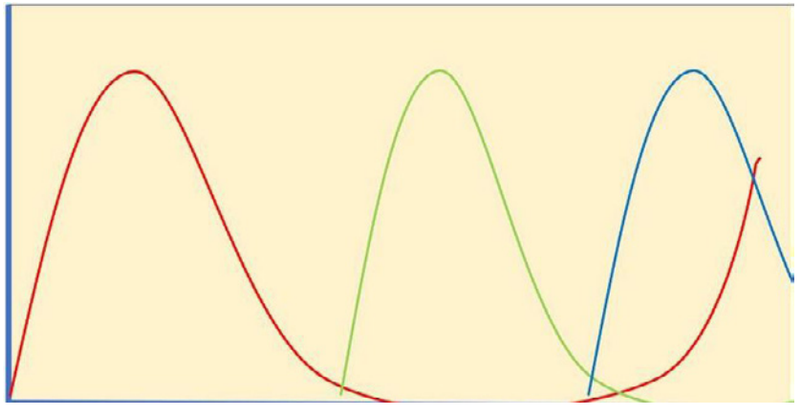
Customers may not have experienced, anticipated, or prepared for this level of commitment to working with teams. In this case, some intermediary may need to exist (e.g., the product-owner role in Scrum). That person must know enough about what is wanted to provide/get answers rapidly based on their understanding of the desired functionality (as expressed in stories).

Sometimes this role is filled by more than one person due to the work required, but a team should only have to listen to one "voice." It will be an impediment to team effectiveness if they have to sort out the opinions of many people to figure out what is wanted. The job of the intermediary is to provide that voice.

Proactive teams and product owners include plans for regular communication during iterations if ad hoc daily communication is an issue. I have seen teams and their product owners meet weekly to go over future (and current) work to prevent too much time elapsing between actual communication. The need for ad hoc communication is certainly going to occur, but if the product owner is separated by substantive time and distance from the team, more formally planned communication is necessary. The goal, however, is to avoid simply falling back on sending text-based communication for everything because actual talking is too hard.

Throughout the project

Though tightly coupled to daily communication, having customers engaged throughout the entire development time is a further challenge for many organizations. The following diagram suggests what a client/project manager may expect their attention commitment to be in multiple, overlapping waterfall projects. They spend considerable time up front during requirements definition and clarification, then expect to go away during implementation, and finally return at the end of the project to review results.



Attention expectation by customers, based on waterfall experience.

People probably recognize the danger inherent in such long periods with little or no communication. However, the traditional waterfall approach presumes behavior like this, trusting in the adequacy of fully defined documentation to serve as the major communication approach.

PART 11

Motivated Individuals

"Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

Before I even get into the elements of this principle, I should say how I have heard people react to it. I have spoken about and trained people on the values and principles many times. When I get to this one, I have heard, more than a few times, a comment like "Well sure, if we could do all that, things would be great." I ask them why from an agile perspective don't/can't they do that then. Making these things true is an important continuous-improvement goal, especially for management.

Motivated individuals

I'm not going to talk about how to motivate people. You've probably heard talks like that. I've given talks like that. When I was asked to do such a talk at one company, I heard the best comment on doing such talks. I used to sit right by a VP in software development who said to me the day after I got the request to do such a talk, "Scott, we don't need talks on motivating people. We need to figure out how to stop demotivating them. People don't come to work here demotivated. What do we do to them after they get here that people seem to feel we need to figure out how to motivate them?" He was right. I still did the talk but it became a workshop on avoiding demotivational behavior.

W. Edwards Deming, in his book *Out of the Crisis* (1986), said, as part of one of his 14 Points of Total Quality Management, "Remove barriers that rob people of pride of workmanship…." If you are not familiar with Deming's work, you should look at it and understand his impact on quality and productivity. The Japanese did, and so respected what he taught them that in 1951, Japan named their national quality award after him: the Deming prize.

Environment and support

If teams are going to be asked to deliver high-quality results to customers, then they should be given all they need and every opportunity to do that. It's as simple as that. "But you don't understand the reality here," I have heard people say. Becoming agile is about changing that reality, not just

trying to push a bunch of new practices (the square peg) into the existing organizational approach (the round hole). This is the reason I think an appreciation for what the agile values and principles have to say is so important for an organization contemplating the transformation to an agile way of being (and doing). Part of that appreciation is that it is a transformation, not just a transition. It's a change to the environment and system within which people work, not just sliding in new titles and ceremonies.

Sometimes, when people think of "environment", they think of the physical structure and the move to open workspaces with no permanent walls and totally moveable furniture. There has been plenty of criticism of such structures. I've seen it work (in various forms) both wonderfully and not so wonderfully. But I've seen that with more typical office and cube structures as well. The idea of the open space was to cut down barriers to the frequent interaction and communication assumed in agile teams. Not everyone is comfortable working that way, and I have certainly seen teams that worked well without this. If the motivation to communicate, collaborate, improve, trust, and pursue excellence is there, people can work in a variety of physical spaces.

Trust them

Maybe this is the hardest thing in the principle to achieve. Much organizational structure and process, as I noted in Chapter 2, seems to exist because trust isn't present. I've certainly seen examples of this.

Before I became involved in agile methods, I worked at a place where the company president's view of quality was that, "Some people just don't do their work right. If we could find them and get rid of them, we'd be fine." His view was that poor quality was about individual moral failure to perform properly. He had turned what Deming decades before had treated as an objective, engineering system matter into a moral, subjective, personal one.

Many years before that, I gave a lecture on quality and process at a local university's MS in software engineering program. I was asked how cheaply one could expect to hire people if the process was rigorous enough. That is, if we don't want to trust/expect much competence from people, how inexperienced (and cheap) a staff could a process let us get away with?

The way many organizational processes grow over the years is often due to similar thinking. When something goes wrong in a project, it's often decided to prevent that from happening again by instituting some more rigorous rules/oversight procedures. Over time, the process bureaucracy grows and grows bit by bit until more time is spent adhering to process regulations then developing the product.

Kent Beck has said (and I'm paraphrasing a bit) that software development is like "driving a car, not aiming a bullet." Rules of the road and various traffic indicators exist as a framework within which we hope most people can safely drive. We expect people to take a constant inspect-and-adapt approach as things may unexpectedly happen and demand a more impromptu response than planned. On the other hand, once you fire a bullet, you've lost control over it, so a lot of up-front planning is needed.

Now, we could try to restrict people's driving mistakes by building all cars and roads like the old speedway ride in Disney parks where you cannot go very fast, there are thick bumpers all around the car, and barriers channel the path you can follow. Some organizational process seems a lot like that. I believe methods should be designed for people who know how to drive rather than those who do not. Deming said that, in most cases, it was the system people had to work under that needed to be reformed rather than the people. Of course, some people need to change how they work and Deming did address training people who need to perform better.

A great deal could be said about trust. But look at your existing processes (do a value-stream map as mentioned in Chapter 7) and ask what the impact would be if teams were allowed to manage each one themselves or between one another. Maybe it's a trust issue if you find this inconceivable.

PART 12

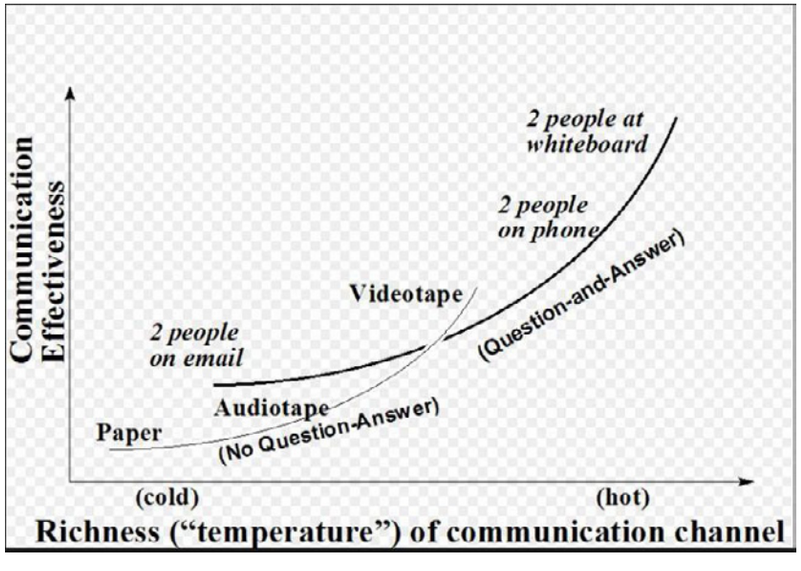
Face-to-Face Conversation

"The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

This chapter, addressing communications modes, will comment upon distributed team issues, daily meetings and iteration reviews as examples of "conversations" that occur.

