martinfowler.com

The New Methodology

Martin Fowler

51-65 minutes

In the past few years there's been a blossoming of a new style of software methodology - referred to as agile methods. Alternatively characterized as an antidote to bureaucracy or a license to hack they've stirred up interest all over the software landscape. In this essay I explore the reasons for agile methods, focusing not so much on their weight but on their adaptive nature and their people-first orientation.

Probably the most noticeable change to software process thinking in the last few years has been the appearance of the word 'agile'. We talk of agile software methods, of how to introduce agility into a development team, or of how to resist the impending storm of agilists determined to change well-established practices.

This new movement grew out of the efforts of various people who dealt with software process in the 1990s, found them wanting, and looked for a new approach to software process. Most of the ideas were not new, indeed many people believed that much successful software had been built that way for a long time. There was, however, a view that these ideas had been stifled and not been treated seriously enough, particularly by people interested in software process.

This essay was originally part of this movement. I originally published it in July 2000. I wrote it, like most of my essays, as part of trying to understand the topic. At that time I'd used Extreme Programming for several years after I was lucky enough to work with Kent Beck, Ron Jeffries, Don Wells, and above all the rest of the Chrysler C3 team in 1996. I had since had conversations and read books from other people who had similar ideas about software process, but had not necessarily wanted to take the same path as Extreme Programming. So in the essay I wanted to explore what were the similarities and differences between these methodologies.

My conclusion then, which I still believe now, is that there were some fundamental principles that united these methodologies, and these principles were a notable contrast from the assumptions of the established methodologies.

This essay has continued to be one of the most popular essays on my website, which means I feel somewhat bidden to keep it up to date. In its original form the essay both explored these differences in principles and provided a survey of agile methods as I then understood them. Too much has happened with agile methods since for me to keep up with the survey part, although I do provide some links to continue your explorations. The differences in principles still remain, and this discussion I've kept.

From Nothing, to Monumental, to Agile

Most software development is a chaotic activity, often characterized by the phrase "code and fix". The software is written without much of an underlying plan, and the design of the system is cobbled together from many short term decisions. This actually works pretty well as the system is small, but as the system grows it becomes increasingly difficult to add new features to the system. Furthermore bugs become increasingly prevalent and increasingly difficult to fix. A typical sign of such a system is a long test phase after the system is "feature complete". Such a long test phase plays havoc with schedules as testing and debugging is impossible to schedule.

The original movement to try to change this introduced the notion of methodology. These methodologies impose a disciplined process upon software development with the aim of making software development more predictable and more efficient. They do this by developing a detailed process with a strong emphasis on planning inspired by other engineering disciplines - which is why I like to refer to them as **engineering methodologies** (another widely used term for them is **plan-driven methodologies**).

Engineering methodologies have been around for a long time. They've not been noticeable for being terribly successful. They are even less noted for being popular. The most frequent criticism of these methodologies is that they are bureaucratic. There's so much stuff to do to follow the methodology that the whole pace of development slows down.

Agile methodologies developed as a reaction to these methodologies. For many people the appeal of these agile methodologies is their reaction to the bureaucracy of the engineering methodologies. These new methods attempt a useful compromise between no process and too much process, providing just enough process to gain a reasonable payoff.

The result of all of this is that agile methods have some significant changes in emphasis from engineering methods. The most immediate difference is that they are less document-oriented, usually emphasizing a smaller amount of documentation for a given task. In many ways they are rather code-oriented: following a route that says that the key part of documentation is source code.

However I don't think this is the key point about agile methods. Lack of documentation is a symptom of two much deeper differences:

- *Agile methods are adaptive rather than predictive.* Engineering methods tend to try to plan out a large part of the software process in great detail for a long span of time, this works well until things change. So their nature is to resist change. The agile methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.
- *Agile methods are people-oriented rather than process-oriented.* The goal of engineering methods is to define a process that will work well whoever happens to be using it. Agile methods assert that no process will ever make up the skill of the development team, so the role of a process is to support the development team in their work.

In the following sections I'll explore these differences in more detail, so that you can understand what an adaptive and people-centered process is like, its benefits and drawbacks, and whether it's something you should use: either as a developer or customer of software.

Predictive versus Adaptive

Separation of Design and Construction

The usual inspiration for methodologies is engineering disciplines such as civil or mechanical engineering. Such disciplines put a lot of emphasis on planning before you build. Such engineers will work on a series of drawings that precisely indicate what needs to be built and how these things need to be put together. Many design decisions, such as how to deal with the load on a bridge, are made as the drawings are produced. The drawings are then handed over to a different group, often a different company, to be built. It's assumed that the construction process will follow the drawings. In practice the constructors will run into some problems, but these are usually small.

