Wuhan University Journal of Natural Sciences Vol. 6 No. 1-2 2001,012~024

Article ID: 1007-1202(2001)01-0012-13

The New Methodology

Martin Fowler

651 W Washington Blvd,Suite 500,Chicago,IL 60661, USA

Abstract: In the past few years there's been a rapidly growing interest in "lightweight" methodologies. 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 lightweight methods, focusing not so much on their weight but on their adaptive nature and their people-first orientation. I also give a summary and references to the processes in this school and consider the factors that should influence your choice of whether to go down this newly trodden path.

Key words: lightweight method; people-first orietation

CLC number: TP 311.5

1 From Nothing, to Heavy, to Light

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.

We've lived with this style of development for a long time, but we've also had an alternative for a long time: methodology. 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.

These 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. Hence they are often referred to as heavy methodologies, or to use Jim Highsmith's term: monumental methodologies.

As a reaction to these methodologies, a new group of methodologies have appeared in the last few years. Although there's no official name for them, they are often referred to as light methodologies - signaling a clear reaction to the heavyweight methodologies. For many people the appeal of these lightweight methodologies is their reaction to the bureaucracy of the heavy weight 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 lightweight methods have some significant changes in emphasis from heavyweight 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

Received date: 2000-12-10

Biography: Martin Fowler(1963-),male,Professor,research direcfion:software engineering. E-mail:fowler@acm.org.

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 lightweight methods. Lack of documentation is a symptom of two much deeper differences:

● Light methods are adaptive rather than predictive.

Heavy 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 light methods, however, welcome change. They try to be processes that adapt and thrive on change, even to the point of changing themselves.

● Light methods are people-oriented rather than process-oriented.

They explicitly make a point of trying to work with peoples' nature rather than against them and to emphasize that software development should be an enjoyable activity.

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, it's benefits and drawbacks, and whether it's something you should use: either as a developer or customer of software.

2 Predictive versus Adaptive

2.1 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 many 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 crucial questions. Can you get a design that is capable of turning the coding into a construction activity? And if so, is construction sufficiently larger in cost and time 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

2.2 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 made a billion on stock market trad-

ing.

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

2.3 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 in 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.

2.4 Controlling an Unpredictable Process

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 between one and three weeks. SCRUM suggests a length of a month. Crystal will 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.

2.5 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 a fixed price contract in conditions where a predictive process cannot be used. This means that the customer has to work differently.

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.

The key benefit for the customer is a 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.

3 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.

3.1 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 lightweight 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]

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, it's roots going back to the impact of Frederick Taylor's Scientific Management approach. In running a factory, this Taylorist approach makes 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.)

3.2 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.) Such schemes as stock options increasingly align the programmers interests with the company's.

(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 ten 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.

3.3 Managing a People Oriented Process

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

One of the key elements is that of accepting the process rather 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. This is where Crystal is such an effective complement as it aims at being minimally disciplined.

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 tech-

nical 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.

3.4 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. Lightweight 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.

4 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. To deepen the process of self-adaptivity I suggest teams do a more formal review and major project milestones following the project retrospective sessions outlined by Norm Kerth. These retrospectives involve a $2 \sim 3$ day offsite meeting and a trained facilitator. Not only do they provide learning for the team, they also provide learning for the whole 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.

This self-adaptivity is most marked in ASD and Crystal. XP's rigid rules seem to disallow it, but that is only a surface impression since XP does encourage people to tune the process. The main difference with XP is that its advocates suggest doing XP by the book for several iterations before adapting it. In addition reviews are neither emphasized, nor part of the process, although there are suggestions that reviews should be made one of the XP practices.

5 The Methodologies

Several methodologies fit under this lightweight banner. While all of them share many characteristics, there are also some significant differences. I can't highlight all the points in this brief survey, but at least I can point you to some places to look.

I also can't speak with significant experience

about most of these. I've done quite a lot of work based on XP, and seen RUP around in many guises, but with most of the others my knowledge is primarily the less adequate book knowledge.

