Chapter 1

What is an agile methodology?

Summary: Rapid business change requires rapid software development. How can we react to changing needs during software development? How can we ensure quality (correctness) as well as fitness for purpose? What are the requirements that an agile process should meet? What are the problems and limitations of agile processes?

1. Rapid business change - the ultimate driver.

It has often been said that the modern world is experiencing unprecedented levels of change, in technology, in business, in social structures and in human attitudes. Of course, this is a complex and poorly understood phenomena but I know of no sources that disagree with the basic axiom that the world is changing fast and that fact is not, itself, about to change. Some may prefer that the world is not like that and others may believe that this phenomena is unsustainable in the long term - the world will simply run out of resources or collapse into social anarchy and destruction.

At the present time, however, dynamic change is a key factor of both business and public life. The other key truism is that computer technology and software in particular, is a vital component of these organisations. It is clear then, that the developers of this software have a problem, the pressure to develop new software support for rapidly changing processes is causing serious problems for the software industry. Traditional software engineering cannot deliver what is required at a cost and within the time scale that is required.

As we have seen in the previous chapter, this is caused by some structural and attitudinal problems associated with traditional software engineering. Deep thinkers about this problem have come up with a number of, what may seem to be paradoxical, insights into the problems.

Firstly they recognise that everything about our current software processes must change. On the other hand the proposed solutions involve a number of well tried and trusted techniques that have been around for years. It is not just a matter of shuffling around a few old favourite techniques into a different order, rather it is a new combination of activities which are grounded in a new and very positive philosophy of *agile* software development.

2. What must agile methods be able to do?

We have seen that any agile process has to be able to adapt to rapid changes in scope and requirements, but it has also to satisfy the needs for the delivery of high quality systems in a manner which is highly cost-effective, unburdened by massive bureaucracy and which do not demand heroics from the developers involved. So we will try to specify the basic properties that a successful agile software development process must satisfy.

The first requirement is the ability to adapt the development of the software as the client’s problem changes.

The second property derives from the need to allow for the future evolution of any delivered solution.

The third issue is that of software quality, how do we know that the software always does what it is supposed to do?

The fourth consideration is the amount of unnecessary documentation and other bureaucracy that is required to sustain and manage the development process.

The fifth issue is the human one which relates both to the experiences of the developers in the development process but also the way in which the human resources are managed.

We will look at all of these in turn.

3. Agility - what is it and how do we achieve it?

When we embark on a software development project the initial and some would say the hardest phase is that of determining the requirements, finding out, with the client wants, what the proposed system is supposed to do.

It might start with a brief overview of the business context and the identification of:

the sort of data that is to be involved;
how this data is to be manipulated and
how these various activities mesh together with each other and with the other activities in the business.

Many techniques exist to do this, ways of collecting information, not just from the client but also from the intended users of the system will be needed in this initial stage. Sifting through this information, making decisions about the relative importance of some of the information and trying to set it into a coherent picture follows. Again a number of different approaches, notations and techniques exist to support this.

Having achieved some indication of the overall purpose of the system and the way that it interfaces and interacts with other business process will be the next issue. We are trying to establish the system boundary during this phase.

From this we construct a detailed requirements document. Some examples of actual documents will be given in a later chapter. Such a document will be structured, typically, into functional requirements and non-functional requirements. Both are vitally important. Each requirement will be stated in English, perhaps structured into sections containing related requirements and described at various levels of detail. The client may well be satisfied at this point with what is proposed. However, it is always difficult to visualise exactly how the system will work at this stage and it may not be right.

Now we would embark on some analysis, looking at these key operational aspects, identifying the sort of computing resources needed to operate such a system and to consider many other aspects of the proposed system. Following analysis we get into the design phase and it is here where we describe the data and processing models and how the system could be created from the available technical options.

This stage is often lengthy and complicated. Rarely will the developers be able to proceed independently of the client. There will be many issues that will arise during this process which should require further consultation with the client. This is often not carried out and the developers start making decisions that only the client should take. We see the system starting to drift from what it should be.