Since the drawings specify the pieces and how they need to be put together, they act as the foundation for a detailed construction plan. Such a plan can figure out the tasks that need to be done and what dependencies exist between these tasks. This allows for a reasonably predictable schedule and budget for construction. It also says in detail how the people doing the construction work should do their work. This allows the construction to be less skilled intellectually, although they are often very skilled manually.

So what we see here are two fundamentally different activities. *Design* which is difficult to predict and requires expensive and creative people, and *construction* which is easier to predict. Once we have the design, we can plan the construction. Once we have the plan for the construction, we can then deal with construction in a much more predictable way. In civil engineering construction is much bigger in both cost and time than design and planning.

So the approach for software engineering methodologies looks like this: we want a predictable schedule that can use people with lower skills. To do this we must separate design from construction. Therefore we need to figure out how to do the design for software so that the construction can be straightforward once the planning is done.

So what form does this plan take? For many, this is the role of design notations such as the UML. If we can make all the significant decisions using the UML, we can build a construction plan and then hand these designs off to coders as a construction activity.

But here lies the crucial question. Can you get a design that is capable of turning the coding into a predictable construction activity? And if so, is cost of doing this sufficiently small to make this approach worthwhile?

All of this brings a few questions to mind. The first is the matter of how difficult it is to get a UML-like design into a state that it can be handed over to programmers. The problem with a UML-like design is that it can look very good on paper, yet be seriously flawed when you actually have to program the thing. The models that civil engineers use are based on many years of practice that are enshrined in engineering codes. Furthermore the key issues, such as the way forces play in the design, are amenable to mathematical analysis. The only checking we can do of UML-like diagrams is peer review. While this is helpful it leads to errors in the design that are often only uncovered during coding and testing. Even skilled designers, such as I consider myself to be, are often surprised when we turn such a design into software.

Another issue is that of comparative cost. When you build a bridge, the cost of the design effort is about 10% of the job, with the rest being construction. In software the amount of time spent in coding is much, much less. McConnell suggests that for a large project, only 15% of the project is code and unit test, an almost perfect reversal of the bridge building ratios. Even if you lump in all testing as part of construction, then design is still 50% of the work. This raises an important question about the nature of design in software compared to its role in other branches of engineering.

These kinds of questions led Jack Reeves to suggest that in fact the source code is a design document and that the construction phase is actually the use of the compiler and linker. Indeed anything that you can treat as construction can and should be automated.

This thinking leads to some important conclusions:

- In software: construction is so cheap as to be free
- In software all the effort is design, and thus requires creative and talented people
- Creative processes are not easily planned, and so predictability may well be an impossible target.
- We should be very wary of the traditional engineering metaphor for building software. It's a different kind of activity and requires a different process

The Unpredictability of Requirements

There's a refrain I've heard on every problem project I've run into. The developers come to me and say "the problem with this project is that the requirements are always changing". The thing I find surprising about this situation is that anyone is surprised by it. In building business software requirements changes are the norm, the question is what we do about it.

One route is to treat changing requirements as the result of poor requirements engineering. The idea behind requirements engineering is to get a fully understood picture of the requirements before you begin building the software, get a customer sign-off to these requirements, and then set up procedures that limit requirements changes after the sign-off.

One problem with this is that just trying to understand the options for requirements is tough. It's even tougher because the development organization usually doesn't provide cost information on the requirements. You end up being in the situation where you may have some desire for a sun roof on your car, but the salesman can't tell you if it adds $10 to the cost of the car, or $10,000. Without much idea of the cost, how can you figure out whether you want to pay for that sunroof?

Estimation is hard for many reasons. Part of it is that software development is a design activity, and thus hard to plan and cost. Part of it is that the basic materials keep changing rapidly. Part of it is that so much depends on which individual people are involved, and individuals are hard to predict and quantify.

Software's intangible nature also cuts in. It's very difficult to see what value a software feature has until you use it for real. Only when you use an early version of some software do you really begin to understand what features are valuable and what parts are not.

This leads to the ironic point that people expect that requirements should be changeable. After all software is supposed to be *soft*. So not just are requirements changeable, they ought to be changeable. It takes a lot of energy to get customers of software to fix requirements. It's even worse if they've ever dabbled in software development themselves, because then they "know" that software is easy to change.

But even if you could settle all that and really could get an accurate and stable set of requirements you're probably still doomed. In today's economy the fundamental business forces are changing the value of software features too rapidly. What might be a good set of requirements now, is not a good set in six months time. Even if the customers can fix their requirements, the business world isn't going to stop for them. And many changes in the business world are completely unpredictable: anyone who says otherwise is either lying, or has already made a billion on stock market trading.

Everything else in software development depends on the requirements. If you cannot get stable requirements you cannot get a predictable plan.

Is Predictability Impossible?

In general, no. There are some software developments where predictability is possible. Organizations such as NASA's space shuttle software group are a prime example of where software development can be predictable. It requires a lot of ceremony, plenty of time, a large team, and stable requirements. There are projects out there that are space shuttles. However I don't think much business software fits into that category. For this you need a different kind of process.

