The Book of Genesys Solutions

A practical guide to agile software engineering

Mike Holcombe

CONTENTS

Preface.

This book is a radical departure from the usual book on Software Engineering and Design Methodologies. Its purpose is to put software development into a context where *professional* skills are developed as well as the technical skills. At the end of a project based around this book, students should be much more like real software professionals than before, ready to embark on a career where professionalism, a quality orientation and an understanding of the business context are better developed than ever before.

The target audience is Computer Science and Software Engineering students in their 2nd or 3rd year who have already covered the basics of programming and design and who have had some experience of building a small piece of software. There is an accompanying Instructor's Manual, jointly written with Helen Parker which distils guidance for running real projects taken from the author's extensive experience of having 2nd year student teams build real software solutions for external business clients.

The intention of the book is to use the new ideas from the so called *Agile Methodologies*, particularly the approach known as *Extreme Programming* or *XP*, as the vehicle for teaching *practical* project development skills. XP is a rapidly evolving set of ideas that can be applied in many different application areas, we focus here on the use of XP practices in the development of some real software for a business client, perhaps from a local company or another part of the University. The book is based on 13 years of experience of teaching in this way and managing such projects. In all I have managed around 40 projects involving a couple of hundred teams, of which 35 have used XP in the last two years. I have learnt a great deal from this and have adapted XP to fit in with the demands of such *fixed period* projects. Some may argue that I am demanding too much from students but I am convinced that well motivated students will be able to perform very well using these ideas, they can not only deliver excellent software to their clients but they will also learn much more than from any other typical course on software development which will concentrate on lots of lectures and artificial exercises. As so many people comment, success in the software industry is much more dependent on personal and teamwork skills than on technical knowledge. If you are unable to work effectively in a team then you will be of little use to a software development company whatever technical knowledge you have. These sorts of projects will teach you a great deal about yourself, the realities of teamwork and of dealing with real clients. One of the key challenges you will face is getting yourself organised and in planning and working in an effective way I have tried to give practical suggestions and mechanisms for doing this. At the end of such a project you will be amazed at what you will have achieved. The appendices contain examples of systems built by my students.

The book will also address the connections between IT development and business pragmatics through the use at the end of each chapter of *Conundrums***.** These are based on real scenarios that either I have faced or have been experienced by business colleagues. In many cases these explore the dilemmas between following the established philosophy of academic software engineering and the realities of businesses driven by the need to make money.

Some basic principles governing its philosophy are:

i. It assumes that the readers will be engaged in a *real* - rather than an academic software project. This means the project is for an external business client and this factor will expose the students

to the very real problems of requirements capture and the need for the highest quality software if the client is to be able to us it in their business. Most team-based projects are designed around problems posed by the instructor and often lack credibility with students most of whom respond enthusiastically to the challenge of building something useful for a client;

ii. It assumes that the readers will be reading it as members of a software development team and will be able to undertake the key activities together;

iii. It aims to develop self learning (and lifelong learning skills in the readers), this is a *problem based learning* approach and it is expected that the students will have the need to supplement their reading by consulting the literature, text books, articles in the professional press etc. This not intended to be an exhaustive and self contained book on software engineering and software project management. I am convinced that, given the responsibility, students will rise to the challenge and develop intellectually and personally far more from this approach than traditional educational approaches;

iv. The book is not a large tome - emphasising the XP philosophy on minimal but informative and reliable documentation;

v. Unlike existing XP books this one deals with some of the practical details and provide effective methods and models for achieving high quality software construction in an `agile' manner;

vi. There is an accompanying guide for instructors/coaches which will provide practical advice on how to motivate students, organise real group projects and deal with many of the problems that arise in a simple and effective way. This is based on over 10 years of running real software projects with student teams.

vii. No specific programming language is used since particular projects will require particular implementation vehicles but some reference is made to the language Java.

Content of the Chapters:

1. *What is an agile methodology?* Discusses the business need for rapid software development and the problems that this produces. The principles of the Agile Methodologies are described.

2. *XP outlined*. This describes the 4 principles and the 12 activities involved in XP. It is meant to set the context of an XP development for a real client.

3. *Essentials*. The basics of group work and software projects. How to set up a team. Carrying out a skills audit. Choosing a way of working. Finding and keeping a client. Day to day activities. Keeping an archive. Some basics of planning. Dealing with problems. When things go wrong. Risk analysis.

4. *Starting an XP project*. Organising the team and learning how to approach the client or customer. Essential planning issues.

5. *Identifying user stories and the planning game*. How small pieces of functionality can be identified and represented. Planning the project. Techniques for estimating resources.

6. *System testing*. Developing the system concept. Building the functional test sets for the stories.

7. *Establishing the system metaphor*. Finding the right initial architecture for the application.

8. *Unit tests*. Choosing the classes. Writing the unit tests. Running the tests.

9. *Evolving the system*. Refactoring the code, working with the client, integrating the releases, user acceptance tests (non-functional testing)

10. *Documenting the code*. Providing maintenance information in the code, coding standards, documentation the test sets. User documentation, help systems.

11. *Reflecting on the process*. What has been learned, what will be useful for projects in the future.

Appendix A. A Requirements Document
Appendix B. A Software Cost Estimation
Appendix C. A User Manual
Appendix D. Writing Unit Tests in VB UNIT and PHP UNIT
Index
Bibliography

Acknowledgements.

Many people have helped me with this book. Firstly, all my students who have taught me so much over them years. In particular, Francisco Macias, Sharifah Abdullah, Chris Thomson, Phil McMinn, Alex Bell, all the students from Genesys Solutions and the Software Hut over the years. The examples of systems built by our 2nd year students in the Appendices are due to David Adams, Mark Bagnall, Terry Carter, and Andrew Cubbon.

I must also thank my academic colleagues, Marian Gheorghe, Andy Stratton, Helen Parker, Kirill Bogdanov, Tony Simons and Tony Cowling for helping me with many aspects of the work but especially Marian, Helen and Andy who have played a large part in making our real student projects such an amazing success.

A number of XP industrial experts from around the world have looked at drafts of this book, including Ivan Moore, Tim Lewis and Graham Thomas. Fellow academics and collaborating researchers from a number of universities who have also been a great help include Mike Pont, Giancarlo Succi, Michelle Marchesi, Bernard Rumpe, Leon Moonens and Jose Canos.

I would also like to thank a number of anonymous reviewers whose comments on drafts have helped to improve the book immeasurably.

Finally I must thank my wife Jill whose tolerance and support were invaluable.

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, 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 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 the cost and within the time scale that is 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 [Pressman2000] and [Sommerville2000] present a broad survey of traditional software engineering that documents many of the current approaches. Other thinkers such as [Gilb1988] and more recently [Beck1999], are beginning to question the way in which software engineering has been carried out.

Firstly thinkers such as Beck, recognise 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 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 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 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.

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.

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 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 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 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 although there may be pressure on them from managers to do so. 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 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 envisages a steady and systematic sequences 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 Figure 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.

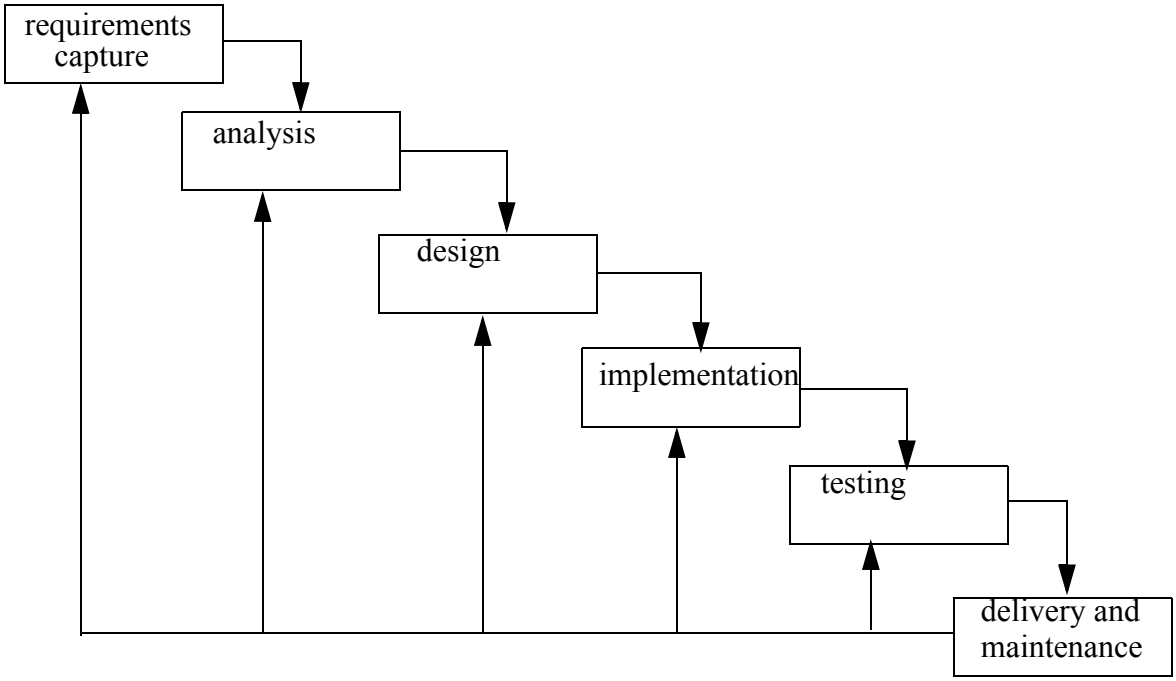


Figure 1. The Waterfall model of software development.

The first two key issues are, therefore:

to find 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.

There are a number of newer approaches to software development that have attempted to address these issues. The Spiral model ([Sommerville2000]) describes this approach, see Figure 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 recognise that for many projects there is an ongoing relationship with the customer which 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*.

