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?

1.1 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 phenomenon, 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 not be like that, and others may believe that this phenomenon 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, rapid change is a key factor of both business and public life. The other important truism is that computer technology, and software in particular, is a vital component of many businesses and organizations. 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 has repeatedly failed to deliver what is needed at the cost and within the timescale that are required.

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. Key texts such as Pressman (2001) and Sommerville (2006) present a broad survey of traditional software engineering that documents many of the current approaches. Other thinkers such as

Running an Agile Software Development Project. By Mike Holcombe

Gilb (1988) and more recently Beck (1999) are beginning to question the way in which software engineering has been carried out.

Thinkers such as Beck recognize that everything about our current software processes must change. On the other hand, their proposed solutions partly 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 favorite techniques into a different order; rather, it is a new combination of activities that are grounded in a new and very positive philosophy of *agile* software development.

1.2 WHAT MUST AGILE METHODOLOGIES BE ABLE TO DO?

We note that any agile software development 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 that is highly cost-effective, unburdened by massive bureaucracy, and that does not demand heroics from the developers involved. Thus, we will try to specify the basic properties that a successful agile software development process must satisfy.

1. The first issue is the ability to adapt the development of the software as the client's problem changes.
2. The second issue derives from the need to allow for the future evolution of any delivered solution.
3. The third issue is that of software quality: How do we know that the software always does what it is supposed to do?
4. The fourth issue is the amount of unnecessary documentation and other bureaucracy that is required to sustain and manage the development process.
5. The fifth issue is the human one, which relates both to the experiences of the developers in the development process and to the way in which the human resources are managed.

Coupled with these is a need to have a clear business focus for any software development project and application.

We will look at all of these in turn.

1.3 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, what the proposed system is supposed to do.

It might start with a brief overview of the business context and the identification of the kind 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 follow. Again, a number of different approaches, notations, and techniques exist to support this.

Having achieved some indication of the overall purpose of the system, the way that it interfaces and interacts with other business processes 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 visualize exactly how the system will work at this stage, and our understanding of 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 considering many other aspects of the proposed system. After 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 although there may be pressure on them from managers to do so. There will be many issues that will arise during this process requiring 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 that may or may not still be valid in terms of the client's business needs, which may be evolving.

If we 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, such as the waterfall method, cannot handle this challenge effectively. Because of the investment in the design, there may be a reluctance to change it significantly or to start again.

The waterfall model envisions a steady and systematic sequence of stages starting with the capture and definition of the requirements, the analysis of these requirements, the formalizing of a system and software design, the implementation of the design, and the testing of the software. Finally we have delivery and after-sales, which covers a number of different types of maintenance : perfective maintenance where faults are removed after delivery, adaptive maintenance, which might involve building more functionality in the system, and maintenance to upgrade the software to a different operating environment.

It will always be necessary, and sometimes possible, to backtrack around some of the stages, but the emphasis is on a trying to identify the requirements in one go. The diagram in Fig. 1.1 tries to illustrate the approach.

The need to respond more quickly to the changing nature of the customer's needs does not sit easily with this type of model.

The first two key issues are, therefore, to find an approach that retains a continual and close relationship with the client, and to find 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 more able to adapt to the changing requirements.

There are a number of other approaches to software development that have attempted to address these issues. The spiral model (Sommerville, 2006) describes this approach (Fig. 1.2). It involves a series of iterations around the *requirements capture or specification–implementation–testing or validation–delivery and operation* loop together with periodic reviews of the overall project and the analysis of risks that have been identified during the course of the project.

It attempts to recognize that for many projects, there is an ongoing relationship with the customer that does not end with the delivery of the system but will continue through many further stages involving correcting and extending or adapting the product. In these cases, there is no such thing as a *finished product*.

Rapid applications development and evolutionary delivery are similar sorts of approaches that are built around the idea of building and demonstrating, and in the latter case delivering, parts of the system as the project goes along.

Such approaches can be successful but differ in many ways from the approaches taken by the current agile or lightweight methodologies, one of which we are considering here. One issue is the length of an iteration cycle; in agile approaches, these are very short.