One of the big dangers is to pretend that you can follow a predictable process when you can't. People who work on methodology are not very good at identifying boundary conditions: the places where the methodology passes from appropriate to inappropriate. Most methodologists want their methodologies to be usable by everyone, so they don't understand nor publicize their boundary conditions. This leads to people using a methodology in the wrong circumstances, such as using a predictable methodology in a unpredictable situation.

There's a strong temptation to do that. Predictability is a very desirable property. However if you believe you can be predictable when you can't, it leads to situations where people build a plan early on, then don't properly handle the situation where the plan falls apart. You see the plan and reality slowly drifting apart. For a long time you can pretend that the plan is still valid. But at some point the drift becomes too much and the plan falls apart. Usually the fall is painful.

So if you are in a situation that isn't predictable you can't use a predictive methodology. That's a hard blow. It means that many of the models for controlling projects, many of the models for the whole customer relationship, just aren't true any more. The benefits of predictability are so great, it's difficult to let them go. Like so many problems the hardest part is simply realizing that the problem exists.

However letting go of predictability doesn't mean you have to revert to uncontrollable chaos. Instead you need a process that can give you control over an unpredictability. That's what adaptivity is all about.

Controlling an Unpredictable Process - Iterations

So how do we control ourselves in an unpredictable world? The most important, and still difficult part is to know accurately where we are. We need an honest feedback mechanism which can accurately tell us what the situation is at frequent intervals.

The key to this feedback is iterative development. This is not a new idea. Iterative development has been around for a while under many names: incremental, evolutionary, staged, spiral... lots of names. The key to iterative development is to frequently produce working versions of the final system that have a subset of the required features. These working systems are short on functionality, but should otherwise be faithful to the demands of the final system. They should be fully integrated and as carefully tested as a final delivery.

The point of this is that there is nothing like a tested, integrated system for bringing a forceful dose of reality into any project. Documents can hide all sorts of flaws. Untested code can hide plenty of flaws. But when people actually sit in front of a system and work with it, then flaws become truly apparent: both in terms of bugs and in terms of misunderstood requirements.

Iterative development makes sense in predictable processes as well. But it is essential in adaptive processes because an adaptive process needs to be able to deal with changes in required features. This leads to a style of planning where long term plans are very fluid, and the only stable plans are short term plans that are made for a single iteration. Iterative development gives you a firm foundation in each iteration that you can base your later plans around.

A key question for this is how long an iteration should be. Different people give different answers. XP suggests iterations of one or two weeks. SCRUM suggests a length of a month. Crystal may stretch further. The tendency, however, is to make each iteration as short as you can get away with. This provides more frequent feedback, so you know where you are more often.

The Adaptive Customer

This kind of adaptive process requires a different kind of relationship with a customer than the ones that are often considered, particularly when development is done by a separate firm. When you hire a separate firm to do software development, most customers would prefer a fixed-price contract. Tell the developers what they want, ask for bids, accept a bid, and then the onus is on the development organization to build the software.

A fixed price contract requires stable requirements and hence a predictive process. Adaptive processes and unstable requirements imply you cannot work with the usual notion of fixed-price. Trying to fit a fixed price model to an adaptive process ends up in a very painful explosion. The nasty part of this explosion is that the customer gets hurt every bit as much as the software development company. After all the customer wouldn't be wanting some software unless their business needed it. If they don't get it their business suffers. So even if they pay the development company nothing, they still lose. Indeed they lose more than they would pay for the software (why would they pay for the software if the business value of that software were less?)

So there's dangers for both sides in signing the traditional fixed price contract in conditions where a predictive process cannot be used. This means that the customer has to work differently.

This doesn't mean that you can't fix a budget for software up-front. What it does mean is that you cannot fix time, price and scope. The usual agile approach is to fix time and price, and to allow the scope to vary in a controlled manner.

In an adaptive process the customer has much finer-grained control over the software development process. At every iteration they get both to check progress and to alter the direction of the software development. This leads to much closer relationship with the software developers, a true business partnership. This level of engagement is not for every customer organization, nor for every software developer; but it's essential to make an adaptive process work properly.

All this yields a number of advantages for the customer. For a start they get much more responsive software development. A usable, although minimal, system can go into production early on. The customer can then change its capabilities according to changes in the business, and also from learning from how the system is used in reality.

Every bit as important as this is greater visibility into the true state of the project. The problem with predictive processes is that project quality is measured by conformance to plan. This makes it difficult for people to signal when reality and the plan diverge. The common result is a big slip in the schedule late in the project. In an agile project there is a constant reworking of the plan with every iteration. If bad news is lurking it tends to come earlier, when there is still time to do something about it. Indeed this risk control is a key advantage of iterative development.