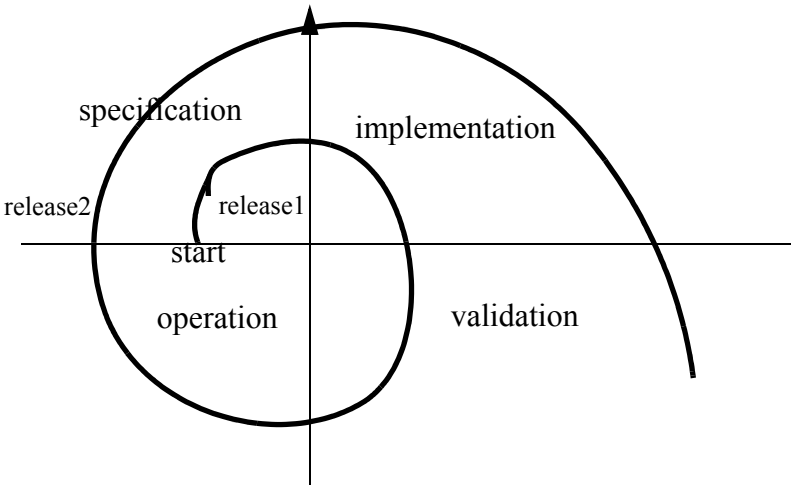


Figure 2. The Spiral model

Rapid Applications Development and Evolutionary delivery are similar sorts of approach which 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.

There have been many analyses of failed software development projects. Failures in communication, both between developers and clients and between and amongst 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 diagrams, for example, can often be interpreted differently by different people.

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 were 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 2001] that an XML database is much easier to evolve than other forms including the traditional relational database. For more about XML see [Hunter 2000]. Medcalf's experiment consisted of building the same small database using various approaches, a common relational database tool, XML and a simple flat file structure. All had the same user interface. Querying and data entry activities were compared with no significant observable difference in performance. He then went on to extend and re-engineer the databases to take account of a significant change, primarily an addition, to the business model and worked out how much effort was needed to adapt the different databases. The results were pretty conclusive with the relational databases taking about 5 times the effort as the XML ones and the flat file types about 3 times the effort. Again performance experiments were carried out in terms of querying etc. and as before no significant difference in performance was found.

The one drawback of the XML files was that they required rather more storage than the others, but storage is cheap compared to the labour of analysts and programmers when change to the database is needed.

Many interesting new developments involving XML are coming through. It is certainly worth looking at this option, see [Ancha2001].

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 or proposed system to deal with some new requirements?

In the case where there is either an existing system which 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 behaviour. 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 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 con-

trolled 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.

For projects which involve 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, 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 is clear and understandable.

If we can introduce a more appropriate database technology, such as XML, [StLaur1999], then all the better.

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.

Neither tasks are 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 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.

An approach to improving quality in a model like the Waterfall model is called the V model. See Figure 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.

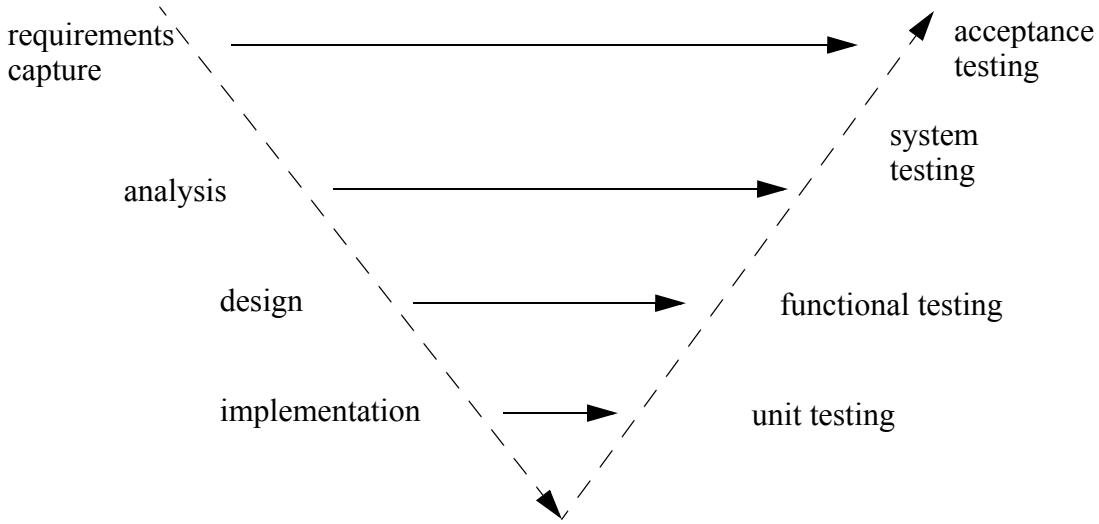


Figure 3. The V model

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, 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 com-

pounds the problems and is very hard to overcome. If a developer has spent a week or longer on some component which is then found to be seriously flawed then they are likely to resent having to start again. This is a potentially serious quality and efficiency problem. In many companies software developers spent 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 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.

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 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, 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 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 often, 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 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?

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 obtain the best work 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 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 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. 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 sufficiently 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 formulae in the cells defined. It does present a possible way forward, however.

Bagnall [Bagnall2002] built an experimental end user system called Program It Yourself, PIY. This was based on a particular approach to identifying the business model for an e-commerce site which was based directly around concrete things involved in the business, products, prices etc. In trials with naive users, i.e. non-programmers, he found that they could build useful and maintainable systems based on the use of and XML database supporting a user friendly GUI that implemented a clear business model. Similar systems for other business domains should be possible.

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.

8.1. Dynamic Systems Development Method (DSDM) [Stapleton1997]

The Dynamic Systems Development Method (DSDM) is an approach that uses an iterative process based on prototyping which 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! So 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.

The Underlying Principles[1]

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

i. 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.

ii. 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, etc. are acceptable without frequent recourse to higher-level management.

iii. The focus is on frequent delivery of products. A product-based approach is more flexible than 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.

iv. 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 timeframe.

v. 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.

vi. 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.

vii. Requirements are baselined at a high level. Baselining high-level requirements means "freezing" and agreeing the purpose and scope of the system at a level, which allows for detailed investigation of what the requirements imply. Further, more detailed baselines can be established later in the development, although the

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

scope should not change significantly.

viii. 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 is moving forward not only in the right business direction but is technically sound.

ix. Collaboration and cooperation between all stakeholders is essential.

DSDM is independent and can be used in unison with other frameworks and development approaches, eXtreme Programming (XP).

8.2 Feature Driven Design (FDD) [Coad1999].

FDD begins by developing a domain object model in collaboration with domain experts which 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 two week, periods.

There are five processes within FDD:

i. Develop an overall model.
ii. Build a features list, these should be small but useful to the client.
iii. Plan by feature.
iv. Design by feature.
v. Build by feature.

A *feature* is a client-valued function that can be implemented in two weeks or less. A *feature set* is a grouping of business-related features.
We can illustrate the process in the following diagram.

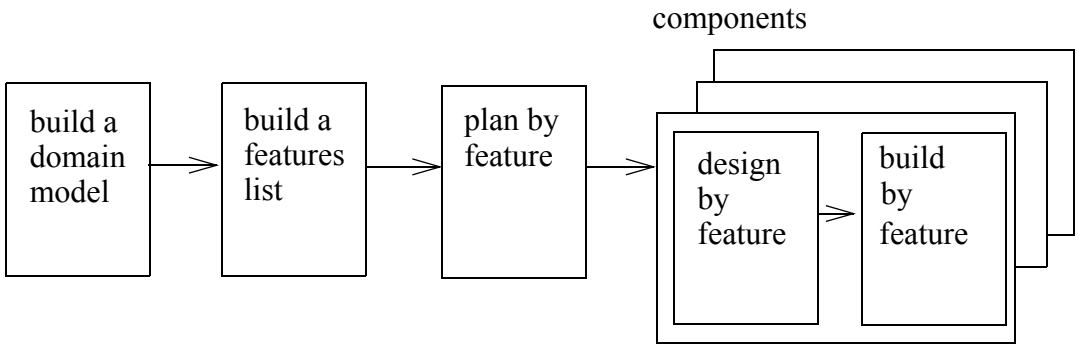


Fig. 2. Feature driven design.

8.3. Crystal [Cockburn2001].

Communication is a key aspect of Crystal and 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, clients.

Cockburn looks at 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 which 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 etc. 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 are some policy standards and guidelines on the numbers of developers and how to assess the critical attributes of projects.

8.4. Agile modeling [Ambler2002].

“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 approach can be taken to requirements, analysis, architecture, and design.
The idea is that whatever modelling 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 modelling. The model should not become the purpose just the vehicle for understanding the customer’s needs better. Because the models are lightweight it easier to adapt or even through them 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 or software delivery, testing activities, although testability is considered through the modelling 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 modelling with XP.

8.5 Summary table:

These different approaches have different strengths and weaknesses, all adopt parts of the agile philosophy. There are a number of other approaches such as Scrum [Schwaber2002] and Lean

Software Development [Poppendieck2002]. We briefly summarize some of their properties below. 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.

Feature	DSDM	FDD	Crystal	Agile modeling
clear business focus	+	+	**?**	**-**
strong quality/testing focus	**?**	**-**	**-**	**-**
handles changing requirements	+	+	**?**	+
human centred philosophy	**?**	**?**	+	+
support for maintenance	**?**	**-**	**-**	**-**
user/customer centred approach	+	+	**?**	+
encourages good communications	+	**?**	+	**?**
minimum bureaucracy	**?**	**?**	+	+
support for planning	+	+	+	**-**

Table 1. A summary of the features of the agile methodologies in this Chapter.

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, 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.

In the next Chapter we will focus on Extreme Programming, (XP), 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.

9. Review.

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

* a clear business focus;

* the ability to plan and 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 maintenance and evolution of any delivered solution;

* the assurance of the quality of the delivered software;

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

* 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.

1. Consider the AGILE MANIFESTO reproduced here from the following web page.
http://www.agilemanifesto.org/principles.html

"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?

2. Have a look at the report produced by Mark Bagnall ([Bagnall2002]). This is the project of a third year undergraduate student who tried to apply some of the ideas behind extreme programming to develop a novel experimental system.

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 on-line 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 which 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 which suggests that they should re-engineer 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***.**

[Ancha2001]. 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.

[Ambler2002]. S. Ambler, "*Agile modeling*", John Wiley, 2002.

[Bagnall2002]. M. Bagnall, "*Extreme programming and end-user programming*", 3rd year undergraduate report, University of Sheffield, 2002, available on line at: http://www.dcs.shef.ac.uk/teaching/eproj/ug2002/abs/u9mab.htm

[Beck1999]. K. Beck, *"Extreme Programming Explained"*, Addison-Wesley, 1999.

[Coad1999]. P. Coad, J. de Luca& E. Lefebre, "*Java modelling in color*", Prentice Hall, 1999.