Modes of communication

In Chapter 3, I showed this diagram and promised to get back to it. Alistair Cockburn had studied teams and came up with this spectrum of communication modes and what he saw as their comparative effectiveness. (Other versions of this diagram that I have seen label the two lines as "non-interactive" and "interactive".) Clearly, the "question and answer" modes are shown as having higher effectiveness and richness.



Comparative effectiveness of communication modes.

Perhaps most people can agree with the intent of this diagram, but some will point out, "Sure, but we have distributed teams." I'll get to that, though it is interesting that the VersionOne survey, run over many years, indicated in their 12th annual survey that about 40% of the respondents felt that adopting an agile approach improved their distributed-team experience

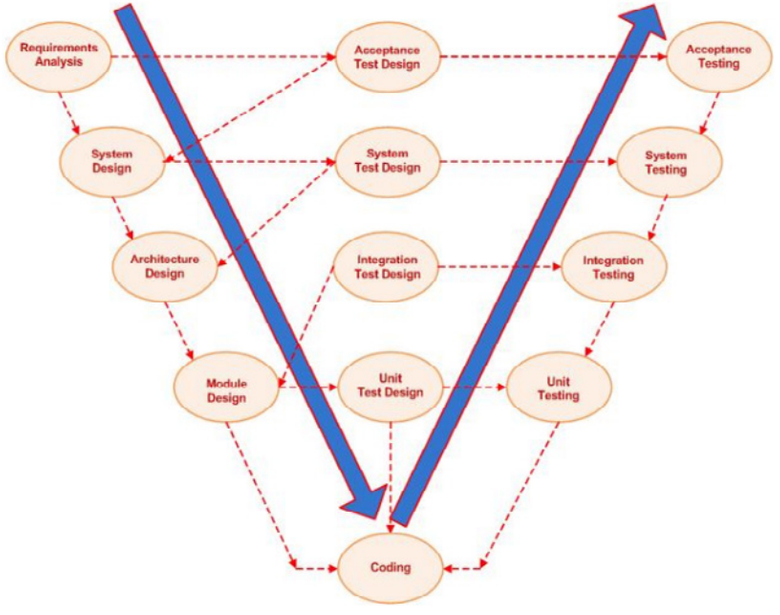
rather than aggravated it; only 17% of those same people initially felt that pursuing agile would help in this regard.

For now, let's just look at some of the modes and what they represent. It is interesting that paper (text forms of documents) ranks the very lowest on this scale yet is traditionally the most commonly (even highly) regarded approach to (archival) communication. Projects spend a great deal of time and money generating requirements specifications, architectural designs, and lower-level detailed designs before they get to implementing anything. Each of these usually involves many people in collecting the information then reviewing and approving it, believing this due diligence is the necessary and responsible way to plan and execute a project. Of course, a major assumption is that people can think of everything up front and not change their minds much later. This is the classic waterfall or phased-sequential philosophy enshrined in traditional project management. The agile approach believes that this is not true and that it is unreasonable to expect people to be able to do this. It is not some form of moral/professional failure when they cannot.

One lifecycle model illustrating this approach and matching it to the verification/validation expected at each phase is the V-model as found in a StackExchange discussion. Each phase of system specification is matched with a testing approach that indicates the focus of the testing taken to complete the system at that level.

At the other end of the diagram is "2 people at whiteboard," representing the face-to-face, real-time interaction highly prized in being agile. People can see one another, which dramatically increases richness since more information is often conveyed by what we see than what we hear (including being able to draw on that whiteboard and rapidly make edits). When you are right there with other people, you can see quizzical looks or when it seems someone might need to say something. You can also avoid most clashes in trying to speak, which most people would like-

ly say is better than trying to conduct such a discussion when people:



- cannot see one another;
- forget they are on mute and talk then must start again when reminded they are on mute;
- do other things while supposedly listening and then ask for things to be repeated; and
- try to talk at the same time then go silent, try to be polite and tell one another to go ahead and speak, then wait in more silence until finally someone does.

Email is at the very bottom of the Q&A line (and can include any form of text messaging). It is interactive in a sense, but not at all guaranteed to be real-time and it has its own issues. Some people like the long email discussion thread as a form of standard documentation. Some like it because it guarantees others cannot deny what was said (which suggests trust issues). Some like it because they don't have to deal directly with other people (see the "cube farm" example mentioned below).

As to the non-Q&A modes, I'd like to describe an example of videotape being used at one place I worked. Instead of having someone at a design discussion taking notes, writing them up, circulating them for edits/approval, then sending the document to clients for review, we videotaped

the session. (It was that long ago. There are certainly better video options these days.) We'd copy the tape and send one to each stakeholder location. They would watch the tape then communicate their thoughts back to us and we could incorporate those ideas into the (ultimately still required) design documentation. What was important was that the stakeholders got to hear everything, not just what we thought was important to document. This included things we might breeze by but the stakeholders picked up on and never would have heard if we'd have sent our version of what they should know. This led to changes that could have only been noted months later when a lot of work had been done and the stakeholders got to see some working software.

You must decide what modes work and when for your own situations, but always ask whether it would be possible to move up and to the right on Cockburn's diagram.

Distributed teams

This principle does not say people must be co-located, only that it allows the most effective communication to occur. If you are going to have discussions among people who are not together in the same room, use some form of video. These days, it doesn't cost much at all to add a USB camera if people are not at a machine with one built in. Unfortunately, I've worked with teams who went months without knowing what one another looked like. My first suggestion was to get pictures of everybody at least. When these teams did get video going, the changes in the discussion were clearly observable. People felt okay in calling on one another, recognizing those puzzled/worried looks. People's attention, or lack of it, to the meeting could be noticed and they could be invited back into the discussion. Putting faces to the disembodied voices over the phone helped with team spirit, which led to increased trust.

But the key imperative for me is that we never let anyone go dark. That is, we make sure everyone feels included in the team and has their chance to contribute to the team. We don't fall back on what's easier to do (i.e., go down and to the left on Cockburn's diagram). This may mean getting some input/feedback using more text-based approaches, but sparingly.

One US-based team I worked with had never had any distributed members. Then, they added two team members in India. The US team held

daily meetings at 8:00 a.m. local time, which could not occur earlier due to childcare considerations. During the non-Daylight Saving Time of the year, that would be about 7:30 p.m. in India. Most companies in India when they work with US companies will shift their work hours from perhaps 10:00 a.m. to 7:00 p.m., so the stretch to 7:30 p.m. wasn't so bad for the two people in India. Sprint reviews were held at 8:00 a.m. as well and often ran until 8:30 p.m. in India, but that was once every four weeks, so still not too bad.

Since retrospectives followed the sprint review after about a half-hour break (by then, 9:00 p.m. in India), the US folks and their Scrum Master agreed that the folks in India did not have to stay on a call (or call back) because it could be 10:00 p.m. in India by the end of the retrospective. I heard about this and told the Scrum Master and local team members that they can't do that." (They were used to me and saw this as an oddly directive statement coming from me.) When they asked why I said "can't", I simply told them that they could not exclude the members in India from the retrospective process. They at least had to get the input from those two people before the meeting if the two of them could not be in the retrospective.