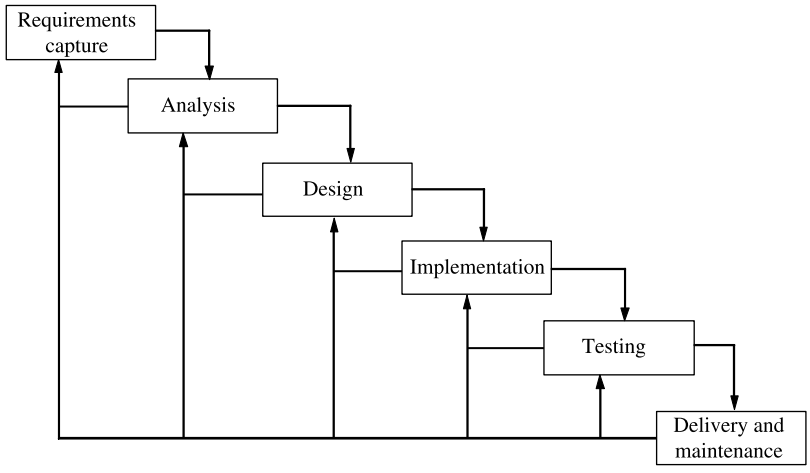

Figure 1.1 The waterfall model of software development.

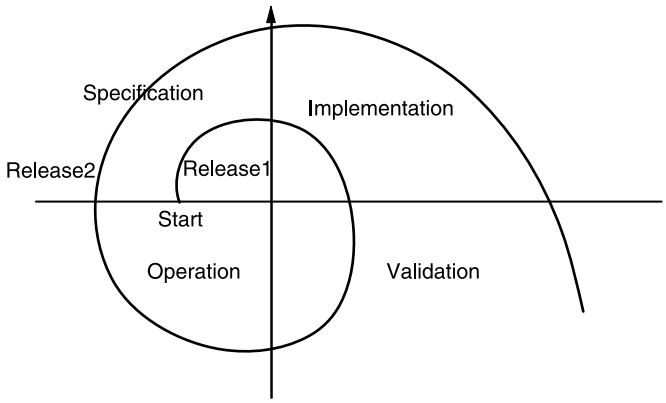

Figure 1.2 The spiral model.

There have been many analyses of failed software development projects. Failures in communication, both between developers and clients and between and among developers, seem to be some of the most common causes of problems. In the traditional approach, the various documents (requirements documents, design documents, etc.) are supposed to facilitate this communication; however, often the language and notation used in these documents fails to support effective communication. UML (Unified Modeling Language) diagrams, for example, can often be interpreted differently by different people.

1.4 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 toward its 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? Agile techniques are an attempt to answer some of these questions.

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 normalizing the data model. When circumstances change, however, the data model may not still be appropriate. What can we do about this? It may not be a straightforward matter to reengineer 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 reengineered, and this could be expensive.

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

In the case where there is an existing system that forms the basis of the development, the first thing to do is to gain a clear understanding of what the current system does. This can be achieved, to a certain extent, by running the software and observing its behavior. A complete knowledge, however, will only be achieved by looking at the design in some detail. 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 understand it well. If we could do 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 may 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.

For projects that require a completely new system to be built, then time needs to be spent on identifying the business processes that will be supported by the new system, along with information about how current manual processes operate, if there are any. The more that is known about the users and their needs, the better.

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 to be clear and understandable.

1.5 THE QUALITY AGENDA

The quality of software is a key issue for the industry although one that it has had great difficulty in addressing successfully.

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.

In terms of the types of faults that are often made in software development, we can identify two important types of faults: *requirements faults* (we tell the computer to do the wrong thing) and *operational faults* (the computer wrongly does the thing we told it to do).

Neither problem is easy to deal with. 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 among the developers but also among the client's staff; 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.

An approach to improving quality in a model like the waterfall model is called the *V model* (Fig. 1.3). Here each stage in the process provides the basis for testing of a particular type. We will discuss more about testing later. Some of the terms may seem unfamiliar at this stage; they are also not always distinct. However, the idea that, for example, the requirements could be used to define some of the acceptance tests, and so on, is a useful indication of what might be a practical approach to ensuring quality.