[Cockburn2001]. A. Cockburn, "*Agile software developmen*t", (A. Cockburn & J. Highsmith (eds)), Addison Wesley, 2001.

[Fowler2000]. M. Fowler, *Refactoring - Improving the design of existing code*, Addison Wesley, 2000.

[Gilb1988]. T. Gilb, *Principles of software engineering managemen*t, edited by Susannah Finzi. - Wokingham : Addison-Wesley, 1988.

[Hunter 2000], D. Hunter. *Beginning XML*. October 2000. Wrox Press.

[Medcalf2001]. A. Medcalf, "*Evolving databases*", 3rd year undergraduate report, University of Sheffield, 2001, available on line at: http://www.dcs.shef.ac.uk/teaching/eproj/ug2001/abs/u8ajm.htm

[Poppendieck2002] <http://www.Poppendieck2002.com>

[Pressman2000]. R. S. Pressman, *"Software Engineering a practitioner's approach,"* McGraw Hill, 2000.

[Schwaber2002]. K. Schwaber & M. Beedle, "*Agile software development with SCRUM*", Prentice Hall, 2002.

[Sommerville2000]. I. Sommerville, *"Software Engineering*", Addison-Wesley, 2000.

[StLaur1999]. S. St. Laurent & E. Ceramie, *Building XML applications*, McGraw Hill, 1999.

[Stapleton1997]. J. Stapleton, "*DSDM: The Dynamic Systems Development Method,*" Addison Wesley, 1997.

Chapter 2

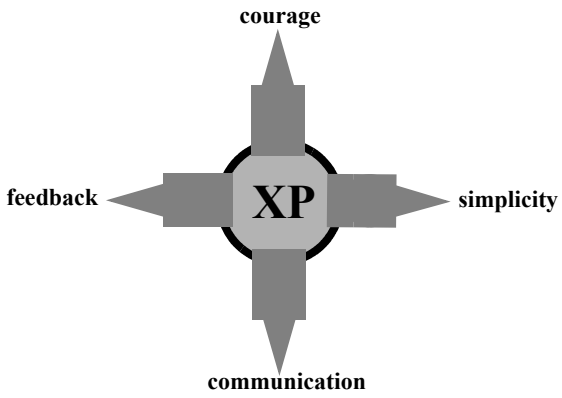
EXTREME PROGRAMMING outlined

Summary: The 4 values and the 12 activities involved in XP are introduced. These are reviewed and discussed in the light of some current experiences in applying XP in industry.

1. The four values.

Before we get into the more detailed description of what XP is all about we need to understand the fundamental values that are its reason for existence and the reason for its success.

These four basic values of XP are:



Communication

Almost all the research that has been attempted into the great software engineering disasters has concluded that breakdowns in communication between developers and client, amongst the clients and amongst the developers play a major role. In a sense, computing is all about communication from human to computer to human and thus the very essence of our subject requires that we address this in a fundamental way.

XP tries to emphasise this factor by building a rich collection of procedures and activities that emphasise effective communication amongst ALL the stakeholders.

Let us look at some of the most important areas where communication is vital. The first one doesn't involve the developers at all. Consider a company that wishes to have some software developed to support its business activities. The first and most vital requirement is that they can decide what the principal objective of the software that they need is. This requires them to understand their business, its context, the strategy of the business and so on. For this to be done successfully there has to be good communication amongst the principle players in the company, the directors, managers, operators and possibly their clients and business backers. Many

software disasters have been caused by failures at this level. Perhaps the company has not thought through its business objectives properly, is the proposed software either needed or providing the most business value? It is often the case that the reason for the software becomes obscured, perhaps the principle *champion* in the company leaves or changes their role in the company. Someone else might take over this responsibility and may either be unaware of the motivation for the development or unsympathetic to it.

It is therefore vital that the company is clear about why it wants the software developed, has analysed its operations sufficiently well to be able to justify it on business grounds and that there is a knowledgeable champion for the development who is well connected with ALL the stakeholders in the company. We will rely on the existence of these parameters during our project. If something is wrong here then there is a strong chance that we will be building the wrong system, a waste of time for all concerned.

The next issue to address is the communication among developers and the client. This is also, vital. It is no good having one meeting at the start of the project and then to meet again when the supposed solution is delivered. This is bound to be a disaster unless the system is fairly trivial in nature. So much can change in the business between the start of the project and the final commissioning of the solution that there has to be much more regular communication between these two parties.

The communication needs to provide several benefits. Firstly it has to provide a continuous or, at least, frequent, renewal of the business requirements that are being addressed. As has been pointed out earlier, business needs can change rapidly and the purpose of the software could change with them. We must be aware of what is happening in the business and the way that things are changing. This agility depends heavily on the communication mechanism between clients and developers (also between developers and amongst the clients' business partners).

As well as receiving this information from the clients the developers need to keep the client informed of how they are doing. There is nothing more frustrating for a client than not to know how things are getting on. They are paying for all this and there will be many other demands on their money. Regular feedback on progress, and demonstrable signs of progress are needed.

The third aspect is the communication between the developers. This is often sadly lacking in traditional development regimes. The communication process here involves keeping all the team involved in the planning of the project, keeping everyone up to date with progress, with objectives and with the changing nature of the target. This is very difficult and usually results in some of the team becoming disengaged and de-motivated if it is not addressed. As we have seen in previous chapters the human side of the management of the team becomes crucial. Giving people respect and responsibility provides a good basis for the development of rich and productive communication processes within the team. Several XP practices contribute directly to this goal, as we shall see.

Feedback

Feedback is closely related to communication, they are two dimensions of the same phenomena.

We need to establish very rich mechanisms, as we saw above, to keep the client informed and involved in the project. This is to ensure that we are building the right system for the business and that we are making clear progress towards the joint objectives of all concerned. Thus there needs to be a mechanism for the client to see real results of the developers' efforts and to try to relate them to his/her business activities and needs. Traditional design-led approaches rely on producing large amounts of, often, incomprehensible, documents to do this. This is a ponderous and, ultimately, unrewarding endeavour. Regular increments of software can help this but can cause a distraction if the quality is poor and the client is sidetracked into doing the testing that should have been carried out by the developers and having to report faults and bugs. It is no good delivering a prototype or an increment of the solution if it is unreliable and fails to meet the clients quality expectations. We must avoid this - previous approaches such as Rapid Applications Development (RAD) failed in this respect because it was based on the rapid development of, possibly arbitrary, increments rather than on the rapid development of *high quality* increments *that add business value* to the client's business.

Within the development team we need to ensure that everyone knows what is going on, where the project has got to and how their work fits into the *big picture*. They also need to know how good their work is and how good the work of all the others is. Building on the work of others when you have doubts about its quality is always a frustrating process. We need to avoid this. It is no good relying on the occasional review meeting. Although these are necessary and often productive they can also be a source of great problems.

Imagine the following scenario, typical of most traditional development projects. The managers have allocated you some aspect of the development to code up. You might receive some textual descriptions or requirements of what is needed, you might receive some design documents and it is your task to deliver some code by a deadline, perhaps a week or longer. So, you go to your machine, which may well be separate from or shielded from others working on the project. You then spend the next few days trying to get your head round what it is that you are supposed to do. After a while the manager gets fed up with your questions and requests for clarification - probably he/she doesn't know the answer, maybe the client should be asked but everyone is too busy for that. So you struggle on and eventually manage to deliver the code by the deadline.
There is then a review or inspection meeting where your code is looked at by others, managers, other programmers etc.[1] At the review they start criticising your code. You have sweated over this and have done your best yet they complain about many things. You misunderstood a requirement but when you asked them about that very thing they either didn't know or told you to sort it out yourself. At points where you showed initiative they criticise you for failing to

1. This would only happen in a so-called "well organised" company, in many review is not a formal process and the only reviews take place during integration testing when vast amounts of time and money are spent on the futile task of trying to find and fix bugs.)

follow some, previously unknown, house convention or requirement. Criticising your detailed code may involve taking your algorithms apart and suggesting that they would have used "better" ones. Perhaps some smart guy knows about a clever way to do what you did with half the effort. I could go on. Suffice it to say that you are soon on the defensive and getting angry or demoralised. They want big changes and you would prefer to try to fix the problems by some judicious *tweaking* of the code. In many situations the best solution is to start again having obtained a better understanding of what is wanted and what the "best" solution might be. However, human nature often conspires against this and the tweaking approach is often adopted. Anyway, it is probably too late to do anything else with the deadline approaching.

We have to find a better way.

Simplicity

How many times have you used some software where there were complicated and confusing features that *got in the way*? If this is the case of computing experts how much more is it the case for ordinary users?

Many projects get into trouble because the developers get sidelined into doing something that is technologically novel or "clever" when, in fact, the feature in question is just not really needed. Clients can be seduced by such "enhancements" too and could agree to some new fancy feature being added when it makes no sense to do so, it adds nothing to their business capability. These extra features are a potential threat to the success of the system. They introduce unwanted complexity into the systems, especially if the delivery deadline is fast approaching because the work on the new feature will, probably, be at the expense of more thorough testing of the software. Some call this *feature creep*.

Einstein once said that "any solution should be as simple as possible but no simpler".

We need to adopt the same attitude. Every aspect of the system should be considered, can we really justify the time and effort in adding some supposed enhancement. However, if the reason for adding a layer of complexity is a good one, for example in order to make the software more robust by trying to trap inappropriate data input, then we have to do this. But we must have suitable tests to demonstrate that we have done it properly.

Courage

This means having the confidence to do things that might otherwise be considered risky. Much of the philosophy of XP derives from abandoning some of the traditional ways of software development, ways that are widely taught and widely used in industry. It takes some nerve to turn one's back on all this expertise and experience.

One aspect of courage that XP and other agile approaches promote is the enthusiasm for change, in particular a willingness to adapt to the clients' changing needs as the project devel-

ops. This does take some courage since it may involve changing some of our previous work, there is a natural tendency to resist change in traditional approaches under these conditions. The ability to relish new challenges is part of the underlying philosophy of XP.

Extreme programming, like an extreme sport, is software development without the normal constraints. Like climbing mountains without a rope, building software without a design seems, at first sight, to be suicidal. Why it isn't is the subject of much of this book. There are constraints, and the practices of XP are meant to be followed.