Agile methods take this further by keeping the iteration lengths small, but also by seeing these variations in a different way. Mary Poppendieck summed up this difference in viewpoint best for me with her phrase *"A late change in requirements is a competitive advantage"*. I think most people have noticed that it's very difficult for business people to really understand what they need from software in the beginning. Often we see that people learn during the process what elements are valuable and which ones aren't. Often the most valuable features aren't at all obvious until customer have had a chance to play with the software. Agile methods seek to take advantage of this, encouraging business people to learn about their needs as the system gets built, and to build the system in such a way that changes can be incorporated quickly.

All this has an important bearing what constitutes a successful project. A predictive project is often measured by how well it met its plan. A project that's on-time and on-cost is considered to be a success. This measurement is nonsense to an agile environment. For agilists the question is business value - did the customer get software that's more valuable to them than the cost put into it. A good predictive project will go according to plan, a good agile project will build something different and better than the original plan foresaw.

Putting People First

Executing an adaptive process is not easy. In particular it requires a very effective team of developers. The team needs to be effective both in the quality of the individuals, and in the way the team blends together. There's also an interesting synergy: not just does adaptivity require a strong team, most good developers prefer an adaptive process.

Plug-Compatible Programming Units

One of the aims of traditional methodologies is to develop a process where the people involved are replaceable parts. With such a process you can treat people as resources who are available in various types. You have an analyst, some coders, some testers, a manager. The individuals aren't so important, only the roles are important. That way if you plan a project it doesn't matter which analyst and which testers you get, just that you know how many you have so you know how the number of resources affects your plan.

But this raises a key question: are the people involved in software development replaceable parts? One of the key features of agile methods is that they reject this assumption.

Perhaps the most explicit rejection of people as resources is Alistair Cockburn. In his paper Characterizing People as Non-Linear, First-Order Components in Software Development, he makes the point that predictable processes require components that behave in a predictable way. However people are not predictable components. Furthermore his studies of software projects have led him to conclude the people are the most important factor in software development.

> In the title, [of his article] I refer to people as "components". That is how people are treated in the process / methodology design literature. The mistake in this approach is that "people" are highly variable and non-linear, with unique success and failure modes. Those factors are first-order, not negligible factors. Failure of process and methodology designers to account for them contributes to the sorts of unplanned project trajectories we so often see.
>
> -- [Cockburn non-linear]

One wonders if not the nature of software development works against us here. When we're programming a computer, we control an inherently predictable device. Since we're in this business because we are good at doing that, we are ideally suited to messing up when faced with human beings.

Although Cockburn is the most explicit in his people-centric view of software development, the notion of people first is a common theme with many thinkers in software. The problem, too often, is that methodology has been opposed to the notion of people as the first-order factor in project success.

This creates a strong positive feedback effect. If you expect all your developers to be plug-compatible programming units, you don't try to treat them as individuals. This lowers morale (and productivity). The good people look for a better place to be, and you end up with what you desire: plug-compatible programming units.

Deciding that people come first is a big decision, one that requires a lot of determination to push through. The notion of people as resources is deeply ingrained in business thinking, its roots going back to the impact of Frederick Taylor's Scientific Management approach. In running a factory, this Taylorist approach may make sense. But for the highly creative and professional work, which I believe software development to be, this does not hold. (And in fact modern manufacturing is also moving away from the Taylorist model.)

Programmers are Responsible Professionals

A key part of the Taylorist notion is that the people doing the work are not the people who can best figure out how best to do that work. In a factory this may be true for several reasons. Part of this is that many factory workers are not the most intelligent or creative people, in part this is because there is a tension between management and workers in that management makes more money when the workers make less.

Recent history increasingly shows us how untrue this is for software development. Increasingly bright and capable people are attracted to software development, attracted by both its glitz and by potentially large rewards. (Both of which tempted me away from electronic engineering.) Despite the downturn of the early 00's, there is still a great deal of talent and creativity in software development.

(There may well be a generational effect here. Some anecdotal evidence makes me wonder if more brighter people have ventured into software engineering in the last fifteen years or so. If so this would be a reason for why there is such a cult of youth in the computer business, like most cults there needs to be a grain of truth in it.)

When you want to hire and retain good people, you have to recognize that they are competent professionals. As such they are the best people to decide how to conduct their technical work. The Taylorist notion of a separate planning department that decides how to do things only works if the planners understand how to do the job better than those doing it. If you have bright, motivated people doing the job then this does not hold.

Managing a People Oriented Process

People orientation manifests itself in a number of different ways in agile processes. It leads to different effects, not all of them are consistent.

One of the key elements is that of accepting the process rather than the imposition of a process. Often software processes are imposed by management figures. As such they are often resisted, particularly when the management figures have had a significant amount of time away from active development. Accepting a process requires commitment, and as such needs the active involvement of all the team.