At the end of this process we will have a large and complicated detailed design which may or may not still be valid in terms of the client’s business needs.

If we were to go back to the client at this stage we may very well find that the business has moved on and the requirements have changed significantly. The traditional development methods cannot handle this challenge effectively because of the amount of investment that has been made into the design there will be a reluctance to change it significantly or to start again.

The first two key issues are, therefore:

an approach which retains a continual and close relationship with the client
and
an approach to development that does not involve the heavy overhead of a long and complex design phase.

If this is achieved then the development process might be able to adapt to the changing requirements more.

4. Evolving software - obstacles and possibilities.

Even if we are able to deliver a solution that is still relevant it may not remain so for long. Things are bound to change and there is thus a need to see how we can evolve the software towards it new requirements. Some of the old functionality is likely to remain, however, so it would be inefficient to throw it away and start again. How can we develop a method whereby changes can be achieved in the software quickly, cheaply and reliably?

Many systems will involve a database somewhere and this is one of the key issues when it comes to obstacles to evolution. traditionally we use a relational database structure and a relational database management system to manage it. Much time is spent building and normalising the data model. When circumstances change, however, the data model may not be appropriate still. What can we do about this? It may not be a straightforward matter to re-engineer this data model. It may not be possible; to just insert a couple of new fields or a new table or two. It is likely that the whole data model will have to be substantially re-engineered and this could expensive.

Object-oriented databases where claimed by some to be a solution but we are still waiting for a convincing demonstration of this.

A more promising development is the use of XML as a foundation for a database. There is good evidence [Medcalf 01] that an XML database is much easier to evolve than other forms.

It is also possible to query such a database in the usual way and many interesting new developments involving XML are coming through. It is certainly worth looking at this option.

The use of XML databases is not a critical part of the agile methodology movement but it is an important issue, nevertheless and one that may well become much more important in the future.

What are the requirements of a software engineer when faced with the problem of adapting an existing system to deal with some new requirements?

The first thing is a clear understanding of what the existing system actually does. This can be achieved, to a certain extent, by running the software and observing its behaviour. A complete knowledge, however, will only be achieved by looking at the design - or will it. The design may not be reliable and so we have to look at the source code. If this is written in a clear and simple fashion then it will be possible to establish a lot about it. If we could this with a clear

structure to the requirements document we may have a chance of understanding things.

If the original system was built in stages, gradually introducing new functionality in a controlled manner, we will be able to see where features that are no longer needed were introduced and we can explore how we might evolve the software gradually by introducing, in stages, any new functionality and removing some of the old. Throughout, we need to consult the client.

Thus we need:

the system to be built in such a way that the relationship between the requirements and the code is clear

and

the code itself is clear and understandable.

If we can introduce a more appropriate database technology, such as XML, [StLaur99], then all the better.

5. The quality agenda.

Quality software is a key issue although one that the industry has great difficulty in addressing successful.
For real quality systems we have to address two vital issues:

identifying the right software to be built

and

demonstrating that this has been achieved.

Neither tasks are easy. The first task is made more difficult by the possible changing nature of the business need and the consequential requirement to adapt to a changing target. This is one of the key objectives of an agile methodology. However, it might be possible to find a way of adapting and altering the software being built to reflect the developers' changing understanding of the client's needs but it is quite another to be sure that they have got the changes right. Here is where a strong relationship between the developers and the business they are trying to develop a system for is needed. It also requires a considerable amount of discussion and review both between the developers and the client and amongst the developers but also amongst the clients, they really do have to know where their company is going.

Hence an agile methodology must be able to deal with identifying and maintaining a clear and *correct* understanding of the system being built. By correct we mean something that is acceptable to the client, a system that has the correct functional and non-functional attributes as well as being within budget and time.

To satisfy such requirements the agile methodology must provide support, not only for changing business needs but also for giving assurance that these are indeed the real requirements. In order to do this there has to be a continuous process of discussion, question asking and resolution based on clear and practical objectives.