Sometimes as a coach, I get lucky. The next retrospective, that's what happened. Text input from the people in India was sent to the US and was included in the topics discussed by the US members. At the end of the retrospective, the highest-priority improvement item selected was one of the inputs from India. The US members acknowledged that they wouldn't have come up with this on their own, but, discussing it, agreed they should really work on that item.

Dealing with distribution

To overcome the disinclination to move even short distances to speak face to face, one place had a rule that, after going back and forth three times with emails or texting, people had to get up and go talk to the other person. If the distance prevented physically going there, people should call the other person or use some video-based technology to contact them for real-time communication. If the time difference was also large, then people should schedule a time when video communication could happen and speak in real time that way. The idea was to make the effort to at least

simulate face-to-face contact rather than fall back on purely text communication.

I should mention an extreme example of distribution that I have witnessed more than once. I've been in "cube farm" environments where people sat in rows at cubicles walled off in front and to the left and right. If you stood up, you could see over the partitions, but sitting down, it could feel like other people did not exist. In such cases, I have seen people send emails and text messages rather than get up and go to a cubicle near them. It didn't take 30 meters of separation. Sometimes, it was hardly 30 centimeters.

Another such process consideration is to never let people go dark — i.e., make sure everyone on a distributed team knows what is going on in the same way that a face-to-face daily meeting could. For example, I once worked with a team where:

- the product owner was alone in Denmark,
- a developer was alone on the East Coast of the US,
- a developer and tester were together on the West Coast,
- a few developers and testers were together in India, and
- the Scrum Master was alone in Singapore.

It was the most distributed team I've ever worked with yet among the most effective and satisfied in their distributed form. The Scrum Master and I (as coach) were on two meeting calls each day: one in the morning (Pacific time zone) and one at night (Pacific time zone). At the end of each meeting, the Scrum Master would email a short set of notes to everyone on the team about what each session had discussed. People on the team then proactively got in touch with one another when they needed to exchange more detailed information. It was not ideal, but they made it work because they committed to trying to communicate more effectively rather than falling back on creating and sending formal documents.

From my perspective, the absolute minimum would be to make sure people can see one another whenever they meet over a distance. The use of video dramatically improves the relationships among the people on a team. Seeing one another puts a face and a real person to an otherwise disembodied voice over a phone line. This increases the sense of being a real team, which improves the collaborative inclinations and trust. Today, there is no reason not to have video in a meeting.

Daily meetings

You can read a lot about the purpose of the daily (standup) meeting as well as what it isn't (i.e., not a status-reporting session nor a problem-solving one). Personally, I view it as the team's opportunity to make sure everyone knows what's going on within the team and, hearing all that, deciding if there is any need to make changes in the sprint plans. At the end of the meeting, I would want every team member to feel comfortable with the plan going forward for the rest of the iteration. The forms of data that the team might collect in the meeting would be things such as:

- what each person says about what they did since the last meeting, what they plan for that day, and any things in their way (the classic three items recommended for such a meeting);
- the state of the team's progress using some quantitative assessment (e.g., a burn-down chart); and
- input they may request from people outside the team (who may or may not be present at the meeting).

This last point raises the question of who participates in this meeting. It is specifically for the development team, with their Scrum Master/facilitator helping (especially by keeping a "parking lot" of items to discuss after reaching the daily meeting's 15-minute limit). It is not a secret meeting, so I have seen product owners there, people from other teams present, myself standing by, and even management sometimes observing. But all of us not on the development team keep quiet unless we are asked to comment. If we really want to address the team, we should contact the Scrum Master ahead of time who, with the team, could decide if such a request should be fulfilled at the meeting or afterwards. (If they decide they want to say something during the meeting, they could ask if there might be time for their comments at the end.)

After the meeting, the Scrum Master/facilitator can point to the parking lot if there are any notes there, and the team can decide who must be present and how to address any issues noted. With a distributed team, the invitation to the call might indicate 30–60 minutes, but the first 15 are reserved for the daily meeting. The rest of the time is to allow for other discussion when and if necessary, not an indication that the daily meeting will take all that time.

I know of at least one organization of around 50 people where all staff participate in the daily meeting. They work in pairs, so it is like 25 people

speaking. They complete their daily meeting within that 15-minute timebox. People cover their three main statements in 15–20 seconds each on average, making it easy for 25 people to finish in 15 minutes.

Iteration reviews

This meeting, held the last day of the iteration, is for the development team to show to all stakeholders what they have accomplished in the iteration. It is an opportunity for those stakeholders to hear from the team and for the team to get feedback from the stakeholders. The team can hear directly how the stakeholders feel about the work that is presented. Any changes needed can be identified and put on the product backlog for a future iteration. There are, however, some things I'd suggest people keep in mind about the meeting.

Generally, the team members demonstrate their work, but there can be occasions when it would be appropriate for a product owner to do so (assuming the product owner has been closely in touch with the team and knows how to do so). This might be because of language differences between team members and stakeholders or because a product owner knows better how to speak in stakeholder terms. (If this latter point is true, I would suggest an improvement item would be for the team to learn more about how to engage the stakeholders at their level so no intermediary is necessary in the meeting.)

The meeting is not for internal people (e.g., managers within the development company) to engage in debates with one another in front of other stakeholders or to ask, because they have missed some prior reviews, that prior issues be restated/explained for them. I've seen both things happen more than a few times. The product owner or Scrum Master/facilitator needs to step in and, as politely as possible, cut this sort of thing off and deal with it outside the public forum of the iteration review.

Sometimes, especially when people are calling in to the review rather than being in the room with the team, the response from stakeholders at the end of a particular demo of functionality or at the end of the review may be little more than "nice job" if not, in fact, dead silence. That's not much feedback for the team to. The product owner, (supposedly) knowing the stakeholders, should try to encourage more substantive feedback. They can ask specific stakeholders to comment on something they know

was of major interest for that stakeholder, politely putting the stakeholder(s) on the spot, as it were, to say something. Discussion at the review is important. It should not be a one-way show-and-tell session by the team.

PART 13

Working Software... Again?

"Working software is the primary measure of progress."

Wait… wasn't working software one of the Agile Manifesto's values? Yep. So they repeated it again. Yep… they did.

I asked one of the manifesto authors about this and his first comment was, "Well, we thought it was important enough to repeat."

Of course, it isn't exactly a repetition. The value states the preference for having working software over only having comprehensive documentation. This principle states that working software is the main progress metric.

Measuring progress

Progress in projects traditionally has been measured by completion of tasks in the project plan and how closely that tracks to expected completion: i.e., numbers of tasks completed and whether they complete at the time originally estimated. In a traditional waterfall project, a substantial number of tasks will be completed before there is any actual software working, i.e., before functionality is designed, coded, reviewed, tested, and documented. Indeed, a project schedule can be from two-thirds to three-quarters completed (with all the associated tasks) before there is any software verified and validated.

One approach to traditional tracking that associates tasks done with project value is earned-value measurement. Very simply, one assigns a dollar value to tasks based on the hours estimated and an estimated cost paid for those hours of work. This can mean something like deciding the cost per hour paid to a person who performs such tasks and multiplying that by the hours assigned to the task. Or it may mean taking the total estimated cost of the project (however that is computed) and dividing it equally across all the tasks based on the percentage of hours each represents for the entire project. Either way, this gives a value for each task performed. As a task is completed, its value is earned, accumulating throughout the project.

But this doesn't, and nobody suggests it should, represent value delivered to the customer. It just means, given that every task is needed, that tasks and hours spent can be used to track percent completion. Hence, it is a progress measurement since it shows how much of the work of this project has been completed at any time. As you will see if you check Wikipedia, there are a variety of numbers that can be calculated and used to compare planned to actual progress.

The agile approach simply takes the position that progress should be measured based solely on how much working software exists at any point in time. One could use story points, for example, instead of hours or dollars as the baseline number. As the team completes requirements, the value is expressed in completed points and you can compare that to the total points for the entire project/release to compute percent completed.

Burn-up chart

One common visual tracking approach for this is a burn-up chart. The diagram shows a simple form of this.