Rather than being an informal and unregulated exercise it is in fact highly disciplined. You will have to learn how to enjoy the disciplines and to revel in the practices until they become second nature. It is only by making them automatic and natural that you will then gain the confidence to attack any software project with the certainty that you will succeed as well as anyone could.

We will see that there is a coherence and a rationale about the key set of values and practices of XP which will support us in our endeavours.

Confidence is one thing but over-confidence is another. You are not always right, others may have a valid point of view, too. As we have observed, learning how to argue from a position of knowledge has to be moderated with the ability to compromise and agree when others have the best argument. In the end it is important that those involved negotiate an acceptable outcome.

2. The twelve practices of XP.

2.1. Test first programming.

Before writing any code programmers build a set of tests. These tests are run – of course they will fail as no code has been written. Why would one do this?

To get used to testing continuously –
at the end of a session, at the end of the day, whenever a small piece of code has been built -

ALL the test sets are run, this means -
all the relevant unit tests, testing classes and methods as they are coded;
all the functional tests, testing at the integration level and derived from the planning game and subsequent discussions with the client;
all the non-functional; tests.

The test sets are the most important resource and are continually enhanced.

The customer helps to supply tests. So functional tests are derived from the planning game (see below) using techniques defined in later chapters. The quality of these tests is crucial and the methods described will provide test sets of outstanding power.

In a sense the test sets replace the specification and the design. They present us with a rapid feedback mechanism that tells us if the code is “correct”.

If any tests fail the code must be fixed.

This sounds very plausible since it is known that strong testing delivers quality systems. However, is it realistic. Testing as an activity is something of a Cinderella subject both in universities and industry. There are few courses dedicated to the subject and when programming is taught testing is generally ignored. Programmers are often left to their own devices in terms of what techniques to use. Here, though, testing is fundamental, the development process is centered around testing, this is what gives us continuous feedback on how we are doing. But there are tests and tests. Any fool can write test cases but only the smartest developers can write really good ones. Furthermore, we have to design the tests before we start to code. This presents another problem since many test techniques, the so called White Box testing relies on having the code structure available. These types of testing are based on finding test values that will exercise the program graph, for example traversing every path in the code, accessing every decision point etc. Many of these techniques can be automated by using the code as a basis for the generation of the tests.

In terms of functional testing and acceptance testing the tests are often created on a fairly informal basis from whatever requirements are available. There is almost no knowledge of how good the tests are. Many developers will stop testing when the rate of discovery of defects slows down - this does not mean that *all* fundamental flaws have been discovered.

We will address this issue of testing fundamentally in this book.

2.2. Pair programming.

Two people - One machine. This is a key feature. Organise the project so that when any work is being done it is done in pairs. One person using the keyboard and the other looking at the screen.

All code must be written in this way. This is a process of continuous review and ensures that mistakes are made less frequently and the reasons for doing something in a particular way are open to discussion throughout. In fact, it not only applies to coding, all aspects of an XP project should be like this, pairs of people working together, pooling their expertise and intellect and sharing information. Planning and discussing the project with the client should also involve as many of the team as possible.

The pairs swap around regularly, swapping roles within a pair and swapping developers from one pair to another gives a much greater understanding of what is being done in the project.

It is also an excellent mechanism for learning, your partner may be an expert in some aspect of the project or the techniques being use and you will then benefit from this. Perhaps they know

the programming language better than you, you are bound to benefit from such a pairing. Perhaps you have some skills that you could transfer to others. Even if you think that you know all about something the process of trying to explain it to someone else can be very beneficial to improving your own understanding. Everyone should benefit, part of your motivation is thus to become multi-skilled and to enhance your technical knowledge quite apart from completing the project successfully.

Success does need to be built upon mutual respect amongst the team. You will get to know all of the team because different pairs will form up regularly and so communication throughout the team is enhanced. Pairings will change at suitable points in the project, perhaps someone has some specific knowledge that someone else needs to learn, perhaps the change will be driven by availability of personal. Ideally, everyone should have the opportunity to work with everyone else during the project.

One interesting observation of the difference between XP projects and traditional ones is that the XP teams are always talking to each other. When you walk into an XP site this is very noticeable, there is a lot of noise compared to the traditional lab where everyone is silently staring at their screens and very little talking is going on, what there is may not be relevant to the project.

2.3. On-site customer.

This is recommended, if it is possible, since it will enrich the communication between the client and the development team. The customer/client has the authority to define the system functionality, set priorities and to oversee the direction of the project. Of course, it might be difficult to actually have the client in the development team at all times and it may not even be desirable. If the key issue is to be able to respond to sudden changes in business need then the client needs to be well connected back to the business in order to achieve this. I prefer a very close relationship, regular visits and meetings both at the development team but also in the business. Team members need to familiarise themselves with both the operating environment and a representative sample of the users of the system if they are to fully understand the issues involved. This could be better than a permanent presence of the client in the development team.

One of my projects hit problems when we delivered part of the system only to discover that the role of the *actual* users did not correspond to what the client thought, he did not understand some of his business' processes. We had to go back and rebuild the system. We wish, now, that we had spoken with more people in the business, in the presence of the client, of course, and thus been able to identify the business processes better.

It is an old adage that the client never knows what he or she wants and this is often the case. We have to question the clients and all the stakeholders in the business carefully and rigorously if we are to move towards identifying exactly what the business needs are and how they can be supported.

Excellent communications between the development team and the business should reduce the volume and cost of documentation as well as ensuring that the right system is being built.

As with pair programming this aspect of XP encourages intense face-to-face dialogue.

2.4. The planning game.

The customer provides business stories and estimates are made about the time to build software to implement the stories. We will see later how to approach the issue of identifying stories. The essence is to identify small pieces of meaningful functionality and to describe these on a small card in such a way as to illustrate the sequences of interactions that are involved in the story process. From this information, which should be clear and understandable to the client as well as the developers, we construct test sets that will be applied to any implementation that is supposed to implement that story.

Designing the test set for this purpose is a technically challenging task and one that is crucial, if we get it wrong then we are in trouble. Some authors suggest that the client should determine the stories. This must be inadequate, if testing and test set generation is a key professional activity then the task should be carried out by a professional. The client needs to be involved and to identify many of the cases that have to be addressed but for the really rigorous testing that we need to use more sophisticated input is needed. This does not mean that the development team cannot do it. They can and the techniques described in Chapter 6 and beyond, will address this.

For each story we also need to identify any non-functional requirements, see [Gilb88], that are stated or implied in the initial project description. This could relate to usability, efficiency, etc. and accurate metrics for measuring these and criteria for deciding when they have been achieved need to be agreed. This is a system level rather than a unit level exercise although the way the units are built will influence the results of these tests.

Thus we have tests which are determining whether the functionality is correct and tests which will establish whether the non-functional requirements are also satisfied. Neither should be forgotten or skimped.

For each story we need to try to identify the cost of implementing it, how long will it take and how many people will it need. This is a difficult and error prone activity, only experience will help and it is thus *really* important that you record your initial thoughts and compare them, later, with the reality. Only in this way will you develop the experience to make such judgements in the future.

Once a collection of meaningful stories have been agreed and costed then the customer decides which stories provide the most business value. This has to be done with a clear measure of the way these benefits can be measured and in consultation with the other key players in the business.

The programmers then implement the chosen stories.

2.5. System metaphor.

So, now we have some stories to build, how do we get started? The test set generation process, which focuses on the business processes in the stories and how these might be integrated into a solution, will provide us with some clues. As part of this we are, maybe implicitly, building models of the behaviour of parts of the system. This is an important resource and so we will already know quite a lot about the system level, functional requirements needed.

We now try to organise a collection of classes and methods that will achieve the functionality described by the stories under development. As we will see, below, we need to keep in mind that we will integrate these stories into stand alone and deliverable chunks of software and so our decisions here should reflect that. There is something of a trade-off in terms of how much effort is invested in defining a metaphor and the amount of flexibility is needed to deal with changing requirements. Initially, the metaphor may be rather vague as you research the problem with your customer. Soon parts of it will become more firm and these can then be documented more precisely.

The programmers define, perhaps, just a handful of classes and patterns that shape the core business problem and solution. This is like a primitive architecture. There are many ways to try to do this, one may wish to utilise some existing patterns or libraries in order to reuse existing resources.

If this is the case, however, it is important that

a) you fully understand what is being reused and
b) the reuse is natural and provides the sort of software components that really do help with the story.

We will make no assumptions about the quality of the reused components. If they have been produced through an XP approach then there will be full test sets available which you can use, extend and adapt for the new stories. If not then it is vital that they are fully tested and the test results properly analysed.

The system metaphor will be used as a means of communication between programmers and customer. The notation chosen to represent it, therefore, has to be understandable and representative of what you are trying to do.

This area is still a subject for research whether you are using XP or not and sensible notations and approaches are much sought after and rarely found. We will return to this issue later.

2.6. Small, frequent release.

Release early and release often, that is the philosophy. Once we have produced an implementation of a story that provides some coherent business benefit we deliver and install it in the client's business. This then provides the users opportunities to look at it and to provide feedback through the client to the development team. In many cases there are simple interface improvements that can be made or it might lead to a greater awareness of how the whole system might support the business. This might cause some revisions of the project scope and requirements and is thus valuable to the development team. The release might be re-engineered to suit the new understandings.

So, we do not regard these releases as prototypes, each release is real, each release is functionally useful, each release implements more stories, each release is thoroughly tested.

2.7. Always use the SIMPLEST solution that adds business value.

As we have mentioned before, it is often tempting to develop something that is more sophisticated than is needed. We must avoid "bells and whistles", that is unnecessary features which, although they might be smart, technologically impressive or just plain fun to build, are not actually needed.

Always ask – does the customer really need this feature?

For the programmer this philosophy could be embodied in the practise of using, for example, the minimum number of classes and methods to pass the tests. There are some dangers, here, however, and they will be looked at under 2.11. Simplicity of code does not always correspond to simplicity of function, as we have observed.

2.8. Continuous integration.

Code is integrated into the system at least a few times every day. All unit tests must pass prior to integration. All relevant functional tests must pass afterwards.

This is a major source of confidence that the team are getting somewhere. Rather than trying to integrate all the software (classes etc.) together at the end we integrate whenever we can. Adding trusted new stories to the current state of the system which is also well tested, requires the running of all the previous functional or system test cases. If everything passes then we know that we have built a system to supersede the previous version, it works and delivers something useful to the client.