In most development projects that are not completely chaotic, some attempt is also 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, is such that 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 around 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 that is then found to be seriously flawed, they are

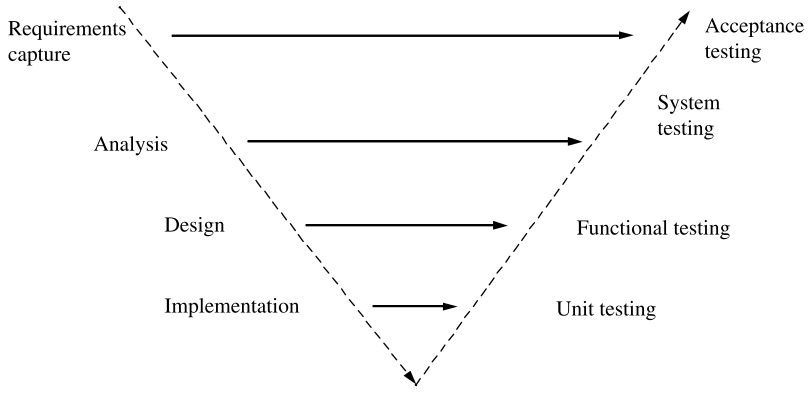

Figure 1.3 The V model.

likely to resent having to start again. This is a potentially serious quality and efficiency problem. In many companies, software developers spend most of their time developing their code on their own with little discussion with others, and when flaws in their output are found, a lot of time has been wasted.

An agile methodology, therefore, needs to address this issue of review and testing and to provide a mechanism that will provide confidence in the quality of the product.

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 behavior 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 system 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 realization 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 well-known in hardware design (microprocessors, etc.) but is not something that seems to exercise software engineering much.

There are many myths about software quality and what the position really is. Little empirical research has been carried out analyzing the quality of software systems. What has been done has often uncovered some uncomfortable news. In terms of the operational faults—leaving aside the problem of capturing the wrong requirements—the evidence is pretty gloomy.

Hatton (1998) in a series of studies discovered the following: Using *static* deep-flow analysis across many different industries and application areas, he measured the consistency of several million lines of software written in various languages: C, Fortran 77, C++, and so forth.

He also measured the level of *dynamic* disagreement between independent implementations of the same algorithms acting on the same input data with the same parameters in just one of these industrial application areas.

On average there were 8 serious faults per 1000 lines for C programs and 12 for Fortran; 10% of the C population would be deemed untestable by any standards.

Object-oriented languages were worse, not only in the density of faults but also in the time it took to correct them—findings also supported by the work of Humphreys (1995).

Many of these faults relate to technical issues in the implementation; the programs compile but do not work in a predictable or correct way. Variables are used before being properly declared, memory overflows, and so on.

The key to managing these problems is to *test thoroughly all the time*.

1.6 DO WE REALLY NEED ALL THIS MOUNTAIN OF DOCUMENTATION?

Most nonchaotic 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.

The desire for 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 that 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 emphasize the benefit of using computer-aided software engineering (CASE) and other tools that might provide support for the maintenance of the design, but many programmers dislike these systems, which are often imposed by the management, and some programmers may feel 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 that 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.

Thus 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 often does, 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.

Thus 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 should 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 maintenance in subsequent years, we should not place all our reliance on these documents. 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 the original team has dispersed.

Another aspect that also relates 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 nobody needs it, why generate it?

1.7 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 team is a vital component in the success of the project. Too often, organizations organize 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 obtain the best work from 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 for 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 focuses 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, and sometimes individuals are just unreasonable and threaten the joint endeavors 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 cooperation from the development teams; they need to be able to adapt to changing circumstances without feeling threatened or pressured. 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 because they may be held responsible for project failure or other consequences caused by problems beyond their control. Many clients are skeptical 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. Providing pieces of functioning software, albeit prototypes in some methodologies, provides some confidence that things are progressing. It also provides a mechanism for feedback from a real implementation rather than from vague abstractions.

The issue of *end-user programming* could be raised here. One of the most ambitious goals of this is to provide users with no programming experience with the facilities to build their own applications, the argument being that they know their business better than the programmers and analysts and thus they should be in a better position to know what they want. If we can give them an environment that enabled them to build their application easily, then this would overcome some of the problems.

Things aren't quite as simple as this, however. Some clients find it difficult to articulate what they want or to step back sufficiently to understand their business processes adequately to create a coherent business model and thus an application to support it.

However, there are some possibilities. In a way, spreadsheets are an example of the sort of application that many business people can create and use, although it is easy to make errors in the way these are set up and the formulas in the cells defined. It does present a possible way forward, however.

Another example is the work of Bagnall (2002) who built an experimental end-user system called Program It Yourself (PIY). This was founded on a particular approach to identifying the business model for an e-commerce site that was based directly around concrete things involved in the business, products, prices, and so forth. In trials with naïve users (i.e., nonprogrammers), he found that they could build useful and maintainable systems based on the use of an XML Hunter (2000) database supporting a user-friendly GUI that implemented a clear business model. Similar systems for other business domains should be possible.