The vertical axis represents story points while the horizontal one represents time (in this case, days in the iteration). The red line is an ideal or baseline drawn from zero story points done at the start of the iteration to the total number of story points planned for the iteration (35 in this example). The blue line tracks the actual number of accumulated story points completed day by day in the iteration.

From this, the team and product owner (and anyone else for that matter) can see the progress the team is making toward completing all the stories they planned for the iteration. Being under the ideal (red) line means the

team is behind in completing stories. Being above the ideal line would mean the team is ahead. The chart does not explain why a team might be behind or ahead, just that it is.

From a product owner's long-term view, the chart could show total points for an entire project/release with the time element being iterations in the project/release rather than days in a single iteration. Both forms can be used. If a team has relatively small-sized stories that they complete regularly, a chart of daily iterations produces a somewhat smooth-looking graph. If the team has larger stories that take many days to complete, you get a more stepped graph, which means that the graph is flat for days at a time until a stories get done, and then it jumps up as those points are recorded. The full project/release version will produce a stepped graph since points are only recorded at the end of each iteration, but it will be a smaller-looking jump each time given that the chart tracks an overall larger baseline of points.

(I should note, however, that simply counting stories completed compared to total stories for the iteration/release is another approach used without using points. Such an idea can be found described in:

Do You Need to Use Story Points to Track Velocity – an IBM article

How estimating with story counts worked for us – a ThoughtWorks article

Story Counting – a Martin Fowler blog post

Stop Using Story Points – a Joshua Kerievsky blog post

How to Enable Estimate-Free Development – a Dave Rooney blog post)

Definition of done

Briefly, to claim that you have a piece of working software, all the quality criteria established are expected to be met. This set of criteria is typically called a Definition of Done. If all the criteria are met, everyone on the team agrees that the requirement is done. Teams should have agreed upon such a set of criteria (at least as an initial example) before starting any work. During a project/release, they will likely adjust the criteria, perhaps formally reviewing it as a part of the retrospective.

Sample items that might be found in a definition of done would address:

- design tasks to be performed,
- coding needed and coding standards to be used,
- unit testing expected,
- peer (or other) reviews to be done,
- user documentation to create or update,
- successful build and test runs,
- customer acceptance criteria met (often called conditions of satisfaction),
- non-functional requirement expectations for things such as performance and security, and
- any internationalization and localization concerns.

Such criteria would be expected to be used for any requirement, though some may not need all the criteria, and some might need additional criteria.

Having met the criteria, the work would be shown at the next iteration review for approval by customers/stakeholders, and the story points would be counted as completed. This would be the measure of progress, regardless of other possible deliverables that might (have to) be generated to get to this point.

PART 14

Sustainable Development

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

Sustainable development

Sustainable development is a development approach that can be conducted over time without burning people out. Agile advocates avoiding mandated overtime. If the latter exists, people are not taking a reasonable approach to planning work. Some might say, given how their organization works, that this ignores reality. As I mentioned in Chapter 11, agile does not ignore reality but aims to change reality.

It is not merely the agile view that it's a bad sign to have to routinely employ overtime to meet a deadline. At least by 1997, one of the main proponents of classical software methodologies noted how common the so-called "death march" projects were. The Project Management Institute also identifies such an approach as undesirable, citing an article describing many signs of project trouble related to the death-march mentality.

A particularly scary sign is when a project starts with overtime already required to meet the deadline with all the functionality expected. What happens when the unexpected happens during the project? Where does the time come to handle that? Even more required overtime?

Tom DeMarco wrote an interesting book (Slack: Getting Past Burnout, Busywork, and the Myth of Total Efficiency) just as the agile movement was getting started (2001) discussing filling (and overfilling) people's time and what that means for effectiveness, including the impact of multitasking.

Constant pace indefinitely

The idea of a constant pace is the idea of working within a predictable rhythm, which the agile iteration and its various elements (i.e., iteration planning, daily meetings, reviews, and retrospective) offer. I have worked with teams that have continued together (with not very frequent personnel changes) for almost a decade. They are still going. These teams have

worked iteration after iteration (13 a year using a four-week iteration length) and been through well over 125 such iterations. The ability of team members to work together smoothly and effectively is an asset to the company. Deliberately splitting the people up, scattering them to other efforts, treats them like an inanimate resource that can be shifted with no loss to the essence of what that resource represents.

The challenge to an organization is to maintain such teams. This is far easier in a product-oriented company than in a project-oriented one. At the end of most projects, teams may routinely split up as people are assigned to new projects. In a product environment, the team stays with the product, release after release, building significant domain expertise as well as growing their skills and ability to work together. I have seen such teams plan with very few words exchanged other than those that help clarify the functionality they are to create during the upcoming sprint. Someone will say, "I can handle that" and that ends the discussion on that point. They have learned that when anyone says something, they mean it and will do it, and they also know that if they need help, other team members will support them. Communication and trust are high on such teams (though the verbiage needed to communicate is low as much is "said" through actions, not words).

What may be more challenging is the ability of sponsors and users to maintain such a pace because such a steady commitment to work with a development team is not usually something they are used to doing (as noted in Chapters 9 and 10).

PART 15

Technical Excellence

The first eight principles in many ways describe what might be viewed as the organization helping to make the work of teams more effective and less stressful. The last four principles, viewed in one way, are specific obligations for teams to pursue to respond in an agile fashion.

"Continuous attention to technical excellence and good design enhances agility."

Continuous attention

Part of the urgency which agile teams are encouraged to pursue includes a focus on continuous improvement. Agile teams and individual members should always be thinking about what they can do to improve how they work. This can happen any time during an iteration. The retrospective is just to imbed such a focus as a formal part of the agile cycle. The idea of continuous attention means always looking around, noticing what is going on in the team, and considering how things could be better.

For example, at the end of each work day, members could take perhaps five minutes to consider their day and ask themselves what they could do tomorrow to help the team work better. I would expect this kind of thinking on the part of the team's facilitator (e.g., Scrum Master) and product owner, but it is something anyone on the team could consider.

Technical excellence

One critical improvement to consider is how to grow the technical abilities of the team members. One team I worked with decided that they would like to get better at object-oriented development and decided to use Robert Martin's book Clean Code to do so. However, the book is 464 pages long. They were not going to do this in one sitting. They chose to have each person on the team individually take on a chapter during an iteration and spend some time to prepare a presentation on the essence of that chapter as part of their retrospective. The team would try to put that idea into practice during the following iteration (while the next person in line tackled the next chapter of the book). It took them many sprints to get through the book's 17 chapters, but they did it and felt it was worthwhile to take the time to do so.

Testing practices are another area where becoming better will lead to greater agility. Being able to test frequently and easily will encourage more complete test coverage and help ensure new changes do not break existing functionality. (Recall the two company examples about frequent testing in Chapter 8.) With today's IDEs, there is considerable support to do this at the team level. This would reduce the likelihood that defects escape the team's iterations and cause problems (and rework) further into the project/release.

Good design

What makes design "good" can mean different things to different people. One way for a team to pursue this and avoid many ongoing arguments about what "good" means is to develop/adopt a set of coding standards for the team. One aspect of what "good" might mean is that code is easy to maintain over time — i.e., people can go into the code to make changes and not have difficulty understanding how to do so because of its structure and how things are named. They will then be less worried about making such changes.

Another aspect of "good" might be adherence to certain object-oriented design principles (the SOLID principles that Martin covers in his book). For example, there is the single-responsibility principle. As Wikipedia puts it, "every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class." Before object-oriented design was popularized, we used to talk about this as concern for "cohesion". Cohesion would be high when every part of a module was related to accomplishing one thing. (A related SOLID principle is dependency inversion, which we used to call "coupling". Coupling was low when a module did not depend on how other modules carried out their work. This allowed changes in modules that did not ripple through other parts of the system.)

Regardless of what the actual standards are, having the entire team agree to them (and constantly look to see if improvements can be made in them) is important.

Enhances agility

How this all enhances agility is reflected in how such standards and ideas improve the quality of the product and the ease with which it can be reliably changed. If you do not have to worry about whether changes can be made safely, you are more likely to be willing to make more changes. Thus, you can move quickly while maintaining your balance, i.e., your system stability.