5.1 XP (Extreme Programming)

Of all the lightweight methodologies, this is the one that has got the most attention. Partly this is because of the remarkable ability of the leaders of XP, in particular Kent Beck, to get attention. It's also because of the ability of Kent Beck to attract people to the approach, and to take a leading role in it. In some ways, however, the popularity of XP has become a problem, as it has rather crowded out the other methodologies and their valuable ideas.

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.

The crucial step from informal practice to a methodology occurred in the spring of 1996. Kent was asked to review the progress of a payroll project for Chrysler. The project was being carried out in Smalltalk by a contracting company, and was in trouble. Due to the low quality of the code base, Kent recommended throwing out the entire code base and starting from scratch his leadership. The result was the Chrysler C3 project (Chrysler Comprehensive Compensation) which since became the early flagship and training ground for XP.

The first phase of C3 went live in early 1997. The project continued since and ran into difficulties later, which resulted in the canceling of further development in 1999. As I write this, it still pays the original 10,000 salaried employees.

XP begins with four values: Communication, Feedback, Simplicity, and Courage. It then builds up to a dozen practices which XP projects should follow. Many of these 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.

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.

On this platform XP builds an evolutionary design process that relies on refactoring a simple base system with every iteration. All design is centered around the current iteration with no design done for anticipated future needs. The result is a design process that is disciplined, yet startling, combining discipline with adaptivity in a way that arguably makes it the most well developed of all the adaptive methodologies.

XP has developed a wide leadership, many of them springing from the seminal C3 project. As a result there's a lot of sources for more information. The best summary at the moment is written by an outsider, Jim Highsmith, whose own methodology I'll cover later. Kent Beck wrote Extreme Programming Explained the key manifesto of XP, which explains the rationale behind the methodology and enough of an explanation of it to tell folks if they are interested in pursuing it further.

Two further books have recently appeared. Three members of the C3 project: Ron Jeffries, Ann Anderson, and Chet Hendrickson wrote Extreme Programming Installed, an explanation of XP based on the C3 experience. Kent Beck and I wrote Planning Extreme Programming, which discusses how you do planning in this adaptive manner.

As well as books, there are a fair number of web resources. Much of the early advocacy and development of the XP ideas occurred on Ward Cunningham's wiki web collaborative writing environment. The wiki remains a fascinating place to discover, although its rambling nature does lead you into being sucked in. To find a more structured approach to XP, it's best to start with two sites from C3 alumni: Ron Jeffries's xProgramming.com and Don Wells's extremeProgramming.org. Bill Wake's xPlorations contains a slew of

useful papers. Robert Martin, the well known author on C++ and OO design has also joined the list of XP promoters. His company, ObjectMentor, has a number of papers on its web site, including an XP instance of RUP called dX. They also sponsor the xp discussion egroup.

5.2 Cockburn's Crystal Family

Alistair Cockburn has been working on methodology ever since he was tasked by IBM to write about methodology in the early 90's. His approach is unlike most methodologists, however. Instead of building on solely personal experience to build a theory of how things should be done, he supplements his direct experience with actively seeking to interview projects to see how they work. Furthermore he isn't afraid to alter his views based on his discoveries; all of which make him my favorite methodologist.

His book, Surviving Object-Oriented Projects, was his first piece of advice on running projects, and remains my number one book recommendation for running iterative projects.

Since that book he's explored light methods further, coming up with the Crystal family of methodologies. It's a family because he believes that different kinds of projects require different kinds of methodologies. He looks into this variation along two axes: the number of people in the project, and the consequences of errors. Each methodology fits into a different part of the grid, so a 40 person project that can lose discretionary money has a different methodology than a six person life-critical project.

The Crystals share a human orientation with XP, but this people-centeredness is done in a different way. Alistair considers that people find it hard to follow a disciplined process, thus rather than follow XP's high discipline, Alistair explores the least disciplined methodology that could still succeed, consciously trading off productivity for ease of execution. He thus considers that although Crystal is less productive than XP, more people will be able to follow it.

Alistair also puts a lot of weight in end of iteration reviews, thus encouraging the process to be self-improving. His assertion is that iterative development is there to find problems early, and then to enable people to correct them. This places more emphasis on people monitoring their process and tuning it as they develop.