1.8 SOME AGILE METHODOLOGIES

There are a number of possible contenders for the description of an agile methodology. We will look at some of the more popular ones, leaving extreme programming until the next chapter where we will look at it in more detail. This account is not

exhaustive but is meant to provide some ideas of the different directions in which agile development is going.

1.8.1 Dynamic Systems Development Method

The dynamic systems development method (DSDM) (Stapleton, 1997) is an approach that uses an iterative process based on prototyping that involves the users throughout the project life cycle. In DSDM, time is fixed for the life of a project rather than starting with a set of requirements and trying to keep going until everything has been done—or we have all given up! Thus resources are fixed as far as possible at the start, and this can provide a more realistic planning framework for a project. This means that the requirements that will be satisfied are allowed to change to suit the resources available.

There are nine underlying principles of DSDM, the key one being that fitness for business purpose is the essential criterion for the acceptance of deliverables. This philosophy should ensure a clearer focus on the purpose of the software rather than on technology for technology's sake.

1.8.1.1 The Underlying Principles

The following principles[1] are the foundations on which DSDM is based. Each one of the principles is applied as appropriate in the various parts of the method.

1. Active user involvement is imperative. Users are active participants in the development process. If users are not closely involved throughout the development life-cycle, delays will occur, and users may feel that the final solution is imposed by the developers and/or management.
2. The team must be empowered to make decisions. DSDM teams consist of both developers and users. They must be able to make decisions as requirements are refined and possibly changed. They must be able to agree that certain levels of functionality, usability, and so forth, are acceptable without frequent recourse to higher-level management.
3. The focus is on frequent delivery of products. A product-based approach is more flexible than is an activity-based one. The work of a DSDM team is concentrated on products that can be delivered in an agreed period of time. By keeping each period of time short, the team can easily decide which activities are necessary and sufficient to achieve the right products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables. The focus of DSDM is on delivering the essential business requirements within the required time. Allowance is made for changing business needs within that time frame.

[1]These are taken from the DSDM site: http://www.dsdm.org.

5 Iterative and incremental development is necessary to converge on an accurate business solution. DSDM allows systems to grow incrementally. Therefore the developers can make full use of feedback from the users. Moreover partial solutions can be delivered to satisfy immediate business needs. Rework is built into the DSDM process; thus, the development can proceed more quickly during iteration.

6 All changes during development are reversible. To control the evolution of all products, everything must be in a known state at all times. Backtracking is a feature of DSDM. However in some circumstances, it may be easier to reconstruct than to backtrack. This depends on the nature of the change and the environment in which it was made.

7 Requirements are baselined at a high level. Baselining high-level requirements means "freezing" and agreeing on the purpose and scope of the system at a level that allows for detailed investigation of what the requirements imply. Further, more detailed baselines can be established later in the development, although the scope should not change significantly.

8 Testing is integrated throughout the life cycle. Testing is not treated as a separate activity. As the system is developed incrementally, it is also tested and reviewed by both developers and users incrementally to ensure that the development not only is moving forward in the right business direction but also is technically sound.

9 Collaboration and cooperation between all stakeholders is essential.

DSDM is independent and can sometimes be used in unison with other frameworks and development approaches, such as extreme programming (XP).

1.8.2 Feature-Driven Design

Feature-driven design (FDD) (Coad, 1999) begins by developing a domain object model in collaboration with domain experts that is then used to create a *features* list. This is used to produce a rough plan, and informal teams are set up to build small increments over short, say 2-week, periods.

There are five processes within FDD:

1 Develop an overall model.

2 Build a features list; these should be small but useful to the client.

3 Plan by feature.

4 Design by feature.

5 Build by feature.

A *feature* is a client-valued function that can be implemented in 2 weeks or less. A *feature set* is a grouping of business-related features.

We illustrate the process in Fig. 1.4.

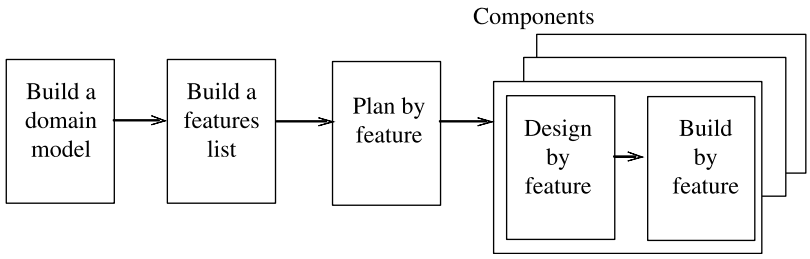

Figure 1.4 Feature-driven design.