The second quality issue is that of ensuring that the delivered system meets its requirements. Here there are serious problems with almost all approaches. Despite the best intentions of many, testing and review are aspects of software engineering that are either done inadequately or too late to be effective.

In most development projects that are not completely chaotic some attempt is made to carry out reviews of the work done. This might be the review of requirements documents, designs or

code and should involve a number of people examining the documentation and code provided by the developers and inspecting it for flaws of various types. The developers then have to address any concerns raised by the review. Human nature, being what it is, developers are often reluctant to accept other people's opinions. In many cases where serious problems have been found the developers will try to adjust and work round the problems rather than carry out significant reworking. In fact one often sees the situation where the best solution is to start again with a component but the resistance to doing this is often profound. This just compounds the problems and is very hard to overcome. If a developer has spent a week or longer on some component which is seriously flawed then they are likely to resent having to start again. This is a potentially serious quality problem.

An agile methodology, therefore, needs to address this issue of review and to provide a mechanism that will not result in this sort of situation.

Another quality aspect is the correctness of the final code. This is usually addressed by testing whereby the software is run against suitable test sets and its behaviour monitored to establish whether it is behaving in the required manner. For this to work we need two basic things, we need to know what the software is supposed to do and we need to be able to create test sets that will give us enough confidence that the code does do what it is supposed to do.

The role of testing in the design and construction of software is a misunderstood and underdeveloped activity. An effective agile methodology must provide a clear link between the identification of what the systems is supposed to do and the creation and maintenance of effective test sets. Furthermore the testing must be fully integrated into the construction process so that we avoid the massive problems, and expense, that arise when the testing is done last.

It also allows us to introduce key *design for test* considerations driven by the realisation that the way the system is constructed will affect the ease and effectiveness of the testing. Some systems are almost impossible to test properly because of the way that they have been built. This is a well known insight in hardware design (microprocessors etc.) but not something that seems to exercise software engineering much.

6. Do we really need all of this mountain of documentation?

Most non-chaotic software development methods are *design led* and *document driven*. We need to examine the purpose of all this paperwork (it might be stored electronically but it still amounts to masses of text, diagrams and arcane notations).

Let's look first at the issue of design, what is it for and where does it fit in a development project.

Design is a mechanism for exploring and documenting possible solutions in a way that should make the eventual translation into working software easy and trouble free. If there is an analysis phase, then typically this will establish the overall parameters of the project and will result in a, usually fixed, set of requirements and constraints for the project. The design phase then takes this information and develops a more concrete representation of the system in a form that is suitable as a basis for programming.

Firstly, the issue of agility means that the analysis phase is likely to be continuous throughout most of the project if it is to be able to adapt to changing business need. If an agile approach is to work the nature and role of analysis must change. Therefore the role of design will also be an issue. How can we deal with the rapid changes that analysis might throw up if the design is proceeding by way of a large and complex process which is trying to identify, at a significant

level of detail, issues that will eventually be the responsibility of programmers to solve. Large complex designs are almost impossible to maintain in this context. Some tool vendors will emphasise the benefit of using Computer Aided Software Engineering (CASE) and other tools which might provide support for the maintenance of the design but many programmers dislike these systems, they are often imposed by the management, who don't have to use them, and programmersfeel that their creativity is compromised.

Creative programmers will also be tempted to solve problems that arise during implementation that were not predicted by the analysis or design phases without updating the design archive. This is a real problem in many projects which may only come to light during maintenance when it is discovered that the design differs from what the system actually is. In other words the code does not work as the design documents indicate in some, possibly crucial areas. Thus maintenance is carried out by reference to the code which is the key resource and the design may not be used or trusted.

So why is it there? The design is a resource that has cost time and money to create and yet it may not seem to provide any reliable value. It might actually damage projects because of the difficulties of ensuring that the design can evolve as the business needs change. It is possible that the existence of a large and complex design may encourage developers to resist changes to the system asked for by the client. If this happens, and I believe that it does a lot, then the client is not going to get the system they want. A standard technique is to tell the client that it would be too expensive to change things and this often works but it is a short term solution. The client is going to be less than satisfied at the end. An agile process needs to be able to deal with this issue.