5.3 Open Source

You may be surprised by this heading. After all open source is a style of software, not so much a process. However there is a definite way of doing things in the open source community, and much of their approach is as applicable to closed source projects as it is to open source. In particular their process is geared to physically distributed teams, which is important because most adaptive processes stress co-located teams.

Most open source projects have one or more maintainers. A maintainer is the only person who is allowed to commit a change into the source code repository. However people other than the maintainer may make changes to the code base. The key difference is that other folks need to send their change to the maintainer, who then reviews it and applies it to the code base. Usually these changes are made in the form of patch files which make this process easier. The maintainer thus is responsible for coordinating the patches and maintaining the design cohesion of the software.

Different projects handle the maintainer role in different ways. Some have one maintainer for the whole project, some divide into modules and have a maintainer per module, some rotate the maintainer, some have multiple maintainers on the same code, others have a combination of these ideas. Most open source folks are part time, so there is an issue on how well such a team coordinates for a full time project.

A particular feature of open source development is that debugging is highly parallelizable. So many people can be involved in debugging. When they find a bug they can send the patch to the maintainer. This is a good role for non-maintainers since most of the time is spent finding the bug. It's also good for folks without strong design skills.

The process for open-source isn't well written up as yet. The most famous paper is Eric Raymond's The Cathedral and the Bazar, which while an excellent description is also rather brief. Klaus Fogel's book on the CVS code repository also contains several good chapters on open-source process that would be interesting even to those who never want to do CVS update.

5.4 Highsmith's Adaptive Software Development

Jim Highsmith has spent many years working with predictive methodologies. He developed them, installed them, taught them, and has concluded that they are profoundly flawed: particularly for modern businesses.

His recent book focuses on the adaptive nature of new methodologies, with a particular emphasis on applying ideas that originate in the world of complex adaptive systems (commonly referred to as chaos theory.) It doesn't provide the kind of detailed practices like the XP work does, but it does provide the fundamental groundwork for why adaptive development is important and the consequences at the deeper organizational and management levels.

At the heart of ASD are three non-linear, overlapping phases: speculation, collaboration, and learning.

Highsmith views planning as a paradox in an adaptive environment, since outcomes are naturally unpredictable. In traditional planning, deviations from plans are mistakes that should be corrected. In an adaptive environment, however, deviations guide us towards the correct solution.

In this unpredictable environment you need people to collaborate in a rich way in order to deal with the uncertainty. Management attention is less about telling people what to do, and more about encouraging communication so that people can come up with creative answers themselves.

In predictive environments, learning is often discouraged. You lay out things in advance and then follow that design.

In an adaptive environment, learning challenges all stakeholders - developers and their customers - to examine their assumptions and to use the results of each development cycle to adapt the next.

—[Highsmith]

As such learning is a continuous and important feature, one that assumes that plans and designs must change as development proceeds.

The overriding, powerful, indivisible, predominant benefit of the Adaptive Development Life Cycle is that it forces us to confront the mental models that are at the root of our self-delusion. It forces us to more realistically estimate our ability.

—[Highsmith]

With this emphasis, Highsmith's work focuses directly on to foster the hard parts of adaptive development, in particular how to foster collaboration and learning within the project. As such his book helps provide ideas to foster these "soft" areas which makes a nice complement to the grounded practice based approaches such as XP, FDD, and SCRUM.

5.5 SCRUM

SCRUM has been around for a while in object-oriented circles, although I'll confess I'm not too au fait with its history or development. Again it focuses on the fact that defined and repeatable processes only work for tackling defined and repeatable problems with defined and repeatable people in defined and repeatable environments.

SCRUM divides a project into iterations (which they call sprints) of 30 days. Before you begin a sprint you define the functionality required for that sprint and then leave the team to deliver it. The point is to stabilize the requirements during the sprint.

However management does not disengage during the sprint. Every day the team holds a short (fifteen minute) meeting, called a scrum, where the team runs through what it will do in the next day. In particular they surface to the management blocks: impediments to progress that are getting in the way that management needs to resolve. They also report on what's been done so management gets a daily update of where the project is.