We can deliver it for further feedback and go on to the next set of stories.

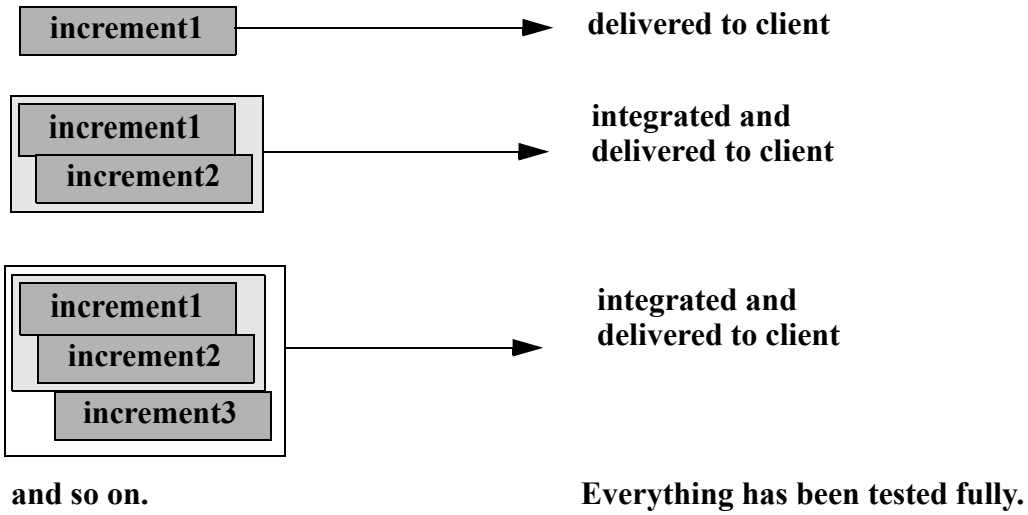


Figure 1. Incremental delivery.

It is sometimes recommended that only one pair should be responsible for integrating all the code into an operational system. The integration process must be done carefully if we are not to undermine much of what has been achieved previously. Whoever does it should do it in the full knowledge and agreement of the entire team at a time that is appropriate to the project. Letting anyone in the team carry out system integration whenever they feel like it will lead to chaos and many different versions of the core system. Successful integration can be achieved by having a project directory structure that gives authorised team members sole write access to the directory with the latest version of the running system.

2.9. Coding standards.

These define rules for shared code ownership and for communication between different team member's code.

They should involve clearly defined and consistent class and method naming protocols that everyone is familiar with.

Everyone should use the same coding styles. These need to be agreed at the start of the project, they will be dependent on the context of the project, the programming language used and the existing resources available.

Similar conventions should apply to the way that test sets are defined and to the user story cards. These need to have a set format and we discuss this later.

The benefits of clear conventions should be obvious. The source code, the stories, the metaphor and the test sets are the major project descriptors, they replace the design. They therefore must not fall into the same trap that much of the design notations suffer from. They must be well understood and relevant for the job in hand.

It is worth exploring the use of XML and of suitable tags in these sources to enhance understanding, to structure thinking and to allow for the use of suitable semantics extractions tools and query mechanisms. Naturally these tags should be "neutralised" as comments in anything that has to be compiled or run.

2.10. Collective code ownership.

ALL the code belongs to ALL the programmers. Anyone can change anything.

This is a controversial aspect of XP and seems to go against common sense and current practice. But we are dealing with a situation here where there are much richer communication processes, where all the team members are fully involved, through pair programming, with all aspects of the project. The common use of code standards will also mean that each team member should be able to understand any piece of code, what its purpose is and how it fits into the overall plan of things. If someone changes some code, perhaps to make it better in some way, then this should be apparent and if others disagree then they can change it back.

Because the code does not *belong* to any one person there is no-one to get defensive and possessive about it. This should lead to a more relaxed, but at the same time, a more consistent awareness of what is happening in the project.

Since there are house rules for writing and documenting code and for communicating between teams we should be able to benefit from this inclusive approach to the project resources.

2.11. Refactoring.

Refactoring is defined to be the *restructuring code without changing its functionality.*

Its use is mainly to SIMPLIFY code – make it more understandable, and thus more maintainable. This is vital. We have no design, although we have observed that the design may not be accurate or that useful for maintenance something has to take its place and be more effective. These are the stories, the test sets and the code.

Refactoring, see [Fowler2000], could involve a number of improvements:

Moving (extracting) methods used in several classes to a separate class;
Extracting superclasses;
Renaming classes, methods, functions;
Simplifying conditional expressions;
Reorganising data

Some basic support for refactoring is supplied by *Refactoring browsers.* [***REF***].

You may have noticed that there is an issue here regarding the unit tests. If we have unit tests defined for each class and the class structure changes because of some refactoring, for example, extracting a method into a ne, dedicated class, then there is a mismatch between the new set of classes and the set of unit tests. What this means is that we should also refactor the tests to preserve the relationships between the classes and their tests.

At the systems level the refactored system should still pass the functional (systems) tests because the functionality of the system should not have been affected by the refactoring. For the sake of maintenance, however, the link between classes and unit tests should be kept, if at all possible.

2.12. Forty hour week.

Tired programmers write poor code and make more mistakes. Since much of the software industry is reliant on the heroics of individuals working round the clock to meet deadlines it is hardly surprising that mistakes are made. We need to get away from this treadmill approach.

The way that XP is organised helps to eliminate stress caused through unrealistic time scales, lack of knowledge and understanding about what is going on, worries about the quality of what is being built, the timeliness and usefulness of the solution for the client and the concern that so much time has been spent on design that the final coding and integration will present a mountain to climb, with testing left to the end and neglected.

So, XP is supposed to minimise this stress with its emphasis on communication, feedback, quality, incremental builds and the rest. It should minimise the need for overtime and remove the panic. In comparative experiments I have undertaken with real projects being carried out by competing teams, XP and traditional, it was quite clear that the stress levels and the panic are much reduced using XP. XP adherents claim that it offers a more sustained approach to development, one that allows a steady pace and an improvement in quality as well as greater job satisfaction. There is some circumstantial evidence that this might be the case.

Because much more progress can be seen to be being made working for fewer hours is now a feasible strategy.

3. Review.

We have briefly described the values and practices of XP. They seem to make a lot of sense but do they work in real projects? The first thing to say is that this style of software development may not be suitable for all types of application and industry. Very large projects involving hundreds of developers will be extremely difficult to bring to a successful conclusion whatever method of working is employed. It may be that some of the elements of XP will be useful in these situations - test first is an obvious one - but only time will tell. Certainly, current methods are problematical as a perusal of the trade literature and its articles about recent failed projects will testify.

In some business contexts it will be difficult to apply all the practices fully. For example, not all software development is of a bespoke nature. Many software houses build speculative or generic software for a particular market and there is no client who could take the role of an on-site customer. There is a community of potential purchasers and users. However, it is possible to adapt XP successfully for this situation and a number of companies have been very successful at doing so. A good model is to set up a separate Quality Assurance (QA) department which plays the role of the customer as well as carrying out some of the acceptance testing including some of the non-functional requirements testing. For this to work the QA group must be very well connected with potential customers and know their business needs extremely well.

Another issue is with companies carrying out bespoke development on a *fixed price contract* basis. Here it does not make sense to have a highly dynamic requirements which is subject to continuing change. Accountants and lawyers on both sides will not accept this. It is important to have the requirements capture phase ring fenced so that after a certain period of the contract the requirements are more or less fixed. It may be possible to make small cosmetic changes later on but the key functionality has to be defined and will be the subject of a formal contract. because this is a fixed price contract the amount of time and resources available to the suppliers will also be limited and this is vital if they are not to get into financial difficulties. So the on-site customer may only be on site during the initial requirements capture stage and then at prototyping and incremental delivery times. In practice it is this type of contract that we will be focusing on in this book. Your time is limited to a semester or whatever and so the fixed price approach is the best. It will require rather more planning and organisation since you only have a limited amount of resource - time and labour - at your disposal.

Some software houses have long term open contracts with their customers. Their role is continual software development, perhaps of a major system that supports a changing list of functions, and so there is a lot of scope for a continual and close relationship between the customer and the developers. An ‘on-site’ customer is then both practical and desirable. In a large software project involving several teams then some of these might have a purchaser/supplier relationship with each other. Thus internal customers can be treated as on-site and the use of XP in a large project might be feasible.

The business context for an organisation employing XP will have an effect on the way XP is implemented. For example, following the financial scandals in some large US corporations, accountants and lawyers are likely to be much more cautious about committing to expenditure without any contractual or documentary evidence about a software project. Thus a clear and well defined requirements document may have to be produced in an XP project. A collection of scrappy story cards will not be sufficient. Even within a software company it will often be necessary for managers to have evidence of clearly planned and resourced projects. The way in which XP adapts to these pressures, which some may resent will be critical to its future success.

4. Preparing to XP.

The purpose of the rest of the book is to provide you with the insight and the support to make a real XP project an enjoyable and successful experience. Nothing can be guaranteed, of course, but whatever happens many lessons will be learned and at the end of it you may be in a much better position to answer the question: does XP work?

Exercise.
This exercise assumes that you are about to start working in a small group of 4 or 5 people on a software development project. It is intended to provide an early experience of some of the XP practices. It is recommended that the exercise is done in a college laboratory or terminal room,

Objectives.
To introduce the idea of pair programming and to carry out a simple pair programming activity which also relates to the activity of writing unit tests. It also tests out communication within the team and points towards the use of coding standards.

Method.

Form into a pair.

Change round on the machine every 20 minutes.

Each pair will develop a simple Java program which does the following:

> *takes as input a list of characters representing team members and a number representing work sessions and outputs something equivalent to a 1x1 table with columns indexed by the number of sessions and lists of 'pair's so that both 'pair's are present in each session.*

Example 1. input: {A, B, C, D, E} - a five person team and 6 sessions.

output:

Table 1:

session	1	2	3	4	5	6
pairs	{A,B}, {C,D}, {E}	{A,C}, {D,E}, {B}	{D,E}, {B,C}, {A}	{B,C}, {A,E}, {D}	{D,E}, {A,B}, {C}	{E,A}, {C,D}, {B}

Thus in the first session A and B pair up and C, D pair up and E operates on their own.