This ends up with the interesting result that only the developers themselves can choose to follow an adaptive process. This is particularly true for XP, which requires a lot of discipline to execute. Crystal considers itself as a less disciplined approach that's appropriate for a wider audience.

Another point is that the developers must be able to make *all* technical decisions. XP gets to the heart of this where in its planning process it states that only developers may make estimates on how much time it will take to do some work.

Such technical leadership is a big shift for many people in management positions. Such an approach requires a sharing of responsibility where developers and management have an equal place in the leadership of the project. Notice that I say *equal*. Management still plays a role, but recognizes the expertise of developers.

An important reason for this is the rate of change of technology in our industry. After a few years technical knowledge becomes obsolete. This half life of technical skills is without parallel in any other industry. Even technical people have to recognize that entering management means their technical skills will wither rapidly. Ex-developers need to recognize that their technical skills will rapidly disappear and they need to trust and rely on current developers.

The Difficulty of Measurement

If you have a process where the people who say how work should be done are different from the people who actually do it, the leaders need some way of measuring how effective the doers are. In Scientific Management there was a strong push to develop objective approaches to measuring the output of people.

This is particularly relevant to software because of the difficulty of applying measurement to software. Despite our best efforts we are unable to measure the most simple things about software, such as productivity. Without good measures for these things, any kind of external control is doomed.

Introducing measured management without good measures leads to its own problems. Robert Austin made an excellent discussion of this. He points out that when measuring performance you have to get *all* the important factors under measurement. Anything that's missing has the inevitable result that the doers will alter what they do to produce the best measures, even if that clearly reduces the true effectiveness of what they do. This measurement dysfunction is the Achilles heel of measurement-based management.

Austin's conclusion is that you have to choose between measurement-based management and delegatory management (where the doers decide how to do the work). Measurement-based management is best suited to repetitive simple work, with low knowledge requirements and easily measured outputs - exactly the opposite of software development.

The point of all this is that traditional methods have operated under the assumption that measurement-based management is the most efficient way of managing. The agile community recognizes that the characteristics of software development are such that measurement based management leads to very high levels of measurement dysfunction. It's actually more efficient to use a delegatory style of management, which is the kind of approach that is at the center of the agilist viewpoint.

The Role of Business Leadership

But the technical people cannot do the whole process themselves. They need guidance on the business needs. This leads to another important aspect of adaptive processes: they need very close contact with business expertise.

This goes beyond most projects' involvement of the business role. Agile teams cannot exist with occasional communication . They need continuous access to business expertise. Furthermore this access is not something that is handled at a management level, it is something that is present for every developer. Since developers are capable professionals in their own discipline, they need to be able to work as equals with other professionals in other disciplines.

A large part of this, of course, is due to the nature of adaptive development. Since the whole premise of adaptive development is that things change quickly, you need constant contact to advise everybody of the changes.

There is nothing more frustrating to a developer than seeing their hard work go to waste. So it's important to ensure that there is good quality business expertise that is both available to the developer and is of sufficient quality that the developer can trust them.

The Self-Adaptive Process

So far I've talked about adaptivity in the context of a project adapting its software frequently to meet the changing requirements of its customers. However there's another angle to adaptivity: that of the process changing over time. A project that begins using an adaptive process won't have the same process a year later. Over time, the team will find what works for them, and alter the process to fit.

The first part of self-adaptivity is regular reviews of the process. Usually you do these with every iteration. At the end of each iteration, have a short meeting and ask yourself the following questions (culled from Norm Kerth)

- What did we do well?
- What have we learned?
- What can we do better?
- What puzzles us?

These questions will lead you to ideas to change the process for the next iteration. In this way a process that starts off with problems can improve as the project goes on, adapting better to the team that uses it.

If self-adaptivity occurs within a project, it's even more marked across an organization. A consequence of self-adaptivity is that you should never expect to find a single corporate methodology. Instead each team should not just choose their own process, but should also actively tune their process as they proceed with the project. While both published processes and the experience of other projects can act as an inspiration and a baseline, the developers professional responsibility is to adapt the process to the task at hand.

Flavors of Agile Development

The term 'agile' refers to a philosophy of software development. Under this broad umbrella sits many more specific approaches such as Extreme Programming, Scrum, Lean Development, etc. Each of these more particular approaches has its own ideas, communities and leaders. Each community is a distinct group of its own but to be correctly called agile it should follow the same broad principles. Each community also borrows from ideas and techniques from each other. Many practitioners move between different communities spreading different ideas around - all in all it's a complicated but vibrant ecosystem.

So far I've given my take on the overall picture of my definition of agile. Now I want to introduce some of the different agile communities. I can only give a quick overview here, but I do include references so you can dig further if you like.