SCRUM literature focuses mainly on the iterative planning and tracking process. It's very close to the other lightweights in many respects and should work well with the coding practices from XP.

There isn't any book on SCRUM at the moment, but there are a number of web resources. Ken Schwaber hosts controlChaos. com which is probably the best overview of SCRUM. Jeff Sutherland has always has an active web site on object technology issues and includes a section on SCRUM. There's also a good overview of SCRUM practices in the PLoPD 4 book.

5.6 Coad's Feature Driven Development

Feature Driven Development (FDD) was developed by Jeff De Luca and long time OO guru Pe-

ter Coad. Like the other adaptive methodologies, it focuses on short iterations that deliver tangible functionality. In FDD's case the iterations are two weeks long.

FDD has five processes:

- Develop an Overall Model;
- Build a Features List;
- Plan by Feature;
- Design by Feature;
- Build by Feature.

The first three are done at the beginning of the project.

The last two are done within each iteration. Each process is broken down into tasks and is given verification criteria

The developers come in two kinds: class owners and chief programmers. The chief programmer are the most experienced developers. They are assigned features to build. However they don't build them alone. Instead the chief programmer identifies which classes are involved in implementing the feature and gathers their class owners together to form a feature team for developing that feature. The chief programmer acts as the coordinator, lead designer, and mentor while the class owners do much of the coding of the feature.

The main description of FDD is in Peter Coad et al's UML in Color book. His company, TogetherSoft, also does consulting and training on FDD.

5.7 DSDM (Dynamic System Development Method)

DSDM started in Britain in 1994 as a consortium of UK companies who wanted to build on the RAD and iterative development. Starting with 17 founders it now boasts over a thousand members and has grown outside its British roots. Being developed by a consortium, it has a different flavor to many of the other lightweight methods. It has a full time organization supporting it with manuals, training courses, accreditation programs, and the like. It also carries a price tag, which has limited my investigation of the methodology. However Jennifer Stapleton has written a book which gives an overview of the methodology.

Using the method begins with a feasibility and a business study. The feasibility study considers whether DSDM is appropriate to the project at hand. The business study is a short series of workshops to understand the business area where the development takes place. It also comes up with outline system architectures and project plan.

The rest of the process forms three interwoven cycles : the functional model cycle produces analysis documentation and prototypes, the design and build cycle engineers the system for operational use, and the implementation cycle handles the deployment to operational use.

DSDM has underlying principles that include active user interaction, frequent deliveries, empowered teams, testing throughout the cycle. Like other light methods they use short timeboxed cycles of between two and six weeks. There's an emphasis on high quality and adaptivity towards changing requirements.

I haven't seen much evidence of its use outside the UK, but DSDM is notable for having much of the infrastructure of more mature traditional methodologies, while following the principles of the light methods approach.

There does seem to be an issue as to whether question on whether it's materials encourage more of a process-orientation and more ceremony than 1 would like.

5.8 Is RUP a light method?

Whenever we start discussing methods in the OO arena, we inevitably come up with the role of the Rational Unified Process. The Unified Process was developed by Phillipe Kruchten, Ivar Jacobson and others at Rational as the process complement to the UML. RUP is a process framework and as such can accommodate a wide variety of processes. Indeed this is my main criticism of RUP - since it can be anything it ends up being nothing. I prefer a process that tells you what to do rather than provide endless options.

As a result of this process framework mentality, RUP can be used in a very traditional waterfall style or in an adaptive lightweight manner. So as a result you can use RUP as a lightweight process, or as a heavyweight process - it all depends on how you tailor it in your environment.

Craig Larman is a strong proponent of using the RUP in a lightweight manner. His excellent introductory book on OO development contains a process that's very much based on his light RUP thinking. His view is that much of the recent push

to lightweight methods is nothing more than accepting mainstream OO development that's been captured as RUP. One of the things that Craig does is spend the first two or three days of a month long iteration with the whole team using the UML to outline the design of the work to be done during the iteration. This is not a blueprint that can't be deviated from, but rather a sketch that gives people a perspective on how things can be done over the iteration.