PART 16

Simplicity

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

The principle is rather straightforward. All the engineers I've met believe in this idea. The goal is not to over-engineer something. Ward Cunningham speaks about working with Kent Beck. When they'd get stuck, Ward said he'd ask, "Kent, what's the simplest thing that could possibly work?" Rather than sit and discuss for too long a time, the agile advice is to try to make it work in the code, i.e., produce some form of working software and determine whether what you have will address the design goal.

My mother taught me some basic cooking ideas when I was a grade-school kid. A key one was "you can always add more but it will usually be impossible to take it out." Incrementally determining if the combination of ingredients is good allows you to test the mixture at some level. If you go too far without such testing, you may end up either throwing it out or adding more of some ingredients to balance out the excess of another, producing more final product than you wanted, which may be wasted.

An incremental approach to a more complex design will make it easier to verify that where you are at any moment seems to be working. It will also allow you to back up to a prior stable state more easily. It can also avoid "goldplating": the tendency to add more than was requested/needed.

Of course, just doing simple things all the time isn't a commandment. You'd ask this question "if you're not sure what to do yet", as stated in Cunningham's wiki notes. But thinking in simple terms whenever possible is essential to good design practice.

PART 17

Self-Organizing Teams

"The best architectures, requirements, and designs emerge from self-organizing teams."

The best

I hear the most controversy expressed about this principle because people think it implies that a team of non-experts can presumably come up with better ideas/solutions than a skilled, experienced expert. The ideas of experts would certainly be included. Indeed, have the expert be a member of the team. This harkens back to the principle of giving the team the environment and support needed to do the work expected.

Regarding the prerequisites, about 14 years ago a book was published called The Wisdom of Crowds. It spoke about how a group could come up with a better decision than any individual member of the group, including an expert. However, it wasn't just any group that could do so. There were some prerequisites (with my agile-related comments):

- Everyone in the group would have individual perspectives and ideas — have a cross-functional team of people motivated to come up with a good solution.
- People's opinions would not be (unduly) influenced by others — avoid groupthink.
- It's possible to draw on diverse knowledge — information is not withheld from the group.
- There is a way for all the opinions to be aggregated in some way to arrive at a collaborative decision — people can communicate with one another freely.

The belief is that the collective decision, given these prerequisites, would end up better than if the expert were left alone to come up with the solution.

Self-organizing teams

The phrase "self-managing, self-organizing teams" is used to indicate that teams both organize themselves as they see fit to accomplish work for each iteration and then manage themselves to do that work. Whatever

is within the team's scope is for the team to decide how to carry it out. This means that teams do their own estimates, determine their own tasks, work together and individually to perform the tasks, present their work at the end of each iteration, and verify with those outside the development team (e.g., product owner, other teams, management, customers) that the work meets expectations.

It can take a while for a team to understand how to do this as many people are accustomed to following directions from others. That is, a manager of some sort will have made (or at least been required to approve) many of the decisions that agile teams are expected to do for themselves. This does not mean managers have no place in an agile environment (though some early agile material sounded that way). However, the goal for managers is to take a servant-leader approach concerned with providing the team with whatever they need to be successful rather than to expect the team to behave in a way that the manager is looked on as the success.

I had a manager once who knew very well what agile was about. He was, however, very sharp and was used to being directive in how he approached his teams. He came to me once and said, "I know I am not supposed to interrupt the team and tell them what to do, but I am worried this team is not following the agile approach well. I don't think they use the retrospective effectively." That is a legitimate concern; a team should be trying to use the retrospective effectively. What I told him was, "The next time the team is due to have a Retrospective, go up to the Scrum Master a short while after that and say, 'Is there anything the team came up with at the Retrospective that I can help with?'" This does three things without being directive or intrusive:

- it makes clear the expectation that the team held the Retrospective;
- it makes clear the expectation that the team came up with some improvement item(s) to pursue;
- it offers to help them achieve the improvement(s) where possible.

If the manager got some sort of "deer caught in the headlights" response, he could pursue the subject a bit further by asking, "Was there difficulty in holding retrospective or settling on improvement items to pursue?" Again, the manager would be asking, not telling, but expecting some reasonable answer.

Teams can still be encouraged to self-organize and manage with this approach, but it would be made clear that they are expected to do so.

PART 18

Team Reflection

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

It is somehow appropriate that this is the last principle, given that the first value is about individuals and interactions. I think this principle is significantly linked to that first value. Retrospectives are a key aspect of how individuals interact in an agile context. Unfortunately, retrospectives, due to schedule pressures, can be skipped when it is felt by (or communicated to) a team that there is no time to hold the retrospective.

This last chapter will be somewhat longer than any of the others because I believe there are a lot of important things to say about retrospectives.

Regular intervals

The point of the retrospective is continuous improvement. To be continuous means improving with some frequency. The minimum frequency is at the end of every iteration. The classic "lessons learned" or "post mortem" sessions would occur only after the end of an entire project or release. There are several problems with this:

- At the end of a project/release, people are often dispersed to new projects/releases, making it hard to get them together for such a session.
- Since such sessions occur usually months after the start of the project, some people may not even be around to participate and those who are may not remember the things they thought of months ago, which could be mentioned at the session.
- A lot of what is discussed in such sessions is about the things that went wrong and how to fix them for the next project/release, so the sessions don't help the project that just ended.
- The sessions are often management-driven events at which some people may not want to speak up because of concern for how they might be viewed if they raise an issue folks would rather not face.

For reasons such as this, the frequency (and participants mentioned below) are why retrospectives are expected to be held as they are.

On the other hand, reflection and adaptation does not have to wait until the end of the iteration. There is an opportunity for this daily as a team

and individually. The daily meeting may raise issues that the team should address immediately instead of adding them to a list for the end of the iteration. At the end of each person's day, they can ask themselves, "What can I do tomorrow to make our team more effective?" I would expect a team's facilitator, at least, to do this.

The team

Who should participate in the retrospective? The Scrum Guide states that it's for the Scrum team and defines that team as consisting of "a product owner, the development team, and a Scrum Master". The latter two are rather universally seen as participating, but that is not always true for the product owner. Sometimes, development teams look upon the product owner as management. As managers are not included in the retrospective, product owners frequently get excluded, too. This is unfortunate since they have direct connection to the work done and clearly have opinions like anyone else on the Scrum team about how things went during the sprint.

I believe there can be approaches to involving a product owner in the retrospective that do not completely exclude them:

- They could attend the beginning of the retrospective, be heard and engage in some discussion, then leave.
- They could offer their perspective to the Scrum Master before the retrospective and those ideas, at least, could be provided for consideration in the retrospective.

Neither of these are ideal, but they are better than total exclusion. In both cases, the product owner should receive feedback on what happened to their ideas and what the development team came up with as improvements.

(I do realize that some teams may have some private issues they wish to work on, but there should be some improvement(s) that can be shared publicly.)

Become more effective

The idea of continual improvement goes back at least to the early lean work in Japan after World War II. It is known as *kaizen* in that context. Improvement does not have to be some large change, just continuous change for the better. Even something that seems to work well might work even better and can perhaps be improved upon.

Certainly, the retrospective is not just about "fixing" things that didn't go well. I have heard teams say they don't need to hold a retrospective because everything went fine in their iteration. This has always indicated to me that they view the retrospective as a problem-fixing meeting. No problems; no meeting. Since the goal is about improvement, unless a team thinks they are perfect at being predictable, being productive, and producing defect-free work, there is always something to improve.

A typical focus of such meetings includes:

- What went well that a team would like to keep doing, i.e., make a part of their regular work process?
- What didn't go so well and should be changed for the better (or perhaps dropped as not useful)?
- What could be tried that the team has not yet tried, i.e., learn something new or experiment with a new idea?

If a team feels that don't have anything in that second category, they should still consider the other two.

Tune and adjust

One common problem that often inclines teams to abandon retrospectives is that they make lists of improvement ideas but end up doing nothing to improve. After doing this a few times, the team will wonder why they should bother.