Since I'm about to start giving more references, this is a good point to point out some sources for general information on agile methods. The web-center is the Agile Alliance a non-profit set up to encourage and research agile software development. For books I'd suggest overviews by Alistair Cockburn and Jim Highsmith. Craig Larman's book on agile development contains a very useful history of iterative development. For more of my views on agile methods look at the appropriate sections of my articles and blog.

The following list is by no means complete. It reflects a personal selection of the flavors of agile that have most interested and influenced me over the last decade or so.

Agile Manifesto

The term 'agile' got hijacked for this activity in early 2001 when a bunch of people who had been heavily involved in this work got together to exchange ideas and came up with the Manifesto for Agile Software Development.

Prior to this workshop a number of different groups had been developing similar ideas about software development. Most, but by no means all, of this work had come out of the Object-Oriented software community that had long advocated iterative development approaches. This essay was originally written in 2000 to try to pull together these various threads. At that time there was no common name for these approaches, but the moniker 'lightweight' had grown up around them. Many of the people involved didn't feel this was a good term as it didn't accurately convey the essence of what these approaches were about.

There had been some talking about broader issues in these approaches in 2000 at a workshop hosted by Kent Beck in Oregon. Although this workshop was focused on Extreme Programming (the community that at that time had gained the most attention) several non XPers had attended. One the discussions that came up was whether it was better for XP to be a broad or concrete movement. Kent preferred a more focused cohesive community.

The workshop was organized, if I remember correctly, primarily by Jim Highsmith and Bob Martin. They contacted people who they felt were active in communities with these similar ideas and got seventeen of them together for the Snowbird workshop. The initial idea was just to get together and build better understanding of each others' approaches. Robert Martin was keen to get some statement, a manifesto that could be used to rally the industry behind these kinds of techniques. We also decided we wanted to choose a name to act as an umbrella name for the various approaches.

During the course of the workshop we decided to use 'agile' as the umbrella name, and came up with values part of the manifesto. The principles section was started at the workshop but mostly developed on a wiki afterwards.

The effort clearly struck a nerve, I think we were all very surprised at the degree of attention and appreciation the manifesto got. Although the manifesto is hardly a rigorous definition of agile, it does provide a focusing statement that helps concentrate the ideas. Shortly after we finished the manifesto Jim Highsmith and I wrote an article for SD Magazine that provided some commentary to the manifesto.

Later that year, most of the seventeen who wrote the manifesto got back together again, with quite a few others, at OOPSLA 2001. There was a suggestion that the manifesto authors should begin some on-going agile movement, but the authors agreed that they were just the people who happened to turn up for that workshop and produced that manifesto. There was no way that that group could claim leadership of the whole agile community. We had helped launch the ship and should let it go for whoever who wanted to sail in her to do so. So that was the end of the seventeen manifesto authors as an organized body.

One next step that did follow, with the active involvement of many of these authors, was the formation of the agile alliance. This group is a non-profit group intended to promote and research agile methods. Amongst other things it sponsors an annual conference in the US.

XP (Extreme Programming)

During the early popularity of agile methods in the late 1990's, Extreme Programming was the one that got the lion's share of attention. In many ways it still does.

The roots of XP lie in the Smalltalk community, and in particular the close collaboration of Kent Beck and Ward Cunningham in the late 1980's. Both of them refined their practices on numerous projects during the early 90's, extending their ideas of a software development approach that was both adaptive and people-oriented.

Kent continued to develop his ideas during consulting engagements, in particular the Chrysler C3 project, which has since become known as the creation project of extreme programming. He started using the term 'extreme programming' around 1997. (C3 also marked my initial contact with Extreme Programming and the beginning of my friendship with Kent.)

During the late 1990's word of Extreme Programming spread, initially through descriptions on newsgroups and Ward Cunningham's wiki, where Kent and Ron Jeffries (a colleague at C3) spent a lot of time explaining and debating the various ideas. Finally a number of books were published towards the end of the 90's and start of 00's that went into some detail explaining the various aspects of the approach. Most of these books took Kent Beck's white book as their foundation. Kent produced a second edition of the white book in 2004 which was a significant re-articulation of the approach.

XP begins with five values (Communication, Feedback, Simplicity, Courage, and Respect). It then elaborates these into fourteen principles and again into twenty-four practices. The idea is that practices are concrete things that a team can do day-to-day, while values are the fundamental knowledge and understanding that underpins the approach. Values without practices are hard to apply and can be applied in so many ways that it's hard to know where to start. Practices without values are rote activities without a purpose. Both values and practices are needed, but there's a big gap between them - the principles help bridge that gap. Many of XP's practices are old, tried and tested techniques, yet often forgotten by many, including most planned processes. As well as resurrecting these techniques, XP weaves them into a synergistic whole where each one is reinforced by the others and given purpose by the values.