1.8.3 Crystal

Communication is a key aspect of Crystal (Cockburn, 2001) by considering development as "a cooperative game of invention and communication." Crystal aims to overcome many of the problems caused by poor communication between all the stakeholders, particularly developers, customers, and clients.

Cockburn looks at a software development project as a bit like an ecosystem "in which physical structures, roles, and individuals with unique personalities all exert forces on each other."

The approach highlights intermediate work products that exist in order to help the team make their next move in the *game*. These products help team members to orient themselves in the project and to remind members of important issues, decisions, goals, and so forth. They also help in prompting new ideas and potential solutions to problems. These products do not have to be complete or perfect but should help to guide and motivate team members. As the game progresses, these products help in the management of it. "The endpoint of the game is an operating software system . . . ."

As we can see, the approach is more of a management framework rather than a set of explicit technical practices. There exist some policy standards and guidelines on the numbers of developers and how to assess the critical attributes of projects.

1.8.4 Agile Modeling

"Agile modeling is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner" [http://www.agilemodeling.com].

An agile modeling (AM) approach (Ambler, 2002) can be taken to requirements, analysis, architecture, and design. The idea is that whatever modeling approach is taken, whether as use case models, class models, data models, or user interface models, the emphasis should be on a lightweight but effective approach to the modeling. The model should not become the purpose but just the vehicle for understanding the customer's needs better. Because the models are light weight, they are easier to adapt or are even thrown away if they become obsolete through requirements change.

AM is often combined with notations from UML and for example the rational unified process (RUP), but the full bureaucratic treadmill often associated with these processes is reduced. AM is not a complete software process; it doesn't cover programming, software delivery, or testing activities, although testability is considered through the modeling process. There is also no emphasis on project management and many other important issues. However, AM is very sympathetic to the principles of XP that we examine in Chapter 2 and it is possible to combine AM with some of the more complete agile processes such as XP, DSDM, or crystal. In this book, we will combine a type of agile modeling with XP.

1.8.5 SCRUM

SCRUM is a management process that can be applied to a number of different activities, not just software development. Introduced in 1995 (Schwaber, 2002), it has had some success in a number of projects.

A project is divided into features with an assigned project value and estimated effort or cost. *Sprints* are time-boxed plans corresponding with iterations. Daily *scrum* meetings—which are short and focused—are held to monitor progress. At the end of each sprint, a review meeting is held to consider the quality of the features produced. Different stakeholders and actors are identified including the product owner, the scrum master, and the team members.

There have been successful approaches that combine SCRUM with other agile methodologies; for example, SCRUM might be used for the overall management of a project, which could include marketing and support activities, while an agile software development method is used for the production.

SCRUM has been applied to a distributed development project across several countries (e.g., Sutherland, 2007).

1.8.6 Summary Table

These different approaches have different strengths and weaknesses, but all adopt parts of the agile philosophy. There are a number of other approaches such as lean software development (Poppendieck, 2002 http://www.Poppendieck2002.com). We briefly summarize some of their properties in Table 1.1. In the table, "+" indicates a strong aspect featured in the approach, "−" means that this aspect is not emphasized in the literature, and "?" indicates that this can be featured but not always, and critical support for this is not a fundamental part of the method.

Some of these approaches are really more like philosophical perspectives on software development rather than a complete set of techniques and methods, some are general approaches to managing and planning software projects, and some are tied into an existing design-based approach such as UML. All have their strengths, and which ones will succeed in the industry over the next few years is dependent on many things.

Table 1.1 A Summary of the Features of the Agile Methodologies in This Chapter

Feature	DSDM	FDD	Crystal	Agile modeling	SCRUM
Clear business focus	+	+	?	−	+
Strong quality/testing focus	?	−	−	−	?
Handles changing requirements	+	+	?	+	+
Human-centered philosophy	?	?	+	+	?
Support for maintenance	?	−	−	−	−
User/customer-centered approach	+	+	?	+	?
Encourages good communications	+	?	+	?	+
Minimum bureaucracy	?	?	+	+	−
Support for planning	+	+	+	−	+

There is a shortage of scientific evidence about the benefits and weaknesses of these approaches. There are plenty of case studies, experience reports, and opinions available—in the proceedings of conferences on agile development and on Web sites that abound. However, no extensive comparative trials have been carried out in an industrial context except those undertaken in the Sheffield Software Engineering Observatory (http://observatory.group.shef.ac.uk/) where the approach described in the next chapter has been compared with a traditional design-led approach.