Another tack at light RUP is Robert Martin's dX process. The dx process is a fully compliant instance of RUP, that just happens to be identical to XP (turn dX upside down to see the joke). dX is designed for folks that have to use RUP, but want to use XP. As such it is both XP and RUP and thus a good example of the lightweight use of RUP.

For me, one of the key things that needs to happen with RUP is that the leaders of RUP in the industry need to emphasize their approach to software development. More than once I have heard people using RUP who are using a waterfall style development process. Due to my contacts in the industry I know that Philippe Kruchtem and his team are firm believers in iterative development. Clarifying these priciples and encouraging lightweight instances of RUP such as Craig's and Robert's work will have an important effect.

5.9 Other Sources

There are a number of other papers and discussions about this theme of light methods. While these may not be full methodologies, they do offer insights into this growing field.

The Patterns Language of Programming conferences has often contained material that touches on this subject, if only because many of the folks interested in patterns are also interested in more adaptive and humane methods. A leading early paper was Jim Coplein's paper at PLoP1. Ward Cunningham's Episodes pattern language appeared in PLoP2. Jim Coplein now hosts the OrgPatterns site, a wiki which collects together patterns for organizational patterns.

Dirk Riehle sent a paper to XP2000 that compares the value systems of XP and Adaptive Software Development. The July edition of the Coad letter compares XP to FDD. The July edition of IEEE Software includes several articles on "process diversity" which touch on these methodologies.

6 Should you go light?

Using a light 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 the these new 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 lightweight approach has the advantage that it is much less of a step than using a heavyweight method. Here much of the advantage of the lightweight methods is indeed their weight. Simpler processes are more likely to be followed when you are used to no process at all.

One of the biggest limitations to these new methodologies is how they handle larger teams. Crystal has been used up to about fifty people, but beyond that size there is little evidence as to how you can use an adaptive approach, or even if such approaches work at all.

Hopefully one message that's clear from this article is that adaptive approaches are good when your requirements are uncertain or volatile. If you don't have stable requirements, then you aren't in the position to have a stable design and follow a planned process. In these situations an adaptive process may be less comfortable, but it will be more effective. Often the biggest barrier here is the customer. In my view it's important for the customer to understand that following a predictive process when requirements change is risky to them just as much as it is to development.

So you'll notice I've said that if you have more than fifty people you should use a traditional predictive process and if you change changing requirements you should use an adaptive process. What if you have both a large project and changing requirements? I don't have a good answer to this, so I'd suggest you seek a second opinion. I can tell you that things will be very difficult, but I suspect you already know that.

If you are going to take the adaptive route, you need to trust your developers and involve them in the decision. Adaptive processes rely on you trusting your developers, so if you consider your developers to be of low quality and motivation then you should use a predictive approach.

So to summarize. The following factors suggest an adaptive process:

- Uncertain or volatile requirements;
- Responsible and motivated developers;
- Customer who understands and will get involved.

These factors suggest a predictive process:

- A team of over fifty.

Fixed price, or more correctly a fixed scope, contract.

7 Which Adaptive Process?

All of these processes are new, and I can only give some first hand experience to guide you. In my choice the question lies in the team size and how much discipline they are prepared to go for.

With a dozen developers or less that are inclined to try it, I'd certainly push for XP. It may be that the team won't go all out in the XP process, at least initially, but you still get a lot of benefit by a partial XP approach. For me the key test of using this process well is automated unit testing. If the team are prepared to do that, then the technical foundations will be in place. If they can't do that, then I don't suspect they'll handle the rest.

If the discipline isn't there, or the team is too large, then I'd be inclined to follow Crystal's advice. It's certainly the lightest of the light, and Cockburn is particularly adaptive to the lessons of development. I'd still use the XP planning process in these cases.

Having said that, however, I'm working with a team of forty who are successfully trying many XP practices and are pretty close to full XP, so with determination and a committed team you can adapt your process outside at least some of these boundaries.

And that's really the key message. Whatever process you start with will not be the process that'll really work for you. You have to take charge of your process, monitor it and adapt it to your circumstances. In the end it must become your process, any other labels are secondary.