It is imperative that teams move from list making to prioritizing their ideas, selecting some (maybe one is enough), determining what it will take to pursue the improvements (e.g., tasks, effort estimates), and committing to follow through on one or more. I have seen some teams create an improvement backlog where improvement ideas are treated like requirements.

At the daily meetings in the next iteration, the team facilitator could ask about the items the team said they would try to pursue as improvements. This could help the team remember which items should get attention during the iteration.

At the next retrospective, these items should be part of the input to consider, i.e., how did the team do in working the items during the iteration?

Retrospective "smells"

Besides endless list making that goes nowhere, other things may discourage a team from conducting retrospectives or using them effectively:

- Boring/repetitive process — Conducting the retrospective the same way every time can become boring. The classic brainstorming approach of going around the room with each person offering up an idea, having it noted down, and doing that until everyone thinks they have no more to say seems rather boring to me. At the very least, it does not engage the whole team right away as people have to wait for each person to say something (or pass) before anyone else speaks. If someone has a lot of ideas, many people are going to sit there until that one person runs through their list. Also, hearing what one person says may incline people not to repeat a similar idea as they do not want to seem redundant, though the difference in how it is presented would be useful to hear.
- Victim of others — Sometimes most of the things a team talks about are problems that come from people outside the team, which, while perhaps true, can result in the team feeling there isn't much they can do to change things because it is others who must change/improve. A facilitator, while acknowledging the truth of what is said, needs to ask, "What could we do about this? Who might help us with this issue? How might we help ourselves?"
- Blaming session — This occurs most often if there is not a sufficiently trusting and collaborative spirit on the team. People will look for whose fault it was that they faced various problems. You want to address issues, but not so people are put on the defensive. Avoiding accusative words like "you", "they", he", or "she" and substituting "we" helps a great deal. Handling this may require using some anonymous data collection (see below) as people may simply not raise the issues

to avoid the blaming, but the issues nevertheless probably need to be heard.

- Issues that are hard to talk about — People may avoid raising issues that are specific to individuals on the team, while perhaps important to address, because it is uncomfortable to do so. Issues may be raised in a way that promotes defensiveness and destructive discussion, like blaming.

While the retrospective is an important team activity, it may also be the hardest one for a team to engage in effectively. The fundamental considerations of communication, collaboration, commitment, continuous improvement, trust, and excellence discussed in Chapter 2 are particularly important in the retrospective context.

Some retrospective ideas

I have some strategies to try in retrospectives that I have used myself or with teams.

To allow the team's facilitator/Scrum Master to participate as a Scrum team member and not have to "switch hats" from that role to contribute ideas, get an outside facilitator to run the session. This could be another team's facilitator/Scrum Master; the two could agree to run one another's retrospectives, bringing in an otherwise uninvolved person to facilitate the meeting. You could also go to your training (or perhaps HR) department and look for someone trained in facilitating meetings and have them do it. This also serves a useful purpose in introducing people outside the agile effort to a bit about the process. I've also had teams who rotated facilitation among team members. Each iteration, a different team member prepared for facilitating, looking for new and different ways to possibly run it. This has the advantage of getting everyone to self-manage this activity.

As noted above, the traditional brainstorming approach has potential problems in collecting feedback. A simple way to overcome many of these is to ask people to write their ideas down on note cards or stickies. This allows everyone to be busy at once and to be able to get out as many ideas as they like without anyone having to wait for a turn to do so. When people seem to have exhausted their ideas, you can have everyone get up and put their ideas up on a wall, board, etc. possibly under categories like

"People," "Culture," "Process," "Environment," and "Tools". This begins the process of grouping ideas together (affinity) for further consideration. It also makes it easy for people to step back, look at the overall patterns, and reflect on those before diving into the details of each specific idea. When I have done this, I usually say the categories mean the following:

- People — interactions among individuals;
- Culture — the organization's beliefs about interactions;
- Process — the ways people work to get things done;
- Environment — the physical environment in which people work;
- Tools — the things people use (manual or electronic) to do their work.

It is almost always the case that the volume of stickies skews toward the first couple, then less for the third, and fewest for the last two. I think you can guess the kind of discussion that ensues when people see this, regardless of how obvious it might be. Seeing the "obvious" presented physically just drives home the impression.

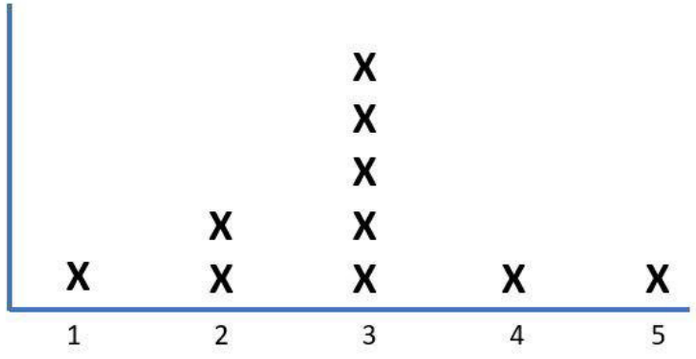
I also used this approach and categories when a Scrum Master asked me to run a retrospective because they felt it would turn a bit ugly due to what had gone on during the sprint. I started the retrospective by acknowledging that I had heard the team had a rough sprint. I asked that instead of writing down ideas about the things that went badly, they should write how they would have liked things to have happened and put them up under these categories. Then I asked them to discuss how they thought they could make these things happen. This turned into a much more positive retrospective than the Scrum Master had (probably rightly) feared.

Collecting Feedback

One issue with collecting feedback can be doing this openly versus more anonymously. If people are reluctant to raise issues for any reason, perhaps some form of anonymous data collection could work. People could still write things down as noted above, but perhaps they'd all be handed in and shuffled before dealing with them further.

I had a Scrum Master ask me to facilitate a retrospective because he wanted to raise the issue of how people felt about teamwork, but he was not sure if people would speak up and was concerned that the feelings were somewhat negative. I asked people to rate their sense of teamwork/spirit on a 1–5 scale (worst to best) and write that number on an index card

which they would hand to me to and I'd shuffle up. A "1" meant they felt the spirit was bad and a "5" meant they felt it was wonderful, etc..



Bar chart.

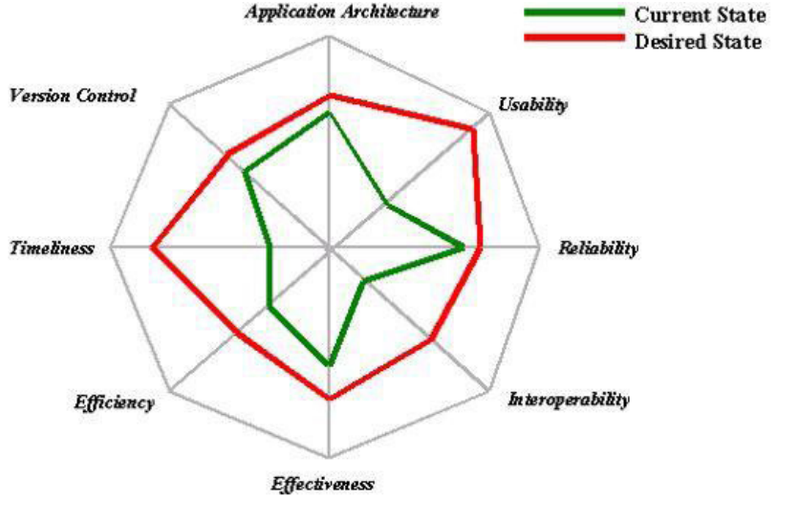
As I would read out the numbers and the Scrum Master would create a "bar chart" like this, putting an "x" whenever a specific number was read out, stacking each "x" when the same number was read out again. I then asked everyone what they thought of the Team's overall data. Some folks were surprised it was not more negative. Everyone could see perhaps there was some work to do but it wasn't as bad as many feared. Even one of the most frequently disengaged members spoke up a lot.