Example 2. input: {A, B, C, D} and 5 sessions.

output:

Table 2:

session	1	2	3	4	5
'pair's	{A,B}, {C,D}	{A,C}, {B,D}	{D,C}, {A,B}	{B,C}, {A,D}	{B,D}, {A,C}

It is not required that the programme has to display the results in such a table, just lists will do.

The first task is to write the test cases. This is not particularly easy as there is usually very little emphasis on testing and writing tests in programming courses. Later we will see how to do this in a more systematic way but for the time being write down simple test sets which provide two things: input values and the corresponding outputs.

You need to develop a test environment based on your test cases and JUnit. Log onto:

www.XProgramming.com

and find the JUnit software for Java. This was written by Kent Beck.

Download this into your account and read up the accompanying documentation.

Change round on the machine every 20 minutes.

Prepare some brief notes - just sufficient so that you can make sense of how it is used.

Now start the coding.

Run your tests even if you have not finished coding.

Debug as needed.

Continue coding and testing.

Don't forget to change places every 20 minutes or so.

Now read up the Java coding standards ((peep ahead to Chapter 8 or look at <http://www.dcs.shef.ac.uk/~wmlh/Java.pdf>).

Review the code to see if the coding standards are met. If not refactor, that is adjust, the code to ensure compliance. Retest the code.

Discuss how you find pair programming. Talk about its good points and those aspects that you found difficult, annoying or wasteful.

Conundrum.

Your client has already built a prototype system and wants you to develop it further so that he can then market it. He needs to demonstrate something fairly soon to his business backers in order to persuade them to put more money into the development of the system.

The original system is very poorly written, the database is badly structured the code is all over the place and it is going to be a nightmare to maintain.

Should you:
a).carefully document the functionality of the system and start re-engineering it before the adding new functionality?
or
b). carry on building the prototype based on what has already been done?

A discussion of this dilemma is to be found at the end of Chapter 11.

References.

Web Sites

The *"Extreme Programming Roadmap",* <http://c2.com.> the biggest web site dedicated to XP .

A popular XP site <http://www.xp.programming.com>, which includes various articles exploring XP in more detail

Books

[Beck1999]. K. Beck, *"Extreme Programming Explained"*, Addison-Wesley, 1999.
[Fowler2000]. M. Fowler, *Refactoring - Improving the design of existing code*, Addison Wesley, 2000.
[Gilb88]. T. Gilb, *Principles of software engineering managemen*t, edited by Susannah Finzi. - Wokingham : Addison-Wesley, 1988. - .
[Jeffries2001]. R. Jeffries, "*XP Installed"* is available on the website (www.xprogramming.com)

Chapter 3

Essentials.

Summary: Group work and software projects. How to set up a team. Carrying out a skills audit. Choosing a way of working. Finding and keeping a client. Day to day activities. Keeping an archive. Some basics of planning. Dealing with problems. When things go wrong. Risk analysis.

1. Software engineering in teams.

Almost all software that is produced commercially is developed with teams of people. The teams might be structured into programmers, testers, requirements engineers etc. It probably has some hierarchical arrangement with managers, team leaders, sub teams and so on. The team could be a small one, perhaps 2 or 3 people or it could involve hundreds. The team may all be working in the same place or it could be scattered over different locations, countries, even. What is common to all these manifestations is that they share a general objective, the production or development of some software product.

Learning how to work effectively in a team is thus a vital part of one's education as a software engineer. Many universities and colleges provide some place in the curriculum where a team project is set up and you have to participate with colleagues in a design project. In many of these activities the professor or instructor will set some problem and you try to solve it, learning along the way from the many experiences you share, good and not so good, which relate to the way your team worked.

There are many sources of advice about how to make the most of a team but there are no easy rules or procedures. A team comprises of a group of distinctive and independent personalities, we cannot generalise very easily about how these personalities will interact and how the team will progress. However, there are some simple basic rules which seem to work and the purpose of this short chapter is to describe these.

2. Setting up a team.

This may not be an issue since your instructor may allocate you to a team without providing any choice in the matter. This is a reasonable reflection of what happens in industry so one can't really complain. However, if that is not the case and you are asked to form yourselves into teams here are some pointers to doing that.

Suppose that we are trying to form a team of 4 or 5. We need to look for a blend of personalities and skills that will knit together and produce an effective force. The nature of the project may determine some of the parameters but let us assume that all the potential candidates for the team are reasonably well prepared in terms of having progressed satisfactorily through the programming, design, and more specialised courses needed for the project.

The key requirement is for the team to be people who get along reasonably well with each other, who can meet in suitable places and who all have a similar interest in doing well in the project - partly because of the desire to get a good grade in the exercise.

A software project involves a number of key activities and it is important that there are members of the team who can make useful contributions to these activities. We need some good programmers but there are many other things to be done in the project, we need people who have abilities to organise, to plan, to negotiate and communicate - with, for example, the client - and there is always the need to document clearly and systematically various important things.

No single person will come to the project with all these abilities to a high level but most people will be capable of most to some extent. Extreme Programming emphasises the equal involvement of all team members in all the important activities and so the project will be a framework within which all team members will develop significant skills across the board. You will learn both technical material and skills as well as how to co-operate, communicate, organise, resolve problems, deliver a successful product and mature as a software professional in a way that is just not possible in other types of learning.

In situations where there is an odd number of team members it makes pair programming awkward to organise. It doesn't really work with three people round a machine so the best way to operate if you have a team of 5 is to have two pairs doing programming, testing and debugging and the other person can do a variety of tasks such as reviewing code, documents and models, system testing, preparing documentation and manuals, etc. The important thing is to keep changing the pairs around, involve everyone equally in contributing to the project and to keep talking to each other.

Doing a *real* project with a *real* business client will change you forever, your perspective on life, on your colleagues and on the process of working together. Your understanding of the profession of a software engineer will be transformed, as will your job prospects, since future interviewers will be really impressed by your experiences in doing real software development, it will set you apart from the rest of the applicants as someone with extra skills and experiences and value for their business.

A skills audit is an important feature of any team building exercise. It may only happen in an informal way, you gradually learn what your colleagues know and can do. It is best, initially, however, to try to write down what your strengths and weaknesses are and to share this with the rest of the potential team. See if there are people with a good selection of the skills needed. If most of your team are good at programming but not very good at talking to people or organising documentation then that should be a cue to try to recruit someone with these skills. The deal is then that the Extreme Programming approach will help them to develop those skills that they are weak in. Extreme Programming is all about multi-skilling and learning all the key skills needed in software development to a high level.

Itemise your groups relevant skills - or at least your own assessment of them in a table like the following :

skill	excellent	moderate	limited
Programming in Java	pete, mary	joe, oscar	jane
Programming in PHP	jane		oscar

skill	excellent	moderate	limited
Communications skills	oscar, joe	mary, jane	pete
Organisational skills	mary	pete, oscar	joe, jane
Documenting skills	jane, oscar	jane	pete, joe
...			

Table 1. A skills log.

This is only a rough guide and the definition of the skills and levels is bound to be vague but it does give you some basis to plan out your project and also a simple benchmark to compare with at the end of the project. One would hope that there is a significant improvement across the board by the end of the course.

3. Finding and keeping a client.

It may be the case that your instructor has found a suitable client with a realistic problem for you to tackle. Ideally the client should not be in your academic department but from outside, either from an external business or other organisation or perhaps from another department within the University. It may be more realistic for your instructor to organise a collection of projects which involve a member of the staff of the department acting as a client. Much can be learned from this experience but a real client provides that unpredictability that XP should be able to handle.

If you have to find your own client, and this is perfectly possible, then there are a number of avenues worth exploring.

Check out your family and friends. It is likely that you know someone who has a small business or who works for a local organisation. See if they would like some high quality software developed exclusively for them at a nominal cost.

Contact the local Chamber of Commerce or similar business organisation.

Approach local charities, these often have interesting and useful problems and cannot always afford to get professional software created for them.

Talk to staff in other parts of the University, this is always a rich source of good projects, in my experience.

Examples of systems that could be useful include:

databases for customers, orders, and other relevant information;

web pages with some useful functionality, perhaps allowing customers to request products, catalogues, or to supply information such as customer details, market surveys etc.;

planning tools which might enable an organisation to organise its resources better, time tabling some of its activities in a more effective way;

there are many other applications that you could consider.

Once you have identified a potential client it is important to establish the following:

is the client prepared to give enough of their time to meet you and identify what it is they require in detail as well as to evaluate your software over the period of the project?

As we shall see later, Extreme Programming requires a very close interaction with the client, if the client cannot afford the time required, say a couple of hours a week for a semester, then look for another client. If the client does not operate locally this could also be a problem.

Having identified a client and a potential problem see your instructor to find out if they think it is appropriate for your capabilities. Your instructor may have originally intended you to do a team project that they had made up. Argue the case that it would be much better for you if you could do a *real* project instead. It would also be much better for your client. Even if you are not totally successful in building a complete solution your client would have learnt a lot about their own business or organisation simply because your questions would have made them think about what they do in a fresh light. It may be possible for an incomplete system to be completed by some of the team during the vacation. Everyone will benefit, even your instructor. It is so much better if your efforts are directed at building something that will be useful to someone rather than something that, once it has been marked, will be thrown away!

You will also learn so much about dealing with a client, about delivering a quality solution and about planning and organising yourselves because you will be better motivated compared with the traditional sort of projects that professors dream up. You cannot learn many of these things from lectures or books, you must learn by doing it all *for real*.

Once you have a client and a project it is vital that you make efforts to keep them both. Regular feedback to the client is essential, so regular meetings must be held. When you attend these meetings make sure that you approach them in a professional way. Think smart and look smart. Give your client confidence that his/her investment will be worth while and they will get something out of the exercise. Never break appointments, if some other crisis occurs it is vital that the client is warned if it is necessary to change a planned meeting.

Always describe what you have achieved since the last meeting. Always appear interested in the client's business, and express some confidence about how the project is going but do not exaggerate progress. Honesty will ultimately pay.

My experience has been that clients really enjoy the activity, many have never been a client for a software development project before and they are getting some useful insights that may be valuable in later years. They also generally like working with bright and enthusiastic young people. So for them it will be both an enjoyable and a productive experience. It should be the same for you.

4. The organisational framework.

We will now assume that you have been allocated to a team or have organised one yourself. We now describe a few simple, and perhaps obvious, things to do. Do not underestimate these factors, many projects fail because of the simplest and most stupid of mistakes and omissions.