In the next chapter, we will focus on extreme programming, XP (Beck, 1999) and see that it will provide all of the desirable features that we have identified. It also gives much clearer guidance on how to achieve them. Some of the other agile approaches, such as DSDM and SCRUM, are proposing to adopt some of the XP ideas in a kind of hybrid approach.

1.9 REVIEW

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

1 a clear business focus;

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

3 the need to allow for the future maintenance and evolution of any delivered solution;

4 the assurance of the quality of the delivered software;

5 the reduction of the amount of documentation and other bureaucracy that is required to sustain and manage the development process;

6 the emphasis on the human dimension must be a key aspect both for the developers and the clients.

Consult the agile modeling manifesto for another perspective on the issues discussed above.

EXERCISE

Consider the Agile Manifesto reproduced here from the following Web page: http://www.agilemanifesto.org/principles.html.

> *We follow these principles:*
>
> *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
>
> *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
>
> *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.*
>
> *Business people and developers must work together daily throughout the project.*
>
> *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
>
> *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
>
> *Working software is the primary measure of progress.*
>
> *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*
>
> *Continuous attention to technical excellence and good design enhances agility.*
>
> *Simplicity—the art of maximizing the amount of work not done—is essential.*
>
> *The best architectures, requirements, and designs emerge from self-organizing teams.*
>
> *At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.*

Think about these principles and reflect on your own experiences in software development, what you have been taught about the process. How do these principles relate to these issues? If you have been involved in a significant development project, what processes did you follow?

CONUNDRUM

The following scenario is based on a real-life business situation that arose in the late 1990s.

The Internet is opening up, and many businesses are now connected. Banks are beginning to consider if they could provide online access to their business customers. One bank considers two strategies.

(A) The bank's IT director suggests that they put together a *quick and dirty* Web site that allows customers to submit transactions through their browser, to get this up and running, and to try to develop a connection with the "back-office" legacy mainframe database system.

(B) The bank also gets a report from some outside consultants that suggests they should reengineer the legacy back-end and build an integrated Web front-end to provide a powerful user-friendly e-banking system engineered to a high standard.

Which strategy would be best and why?
See Chapter 11 for a discussion of this dilemma.

REFERENCES

S. Ancha, A. Cioroianu, J. Cousins, J. Crosbie, J. Davies, K. Ahmed, J. Hart, K. Gabhart, S. Gould, R. Laddad, S. Li, B. Macmillan, D. Rivers-Moore, J. Skubal, K. Watson, S. Williams. *Professional Java XML.* Wrox Press, 2001.

M. Bagnall. The Dyna Cat System. http://www.dcs.shef.ac.uk/intranet/teaching/projects/archive/ug2002/pdf/ugmab.pdf.

S. Ambler. *Agile Modeling.* John Wiley & Sons, 2002.

K. Beck. *Extreme Programming Explained.* Addison-Wesley, 1999.

P. Coad, J. de Luca, E. Lefebre. *Java Modelling in Color.* Prentice Hall, 1999.

A. Cockburn. *Agile Software Development* (A. Cockburn and J. Highsmith, eds.). Addison Wesley, 2001.

T. Gilb. *Principles of Software Engineering Management* (S. Finzi-Wokingham, ed.). Addison-Wesley, 1988.

L. Hatton. Does OO sync with the way we think? *IEEE Software*, 15(3):46–54, 1998.

W.S. Humphreys. *A Discipline for Software Engineering.* Addison-Wesley, 1995.

D. Hunter. *Beginning XML.* Wrox Press, 2000.

R.S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw Hill, 2000.

K. Schwaber, M. Beedle. *Agile Software Development with SCRUM.* Prentice Hall, 2002.

I. Sommerville. *Software Engineering*, 8th ed. Addison-Wesley, 2006.

J. Stapleton. *DSDM: The Dynamic Systems Development Method.* Addison-Wesley, 1997.

J. Sutherland, A. Viktorov, J. Blount, N. Puntikov. *Proc. HICSS.* 2007.

Web Sites

http://www.Poppendieck2002.com.