The Five "Whys"

One technique for dealing with the data collected is called the "5 Whys" and is a root cause analysis approach. You take an issue and ask why that seemed to be true, then ask why that response might be true, and repeat this until you think you've hit the root cause rather than just symptoms of the root cause. For example, a team might say, "We have problems getting stories done each sprint." In response to why they think that happens, they could say, "Because other teams we depend upon don't deliver what we need." Asked why they think that happens, they may say, "Because coordination with them is hard." And this could continue until it was felt something was stated that could be changed, A solution would ripple up the line of issues to help the overall problem.

Data "Tools"

There are also a variety of what can be called tools that help collect, organize, and visualize data. A book called *The Quality Toolbox,* published by the American Society for Quality (ASQ) and also available on Amazon, is filled with examples and explanations of such tools. I have used many of the examples, which include traditional graph types like those in spreadsheet programs and more sophisticated things such as statistical control charts.

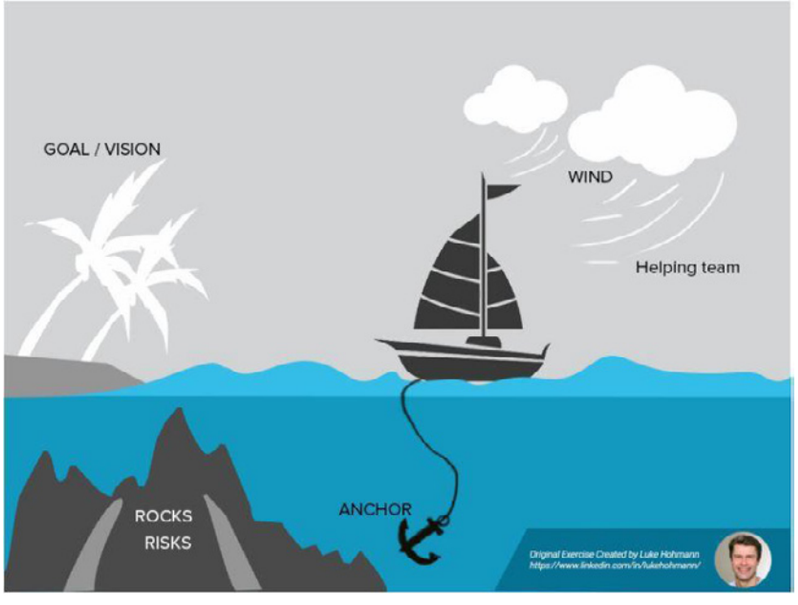


Kiviat diagram.

One I like a lot is the Kiviat diagram, also called a spider chart or radar chart. Topics for which feedback is desired are listed around the outside. The radial lines would usually have tick marks with those closer to the center being the low end of the scale and the outer end being higher. For each topic, ask people to provide numeric values. In the example above, two such values are requested: where people would like to be (red line) and where they feel they are (green line). Now you can average the data or plot each person's two values, connecting the lines as shown. Again, stepping back allows all to see what the chart tells you about the areas where a desired state is furthest from actual. Then you engage in discussion about what to do about what you see.

Another idea is something like the sailboat exercise originally created by Luke Hohmann. (This specific diagram comes from Luis Gonçalves' website and can also be found in his and Ben Linders' book *Getting Value out of*

Agile Retrospectives, which started as a mini book on InfoQ: https://www.infoq.com/minibooks/agile-retrospectives-value). Besides this depiction, I have also seen pictures of a sun and a shark added to such a diagram.



Sailboat diagram.

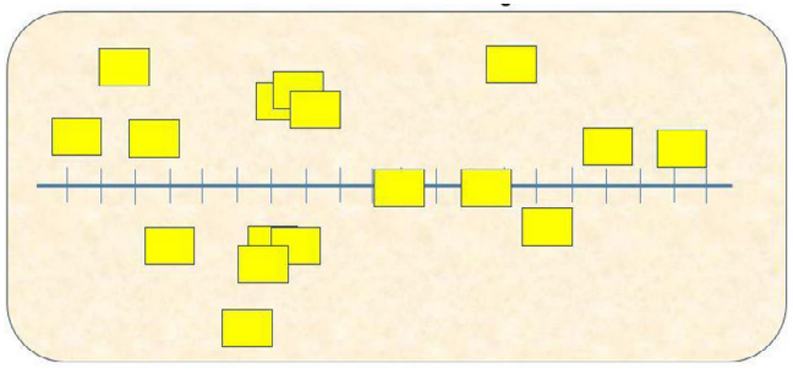
People write down their ideas on stickies then put them where they think they belong on the diagram. The meaning of the individual images could be:

Palm Trees — what the Team goal/vision was for the iteration;

- Rocks — the risks the Team faced and may have hit (the stickie is on the rocks) or avoided (so the stickie is not exactly on the rocks);
- Anchor — something that held the Team back or "weighed them down";
- Wind — something that moved the Team forward;
- Shark — something that "bit" the Team during the iteration;
- Sun — something that "brightened the day" for someone.

Again, after all the stickies are up, you stand back and get an overall sense of what the image suggests with the stickies present. Then you can decide where to begin your more detailed focus.

Finally, there is what I call the "running retrospective", which, as far as I know, I invented. I certainly did not copy from anyone knowingly when I came up with it several years ago. You take a piece of long, wide ("butcher") paper and draw a horizontal line that represents the iteration. Each tick mark on the line represents a day in the iteration.



Running retrospective.

Throughout the iteration, at any time, a person could write a note and stick it on the mark for that day. I presented this to a workshop table at a Scrum Gathering and Alistair Cockburn was part of the table. His wonderful suggestion was to have people put the stickies (a) above the line if it was a positive experience (and the higher above the more positive), (b) below the line if negative (and the lower, the more negative), or (c) on the line if it was just an observation that the person wanted to make with no strong feeling one way or the other.

At the retrospective, there would already be a lot of data (to which more could be added) and that might move the retrospective right into discussion and selection of worthwhile improvement ideas. Of course, people could see patterns forming throughout the iteration and might want to address something they see before the next scheduled retrospective (since the need/decision to improve need not wait to the end of the iteration). But the overall idea is that idea collection runs throughout the entire iteration.

What next?

There are certainly many more things that can be said about retrospectives. Just type into your browser "agile retrospective exercises and games" (or some combination of the words) to find plenty of ideas. There are also books such as *Agile Retrospectives* and *Project Retrospectives* that you can look into.

Whatever you do, do not allow people to give up on the retrospective.

Epilogue

Hopefully, this book has offered some worthwhile perspectives on the Agile Manifesto's values and principles. The next step is yours: trying to put them into practice in your organization or situation. Use them to stop and think rather than simply rush to do.

If you have not had any formal training in these ideas and the basic agile lifecycle (or people around you have not), I strongly encourage you to get that training. I'd then encourage you to seek some initial coaching to help you get moving in a successful direction. Neither of these must, or should, cost a great deal.

One older and two recent perspectives

As noted in Chapter 1, there have been a variety of proposals for updating the manifesto. Kent Beck, in 2010, spoke of one evolution of the Agile Manifesto's value statements:

Team vision and discipline over individuals and interactions

Validated learning over working software

Customer discovery over customer collaboration

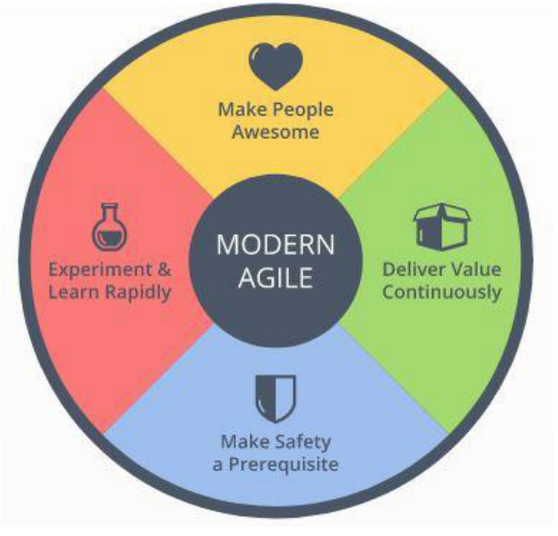
Initiating change over responding to change

This thinking seems to fit nicely with two recent perspectives.