So, is design a key part of an agile process? It has to be made much more responsive to a project's changing needs and it hsould also provide a precise description of the final code, otherwise it is merely of historical value. Design notations can help us to clarify and discuss our ideas and from that point of view they are useful. Bearing in mind that design documents might be misinterpreted by people who were not involved in the development, for example those carrying out maintainance in subsequent years, we should not place all our reliance on them. We will also need to document the code carefully and also the test sets, these will help a great deal in understanding what the software does when all the original team has dispersed.

Another aspect that relates also to the human dimension is that creative people, and good programmers are creative, do not work to their best ability if they feel dominated by bureaucratic processes and large amounts of seemingly irrelevant documentation. It's a natural feeling and applies in all walks of life. If you feel that churning out lots of unnecessary paperwork gets in the way of your ultimate desire - building a quality system to satisfy your client - then you may not put your best effort into creating all this stuff. Good morale, as we shall see next, is vital for good productivity. If we don't need it then bin it!

7. The human factor.

People are individuals with their own desires, values and capabilities. Software engineering is a people based business and the morale of the teams is a vital component in the success of the project. Too often, organisations organise themselves in hierarchical structures whereby those who are above you feed down instructions perhaps without any serious explanation and those below you suffer from you doing the same. It is often difficult to feel valued and to know what is really going on - as opposed to what the managers think is going on.
To get the best out of people we have to consider them as intelligent and responsible individuals and to show interest in their views and an awareness of their objectives. This calls on skilled and sensitive management. This does not mean that the management system abdicates all responsibility and we are left with a chaotic approach where everyone just does their own

thing.

What is needed is a system that focusses on the key issues, involves everyone to the greatest possible extent, jointly identifies the constraints and parameters applicable to the project and provides an open mechanism for discussion, decision making and the taking of responsibility. Over many years of supervising and managing projects I am convinced that this is the most effective way. It is not without problems, there will always be problems, sometimes individuals are just unreasonable and threaten the joint endeavours of the team. In my experience the team, if given the responsibility, will deal with the issues effectively. In the few instances where I have had to intervene the solution has been negotiated quickly and effectively. There are a number of management devices that can work - yellow cards and red cards as used in football (soccer) may be useful, the use of a *sin bin* might also be considered for unreasonable colleagues. It has to be a group decision rather than the manager's to be most effective, however.

Agility requires co-operation from the development teams, they need to be able to adapt to changing circumstances without feeling threatened or pressurised. A flat and inclusive management structure seems to be able to deliver this.

We shouldn't forget the needs of the clients. They are the other people in the loop and one way to ensure that they are kept happy is to keep them informed and to have excellent lines of communication between the development team and the clients and users. Clients also worry about progress since they may be held responsible for project failure or other consequences caused by problems beyond their control. Many clients are sceptical about the reassurances given to them by developers using traditional approaches to software engineering where the only things to show for months of work are incomprehensible diagrams and paperwork. By providing pieces of functioning software, allbeit prototypes in some methodologies, at least provides some confidence that things are progressing. It also provides a mechanism for feedback from a real physical implementation rather than from vague abstractions.

8. Conclusions.

The issues that an agile approach to software engineering must address can be summarised in the following five properties:

* the ability to adapt the development of the software as the client's problem changes and to provide feedback on progress;

* the need to allow for the future evolution of any delivered solution;

* the quality of the delivered software must be assured;

* the amount of documentation and other bureaucracy that is required to sustain and manage the development process should be minimal;

* the human dimension must be a key aspect both for the developers and the clients.

References.

[StLaur99] S. St. Laurent & E. Ceramie, *Building XML applications*, McGraw Hill, 1999.
[Medcalf 2001] A. Medcalf, "Evolving databases", 3rd year undergraduate report, University of Sheffield, 2001, available on line at: http://www.dcs.shef.ac.uk/****