One of the most striking, as well as initially appealing to me, is its strong emphasis on testing. While all processes mention testing, most do so with a pretty low emphasis. However XP puts testing at the foundation of development, with every programmer writing tests as they write their production code. The tests are integrated into a continuous integration and build process which yields a highly stable platform for future development. XP's approach here, often described under the heading of Test Driven Development (TDD) has been influential even in places that haven't adopted much else of XP.

There's a great deal of publications about extreme programming. One area of confusion, however, is the shift between the first and second edition of the white book. I said above that the second edition is a 're-articulation' of extreme programming, in that the approach is still the same but it is described in a different style. The first edition (with four values, twelve practices and some important but mostly-ignored principles) had a huge influence on the software industry and most descriptions of extreme programming were written based on the first edition's description. Keep that in mind as you read material on XP, particularly if it was prepared prior to 2005. Indeed most of the common web descriptions of XP are based on the first edition.

The natural starting place to discover more is the second edition of the white book. This book explains the background and practices of XP in a short (160 page) package. Kent Beck edited a multi-colored series of books on extreme programming around the turn of the century, if forced to pick one to suggest I'd go for the purple one, remember that like most material it's based on the first edition.

There's a lot of material on the web about XP but most of it is based on the first edition. One of the few descriptions I know of that takes account of the second edition is a paper on The New XP (PDF) by Michele Marchesi who hosted the original XP conferences in Sardinia. For discussion on XP there is a yahoo mailing list.

My involvement in the early days and friendships within the XP community mean that I have a distinct familiarity, fondness and bias towards XP. I think its influence owes to marrying the principles of agile development with a solid set of techniques for actually carrying them out. Much of the early writings on agile neglected the latter, raising questions about whether the agile ideas were really possible. XP provided the tools by which the hopes of agility could be realized.

Scrum

Scrum also developed in the 80's and 90's primarily with OO development circles as a highly iterative development methodology. It's most well known developers were Ken Schwaber, Jeff Sutherland, and Mike Beedle.

Scrum concentrates on the management aspects of software development, dividing development into thirty day iterations (called 'sprints') and applying closer monitoring and control with daily scrum meetings. It places much less emphasis on engineering practices and many people combine its project management approach with extreme programming's engineering practices. (XP's management practices aren't really very different.)

Ken Schwaber is one of the most active proponents of Scrum, his website is a good place to start looking for more information and his book is probably the best first reference.

Crystal

Alistair Cockburn has long been one of the principal voices in the agile community. He developed the Crystal family of software development methods as a group of approaches tailored to different size teams. Crystal is seen as a family because Alistair believes that different approaches are required as teams vary in size and the criticality of errors changes.

Despite their variations all crystal approaches share common features. All crystal methods have three priorities: safety (in project outcome), efficiency, habitability (developers can live with crystal). They also share common properties, of which the most important three are: Frequent Delivery, Reflective Improvement, and Close Communication.

The habitability priority is an important part of the crystal mind-set. Alistair's quest (as I see it) is looking for what is the least amount of process you can do and still succeed with an underlying assumption of low-discipline that is inevitable with humans. As a result Alistair sees Crystal as requiring less discipline than extreme programming, trading off less efficiency for a greater habitability and reduced chances of failure.

Despite Crystal's outline, there isn't a comprehensive description of all its manifestations. The most well described is Crystal Clear, which has a modern book description. There is also a wiki for further material and discussion of Crystal.

Context Driven Testing

From the beginning it's been software developers who have been driving the agile community. However many other people are involved in software development and are affected by this new movement. One obvious such group is testers, who often live in a world very much contained by waterfall thinking. With common guidelines that state that the role of testing is to ensure conformance of software to up-front written specifications, the role of testers in an agile world is far from clear.

As it turns out, several people in the testing community have been questioning much of mainstream testing thinking for quite a while. This has led to a group known as context-driven testing. The best description of this is the book Lessons Learned in Software Testing. This community is also very active on the web, take a look at sites hosted by Brian Marick (one of the authors of the agile manifesto), Brett Pettichord, James Bach, and Cem Kaner.

Lean Development

I remember a few years ago giving a talk about agile methods at the Software Development conference and talking to an eager woman about parallels between the agile ideas and lean movement in manufacturing. Mary Poppendieck (and husband Tom) have gone on to be active supporters of the agile community, in particular looking at the overlaps and inspirations between lean production and software development.

The lean movement in manufacturing was pioneered by Taiichi Ohno at Toyota and is often known as the Toyota Production System. Lean production was an inspiration to many of the early agilists - the Poppendiecks are most notable to describing how these ideas interact. In general I'm very wary of these kinds of reasoning by analogy, indeed the engineering separation between design and construction got us into this mess in the first place. However analogies can lead to good ideas and I think the lean ideas have introduced many useful ideas and tools into the agile movement.

The Poppendiecks' book and website are the obvious starting points for more information.