Learn as much about your team members as possible, their names, addresses, phone numbers, e-mail addresses and so on. It is vital that you can contact everyone easily because you will be working on the project in a variety of locations, not just the usual laboratories. This is something that is different to most industrial practice where the team occupies the same premises all day and every day. See if everyone will sign up to a working agreement that identifies the responsibilities and expectations of all the team members.

Agree on the location for the first meeting and make sure everyone turns up on time. This is important if one wants to be treated professionally, as your client will want to do, if you do not behave in a professional way why should anyone treat you like a professional? This is the first test, if a team member does not make it to an important and agreed meeting and they do not have a excellent reason, then this is a major threat to the project and to all of the team's grades. The team agreement should emphasise the obligation on all team members to attend all meetings. If the culprit does not listen to reason then complain to the instructor.

Teams can work well in a variety of ways. Sometimes it is worth agreeing on having a team leader who takes over the responsibilities of organising and chairing meetings, of leading the planning and other key co-ordinating activities. If everyone is happy with this solution then this can work. My recommendation, however, is for the role to be shared, each member of the team taking over the running of the team for, say, two weeks at a time. Thus everyone gets an opportunity to develop their leadership skills and to take responsibility for the team's progress. This is more in keeping with the democratic nature of Extreme Programming.

It is important to establish an effective method of working. First of all you will need to hold planning and progress meetings. Depending on the time scale and your other activities there might be several of these each week. The current project leader should chair the session. There should be another team member to act as secretary - this could be the person who will take over as project leader after the current one. The meetings should be minuted formally. This requires the following information about the meeting to be recorded:

Date;
Location;
Attendance;
Absences (with reason);

and then the record of the meeting.

See Figure 1 for an example of a template that works.

Each item of discussion should be numbered and a brief description of the item made. Any conclusions and decisions taken must be recorded together with any further actions agreed. These must describe:

what is to be done;

who is to do it and
when it must be done by.

All this is absolutely vital if the project is not to suffer from confusion and recriminations.

Each team should appoint an archivist. This role can also be shared around the team. The key requirement is that someone is given the responsibility to maintain a complete and accurate record of the plans and meetings of the project. This person should set up a suitable filestore on some server where all the team has access so that anyone can consult the archive to see what the status and history of the project is. We will later discuss the archiving of other, more technical documents, requirements documents, test cases, code, etc. The regime for these documents is different, however.

Another important activity is the recording of the amount of time each team member spends per week on the project activities. This should be recorded on a weekly time sheet for the team. Examples will be found in a later chapter.

It is vital that we record accurately the time we spend on projects.

Firstly it enables us to track our individual performance and helps us to identify where we are making progress and where we may still have improvements to make as we undertake various types of activity in the software development process. This is vital for apprentice software engineering and, in fact, should be something that we do throughout our professional lives. The Personal Software Process [Humph1996] provides an excellent framework for this.

Secondly it will help us to collect data from which we can predict how much effort future activities might take. Estimating the resources - time, people etc. - needed for the development of software is notoriously difficult. Many decisions are made in an *ad hoc* manner and usually lead to disaster or, at best, to a very inefficient and expensive process. We have to learn to do better. As we will see, later, planning is an important part of Extreme Programming.

Minutes of group meetings.

Group no. …….
Date of meeting [dd/mm]…….Time of meeting [hh:mm]……... Place of meeting…….
Present…….
Absent (reason)……...

Agenda item	**Details:**	**Action by:**	**Deadline:**
1			
2			

3			
4			
5			
6			

Figure 1. A template for the minutes of a meeting.

5. Planning.

The basics of planning include:

decomposing the overall task into a collection of smaller tasks;
identifying the dependencies and relationships between these tasks;
estimating the amount of resource (time and manpower) required to complete the tasks;
setting delivery times for each task;
describing the plan in some suitable notation.

Plans will inevitably require review and alteration since the estimates made of the time needed to complete tasks is often wrong, further understanding of the project could lead to a different structure to the previous task decomposition as well as exceptional circumstances, such as ill-ness, intervening. The regular meetings provide an opportunity to review and re plan the project. Do not shy away from hard decisions in these meetings. It is very easy to pretend that everything is all right when it isn't. Equally one can get depressed about progress. Later in this book we will look at planning again and examine how Extreme Programming can provide some answers to some of the problems met in planning and running software projects.

We now look at some planning techniques. There is software that is widely available for project planning, however, this is often much more complex than is needed here.

It is necessary to split each phase down into activities at a level where these can be assigned to individual team pairs, or possibly, a larger group of team members. It is then necessary to mon-itor how progress is made with each of these activities, and to do this one needs a schedule describing which activities are to be undertaken when. Some activities will be pre-requisites for others, in the sense that one activity will depend on the output of a previous one.

5.1. PERT (Programme Evaluation and Review Technique).
This is a technique that enables a schedule to be constructed that meets all the constraints of

the pre-requisites and identifies the *critical path* through the programme. The critical path is, the part of the schedule which determines the minimum time in which the whole project can be completed. It allows us to identify the activities which are most important in terms of their effect on the overall timing of the programme, and hence to identify those which need to be monitored most carefully.

The basis of PERT is a graphical representation of the activities known as a PERT chart. This diagram consists of nodes to represent activities, which is annotated both with the name of the activity and with its duration (in whatever time units are being used: typically days, weeks or months). Where one activity is a direct pre-requisite for another there is a directed arc from the earlier to the later node. There are also two special nodes, one for the start of the programme and one for its finish. A typical example of such a graph is given in Figure 2, which follows from the description of PERT in Boehm, [Boehm1981].

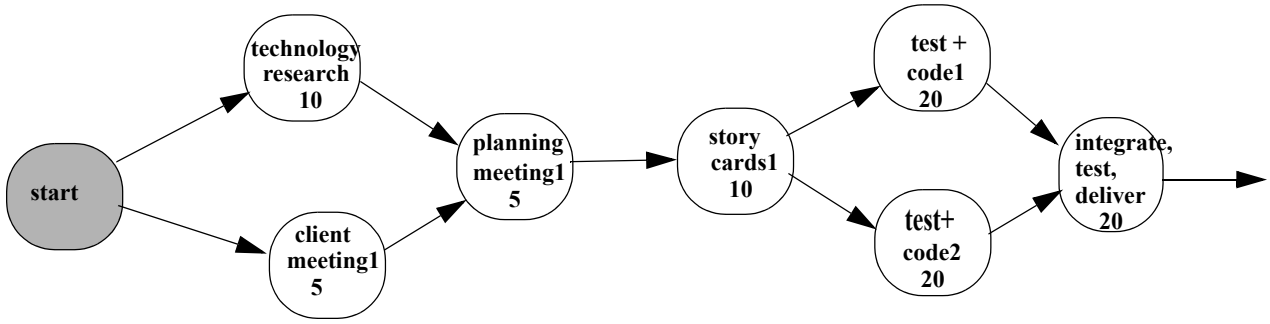


Figure 2. A PERT chart for a project.

In this chart (Figure 2) we have taken a rather *simplistic* view of the XP process. There is likely to be a lot more iteration and the chart will be much more extensive in most cases. The numbers refer to person-hours of work and are just crude estimates, we would expect the activities to be much shorter in a full time XP project. This plan tries to take into account the fact that most student team members will have to attend other classes and activities outside of the project. This needs to be taken account of sensibly.

In constructing a PERT chart it is often easier to start at the finish and work backwards rather than start at the start and work forwards, this way we can try to ensure that all the prerequisites for a node are identified and added to the chart. Even so, it is common that such a chart may need to change as the project develops and additional nodes are identified or different activities are found to be necessary.

In traditional approaches, once the chart has been constructed it is used to determine the critical path, that is the path for the project which will take the longest time.Here the situation is much more dynamic and fluid and the role of the chart is merely to identify potential problems.

In the example above, there is a potential issue in that the team carrying out the technology research might hold the rest up at the planning meeting. It might thus be sensible to involve more people in this aspect, but not loosing site of the problem that too many people working in an uncoordinated way is not only inefficient but it is also a cause of potential team rows if some members feel that they have been wasting their time carrying out work that is not very useful to the project or has been duplicated by others.

A more natural way to illustrate an XP project, particularly the iterative cycle is given in Figure 3. These deals with the weekly cycle of activity, assuming that you can only meet the client once a week. If the client can be involved more then that is fine but it isn't usually possible.

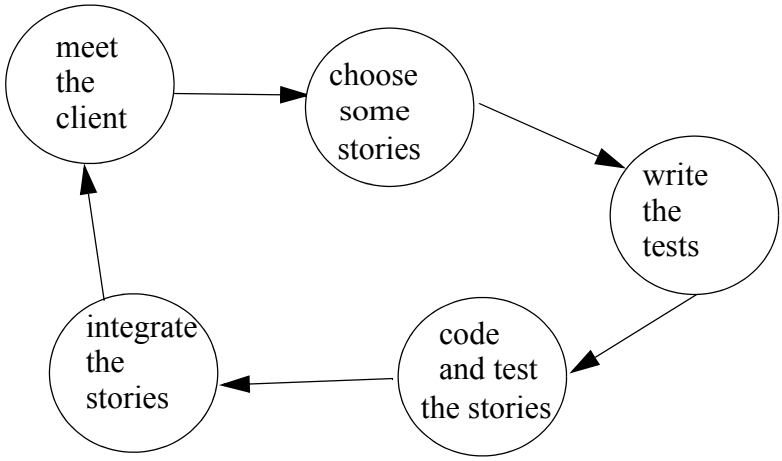


Figure 3. The weekly XP cycle

5.2. Gantt Charts

While the PERT chart displays the relationships between the timing of the different activities, a complicated PERT chart can be difficult to read in terms of deciding which activities will actually be scheduled when during the periods when they can occur. For this reason it is sometime useful to derive from the PERT chart an alternative representation of the schedule known as a Gantt chart. This simply consists of a column for each time period and a row for each activity, with a line drawn on the chart to indicate when that activity is scheduled to take place. The beginning and end of each activity is usually marked with a triangle, and if the chart is being used to monitor actual progress then it is conventional to fill in the triangles as the activities are actually started and completed. One possible Gantt chart for the project illustrated in Figure 2 is given below. It is clear from this that it is not particularly useful for an XP project.