Four values and 12 principles plus many methods, frameworks, and practices — that's a lot to absorb. A common question is just what the essence of being Agile is all about. A couple of people have come up with some short and I believe easily understood ways to think about this that I have found particularly interesting.

Modern agile

A few years ago, Joshua Kerievsky presented his ideas in a form he calls "modern agile" to address what he felt was "antiquated agility". I will let his words (with some editing) speak for themselves and suggest, if they sound interesting, that you investigate this further.



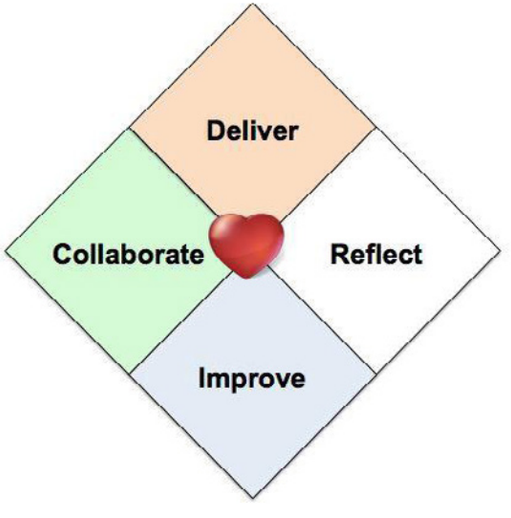
Modern agile.

Kerievsky has said:

- Make people awesome — In modern agile, we ask how we can make people in our ecosystem awesome. This includes the people who use, make, buy, sell, or fund our products or services. We learn their context and pain points, what holds them back, and what they aspire to achieve.
- Make safety a prerequisite — We actively make safety a prerequisite by establishing safety before engaging in any hazardous work. We protect people's time, information, reputation, money, health, and relationships. And we endeavor to make our collaborations, products, and services resilient and safe.
- Experiment and learn rapidly — You can't make people awesome or make safety a prerequisite if you aren't learning. We learn rapidly by experimenting frequently. We make our experiments safe to fail so we are not afraid to conduct more experiments. When we get stuck or aren't learning enough, we take it as a sign that we need to learn more by running more experiments.
- Deliver value continuously — Ask how valuable work could be delivered faster. Delivering value continuously requires us to divide larger amounts of value into smaller pieces that may be delivered safely now rather than later.

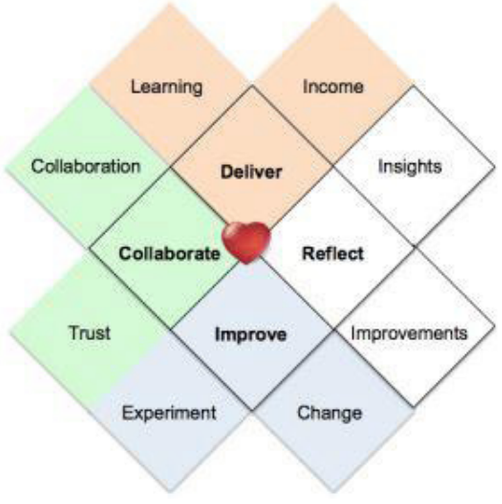
Heart of agile

A few years before I encountered modern agile, I heard about Alistair Cockburn's proposed Heart of Agile as a succinct way to "get back to the center of agile" after what he felt had been many years of "overly decorated" ideas and practices. He feels that the four words in the associated diagram "don't need much explanation. They don't need much teaching. With the exception of 'Reflect', which is done all too little in our times, the other three are known by most people. You know if you're doing them or not."



Heart of Agile.

These four words can be expanded as follows:



Heart of Agile first-level expansion.

And even further into:

Heart of Agile second-level expansion.

Cockburn reminds us that "the point of the Heart of Agile isn't to pretend that complexities and subtleties don't exist; it is to remind us to scrape away all the complexities from time to time, and restart from the beginning."

Over the years, I have always found Cockburn's ideas and presentation of those ideas significant. The first book of his I read, *Agile Software Development* (revised in 2006), covers a wide scope of not just agile ideas but how they relate and can blend with other ideas. Another of his books, *Crystal Clear*, covers his own views on implementing an agile approach and contains many interesting ideas.

One of those ideas is that there is more than one set of practices that a project may need based on the perceived impact (loss) if a project is not successful combined with the number of people who must be involved in the project. Greater potential for loss would suggest more rigorous verification and validation practices make sense. Larger numbers of people imply more formal communication considerations.

Another is that he advocates building up your approach rather than tailoring it down. That is, start with an approach that is the simplest thing that could work and add rigor where it seems needed rather than start with a very rigorous, complex approach and throw away things you feel you don't need. It is better to encourage people to add rigor than to encourage the mindset of trying to get away with less.

Further Reading

I hope the book has interested you enough to consider some of the ideas in it for your own practice of agile. Here are some books I think you would find interesting to further explore Agile and Agile-related ideas:

Extreme Programming Explained by Kent Beck (explains development practices and the XP approach to "managing" a project by the team)

Balancing Agility and Discipline by Barry Boehm and Richard Turner (a book addressing dealing with more traditional, more rigorous development demands while pursuing Agile ideas)

Management 3.0 and *Managing for Happiness*, both by Jurgen Appelo (a book addressing complexity and management in complex situations and a book about techniques, exercises, etc. to use in working with people and teams)

Coaching Agile Teams by Lyssa Adkins (which is just what it sounds like, offering advice to coaches, Scrum Masters, managers, and anyone who wants to help teams grow and develop, including team members themselves)

Succeeding with Agile by Mike Cohn (implementation advice when working in a Scrum approach)

Essential Scrum by Ken Rubin (another "guide" to implementing Scrum)

Kanban by David Andersen (a key book on applying Kanban, especially in a software context)

Scrumban by Corey Ladas (combining Scrum and Kanban approaches to managing work and workflow).

Lean Software Development, *Implementing Lean Software Development* and *Leading Lean Software Development* by Mary and Tom Poppendieck (applying Lean concepts to software development)

The Software Project Manager's Bridge to Agility by Michelle Sliger and Stacia Broderick (Michelle was one of the people that advised PMI in the creation of their PMP-ACP program)

Agile Product Management with Scrum by Roman Pichler and *Scrum Product Ownership: Balancing Value from the Inside Out* by Bob Galen (Roman's is perhaps more frequently mentioned but Bob's is good as well)

The Quality Toolbox by Nancy R. Tague (the ASQ publication mentioned after the Kiviat diagram example)

The Scrum Guide (free PDF) by Ken Schwaber and Jeff Sutherland (just type "The Scrum Guide" in your browser to find the site (scrum.org) where you can download the latest version of the "definitive" definition of what Scrum is all about)

About the Author

Scott Duncan has been in the software field for some 47 years as a developer, technology transfer researcher, process consultant, trainer, and Agile coach. He has worked in fields such as book sales and distribution, state government, mainframe database and natural language query products, telecom (14+ years in Bell Labs, Bellcore, Telcordia), credit card transaction processing, and banking.

His most recent full-time role was as enterprise coach and trainer for an organization that developed software for the design, construction, and operation of power plants (including nuclear), processing plants (e.g., oil refineries, chemicals), and ships. There, he helped grow the organization to 144 Scrum teams in the US, India, Israel, UK, Germany, France, and Canada.

Along the way he was trained as an ISO 9001 Auditor and CMM assessor, was a member of ISO & IEEE Standards Committees for some 15 years, and was a member of the Scrum Alliance Board of Directors for 2 years.

Currently, he does training and coaching as an ICAgile certified trainer for Agile fundamentals, coaching, testing, business agility, and leadership as well as a variety of non-certified scrum master, product owner, manager, and team member courses.

Scott has a basic website at www.agilesoftwarequalities.com and can be reached at agileswqual@gmail.com.