(Rational) Unified Process

Another well-known process to have come out of the object-oriented community is the Rational Unified Process (sometimes just referred to as the Unified Process). The original idea was that like the UML unified modeling languages the UP could unify software processes. Since RUP appeared about the same time as the agile methods, there's a lot of discussion about whether the two are compatible.

RUP is a very large collection of practices and is really a process *framework* rather than a process. Rather than give a single process for software development it seeks to provide a common set of practices for teams to choose from for an individual project. As a result a team's first step using RUP should be to define their individual process, or as RUP calls it, a *development case*.

The key common aspects of RUP is that it is Use Case Driven (development is driven through user-visible features), iterative, and architecture centric (there's a priority to building a architecture early on that will last the project through).

My experience with RUP is that its problem is its infinite variability. I've seen descriptions of RUP usage that range from rigid waterfall with 'analysis iterations' to picture perfect agile. It's struck me that the desire of people to market the RUP as the single process led to a result where people can do just about anything and call it RUP - resulting in RUP being a meaningless phrase.

Despite all this there are some very strong people in the RUP community that are very much aligned with agile thinking. I've been impressed in all my meeting with Phillippe Kruchten and his book is best starting point for RUP. Craig Larman has also developed descriptions of working with RUP in an agile style in his popular introductory book on OO design.

Should you go agile?

Using an agile method is not for everyone. There are a number of things to bear in mind if you decide to follow this path. However I certainly believe that these methodologies are widely applicable and should be used by more people than currently consider them.

In today's environment, the most common methodology is code and fix. Applying more discipline than chaos will almost certainly help, and the agile approach has the advantage that it is much less of a step than using a heavyweight method. Here the light weight of agile methods is an advantage. Simpler processes are more likely to be followed when you are used to no process at all.

For someone new to agile methods, the question is where to start. As with any new technology or process, you need to make your own evaluation of it. This allows you to see how it fits into your environment. As a result much of my advice here follows that I've given for other new approaches, bringing back memories of when I was first talking about Object-Oriented techniques.

The first step is to find suitable projects to try agile methods out with. Since agile methods are so fundamentally people-oriented, it's essential that you start with a team that wants to try and work in an agile way. Not just is a reluctant team more difficult to work with, imposing agile methods on reluctant people is fundamentally at odds with the whole notion of agile development.

It's valuable to also have customers (those who need the software) who want to work in this kind of collaborative way. If customers don't collaborate, then you won't see the full advantages of an adaptive process. Having said that we've found on several occasions that we've worked with customers who didn't want to collaborate, but changed their mind over the first few months as they begun to understand the agile approach.

A lot of people claim that agile methods can't be used on large projects. We (ThoughtWorks) have had good success with agile projects with around 100 people and multiple continents. Despite this I would suggest picking something smaller to start with. Large projects are inherently more difficult anyway, so it's better to start learning on a project of a more manageable size.

Some people advise picking a project with little business impact to start with, that way if anything goes wrong then there's less damage. However an unimportant project often makes a poor test since nobody cares much about the outcome. I prefer to advise people to take a project that's a little bit more critical than you are comfortable with.

Perhaps the most important thing you can do is find someone more experienced in agile methods to help you learn. Whenever anyone does anything new they inevitably make mistakes. Find someone who has already made lots of mistakes so you can avoid making those yourself. Again this is something true for any new technology or technique, a good mentor is worth her weight in gold. Of course this advice is self serving since ThoughtWorks and many of my friends in the industry do mentoring on agile methods. That doesn't alter the fact that I strongly believe in the importance of finding a good mentor.

And once you've found a good mentor, follow their advice. It's very easy to second guess much of this and I've learned from experience that many techniques can't really be understood until you've made a reasonable attempt to try them out. One of the best examples I heard was a client of ours who decided to trial extreme programming for a couple of months. During that period they made it clear that they would do whatever the mentor said - even if they thought it was a bad idea. At the end of that trial period they would stop and decide if they wanted to carry on with any of the ideas or revert to the previous way of working. (In case you were wondering they decided to carry on with XP.)

One of the open questions about agile methods is where the boundary conditions lie. One of the problems with any new technique is that you aren't really aware of where the boundary conditions until you cross over them and fail. Agile methods are still too young to see enough action to get a sense of where the boundaries are. This is further compounded by the fact that it's so hard to decide what success and failure mean in software development, as well as too many varying factors to easily pin down the source of problems.

So where should you not use an agile method? I think it primarily comes down to the people. If the people involved aren't interested in the kind of intense collaboration that agile working requires, then it's going to be a big struggle to get them to work with it. In particular I think that this means you should never try to impose agile working on a team that doesn't want to try it.

There's been lots of experience with agile methods over the last ten years. At ThoughtWorks we always use an agile approach if our clients are willing, which most of the time they are. I (and we) continue to be big fans of this way of working.