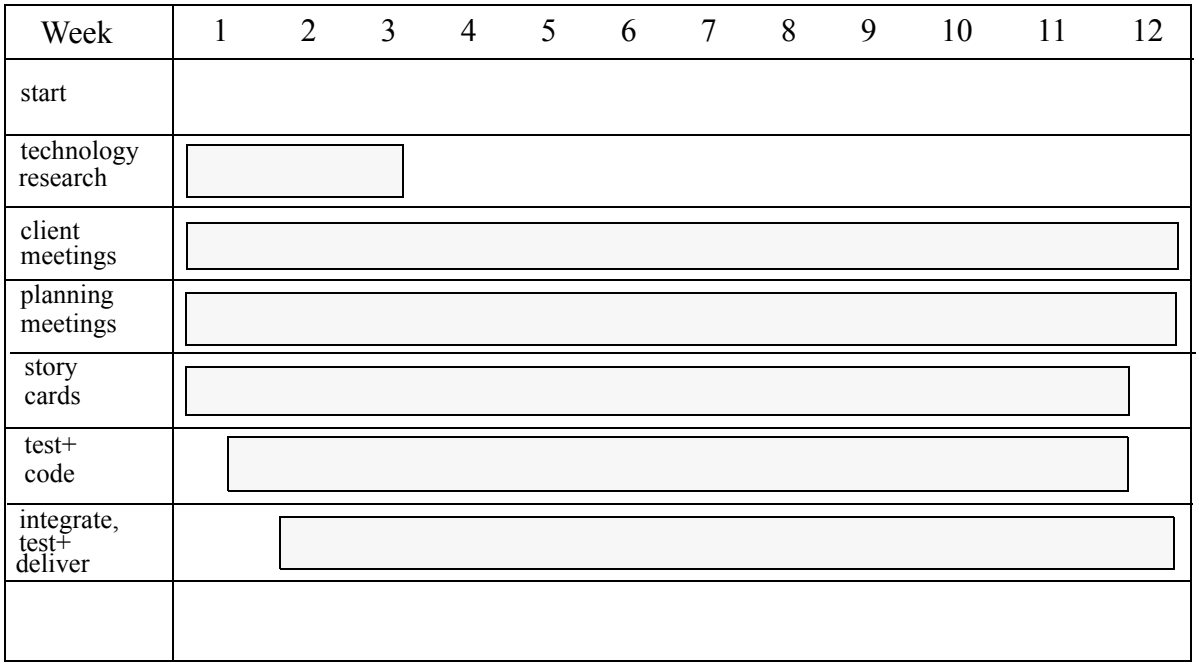


Figure 4. A GANTT chart.

There are a number of tools that help to create and maintain these charts. They may be worth investigating but many are more complex than we usually require and they do not fit the highly iterative nature of an XP project.

It would be better to produce a simple table of the weeks activities.

Week	Activity	Comments
1	Meet the client, carry out some research, identify some simple stories, write some tests, write the code, produce some simple architectures and screen ideas,	Mary and Pete to do the research, everyone else to do the other tasks
2	Meet the client, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Swap round who does the research, everyone involved in everything else
3	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
4	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything

Week	Activity	Comments
5	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far. Produce a summary requirements document, (list of stories to be built, non-functional requirements, glossary)	Everyone involved in everything
6	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
7	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
8	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
9	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
10	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
11	Meet the client, demonstrate code, discuss ideas, identify and prioritise some stories, estimate, test and code stories, further research, refine screens etc.. Integrate stories so far.	Everyone involved in everything
12	Prepare for the final handover. At this stage we should be doing only minor changes to the system, fixing problems etc. Write basic user and maintenance documentation. Update the requirements document.	Everyone involved in everything

Table 2. A project activity table

Now we have to make an allocation of team members, usually pairs, to the various tasks identified. This could be done by a more detailed table. It is likely that this table will need to be revised on a regular basis as the project progresses. In most cases the length of the project is fixed and this provides a major constraint. You may have to compromise on what you are hoping to build, better to build a simple system that works than a fancy one that doesn't - your client won't thank you for that!

6. **Dealing with problems.**

It is inevitable that things will go wrong from time to time. It is the teams that are able to deal with problems effectively that turn out to be successful. It is not about how clever people in the team are but the culture within the team. If this culture is one of co-operation, discussion, build-

ing consensus and treating each member as an intelligent individual with a legitimate point of view then resolutions can be found. If people are stubborn, arrogant, dismissive and unco-operative then it is much harder. Try not to loose one's temper, be patient and considerate to others, discuss the issues on the basis of an informed knowledge of the matters under discussion rather than based on prejudice and guesswork. Seek expert advice if the argument is about a technical point, the benefits of different strategies or approaches. Talk to the client as well. All these things can be sorted out if everyone is positive and prepared to give and take. Extreme programming is all about co-operation, communication and treating people with respect and trust. All problems are soluble somehow.

If you really hit a crisis and there seems no way out seek arbitration. Find someone that everyone in the team respects, perhaps a tutor or professor, and explain the issues to them and ask them to make a judgement. This might be a simple compromise that everyone was too uptight to see or it might be a ruling in favour of one side or another.

Try not to be upset if your argument is not the one that is successful in this process. Think about what has happened and see how you might benefit from the experience. Perhaps the way you handled your argument or yourself was counter-productive. Successful people in life reflect on their experiences and learn from them, adapting their future behaviour in order to ensure future success.

Sometimes you get into an argument where nobody is prepared to give way. This can happen if you are just working as a pair or it might occur with more people in the team involved. A simple suggestion from [Miller2002] might be useful. He discusses the issue in terms of pair programming but the technique can be extended to bigger groups.

First, every person involved has to ensure that they understand the conflicting opinions.
Then each person is asked to rank his/her opinion on a scale of 1 to 3 with 1 meaning "I don't really care" to 3 meaning "I'll quit if I don't get my way". So 2 could be "I am interested in this argument and I am prepared to spend some time looking into it, I want to hear your point of view but I will take some convincing that it is better than mine".
If there is a highest ranked option then that is what is pursued until evidence emerges that it might not be the best.
If there is a tie then you can pick a direction at random if the scores are low. If both views are ranked at 2 then it needs more time to research and analyse the issue. A trick here might be for each individual to try to make the best case they can for the *opposing* view. If that doesn't lead to a preferred option then either ask a third party or spend a little time taking forward both ideas until a clearer position and, hopefully, a consensus emerges.

Failing all of this then seek advice from the project supervisor.

Sometimes differences are not as real as they seem and are mainly due to poor communication and a lack of understanding of what each other mean. These problems should resolve themselves if you try to listen carefully to what the others are saying, perhaps even writing it all down, this act often forces you to be a little more precise than before and can be the key to clarity on both sides.

7. Risk analysis.

Projects can always go wrong. One way to minimise the impact of this is to carry out a risk analysis. This involves and identification of the *hazards* (things that can go wrong), and their associated *risks* (estimates of the probability of those hazards occurring, and the likely severity of the consequences).

There are many hazards including:

technical hazards - using the wrong technology (one that cannot be used to solve the problem) or one that the team is insufficiently experienced with;
planning hazards - the software being developed is too complex for the resources available and the project plans are far too ambitious;
personnel hazards - some of the team members are not capable of delivering, perhaps they are lazy and poorly motivated or perhaps their technical knowledge is weak.
client hazards - the client is too busy or lacks interest in the project, the client is trying to exploit the team by demanding too much for too little.

These hazards relate to the project and its overall management. There are other hazards in the form of delivering an unacceptable final product. XP tries to deal with this by encouraging the frequent releases and close client contact. Even this may still fail to prevent problems. Many failures are due to the non-functional attributes not being met, [Gilb1988].

In order to prevent problems with the non-functional or quality attributes it is important that these are identified clearly and precisely and means for testing for compliance developed. This will be a concern of a later Chapter. We need to be able to estimate the likely range of variation in these attributes, and realise that the risks to the success of a project come essentially from the possibility of actual attribute values finishing up outside the specified range. Thus, the risk to a project must be controlled and we need to find solutions which will meet at least the minimum required levels for all the critical attributes.

Part of this risk control process therefore involves identifying which attributes pose the greatest risk to the project, and this comes in two forms. Some attributes will be mandatory and others merely desirable. Clearly we need to focus on the former for most of the time. These critical attributes have to monitored.

8. Review.

This chapter has tried to provide some practical guidance about how to organise your XP project team. Most of it is just common sense but it is surprising how often these simple practices get forgotten in the heat of the moment. By forcing yourselves to act professionally, to document and plan your approach you should avoid many of the common pitfalls that so bedevil software development projects.

Don’t assume that, because you have been made aware of potential pitfalls and ways to avoid them that everything will be plain sailing. There will be problems, some of these will be down to poor organisation, not planning the project properly and delivering what is needed at a time and to a satisfactory level of quality. However some problems may be beyond your control. Perhaps the client hasn’t given you the correct information or hasn’t reviewed your ideas quickly enough. Perhaps team members have been ill. There is not much you can do about the latter except try to adapt the project be reorganising the plans and team activities.mSometimes, howev-

er, problems arise because of personal differences and lack of interest or commitment amongst the team. There is no easy solution to this, discussion on the basis of a friendly meeting, perhaps held away from the lab, might be useful. Building a pleasant social atmosphere in the group can be helpful. One senior developer, who is often called in to rescue problem projects in his company said: “Projects must party”, meaning that spending some time relaxing together, perhaps over a meal or a drink, or some other outing, pays large dividends in terms of morale. many problems in software engineering are human and social ones and should not be ignored.

Exercises.

1. Meet with your team members and agree on a mode of working -
   where and when will you meet,
   decide on individual responsibilities e.g. who is responsible for archiving the documentation, chairing meetings maintaining the project plan etc.
2. Read one or more articles on project management - these are readily available. Research into the question - *why do software projects fail?* Identify some of the possible pitfalls that your project might suffer from, what are you going to do to avoid these?
3. Develop PERT or Gaunt charts for the project, to cover at least the first few weeks.
4. Carry out some risk analysis - how can you control and minimise these risks?

Conundrum.

Your project involves programming in a language which is familiar to only one member of your team. Two others have a slight knowledge of the language but have never written anything serious in it. You are trying to do pair programming but the ‘expert’ is getting frustrated because whenever she is paired with another team member progress is very slow (because much of the time is taken up with explanations of what she things is obvious}. She feels that it would be better if she worked on her own on the program and the other team members did other things, such as writing documentation and testing.