The book of

A practical guide to agile software engineering in an industrial context

Mike Holcombe

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 14 years of experience of teaching in this way and managing such projects. In all I have managed around 50 projects involving a couple of hundred teams, of which 49 have tried to use XP in the last three 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. Life is never as simple as most writers seem to imagine, sticking to *pure* XP is rarely going to work in most projects but by adapting it to suit the context - both in terms of clients and developers, has proved extraordinary effective;

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, Andres Barravalle 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.

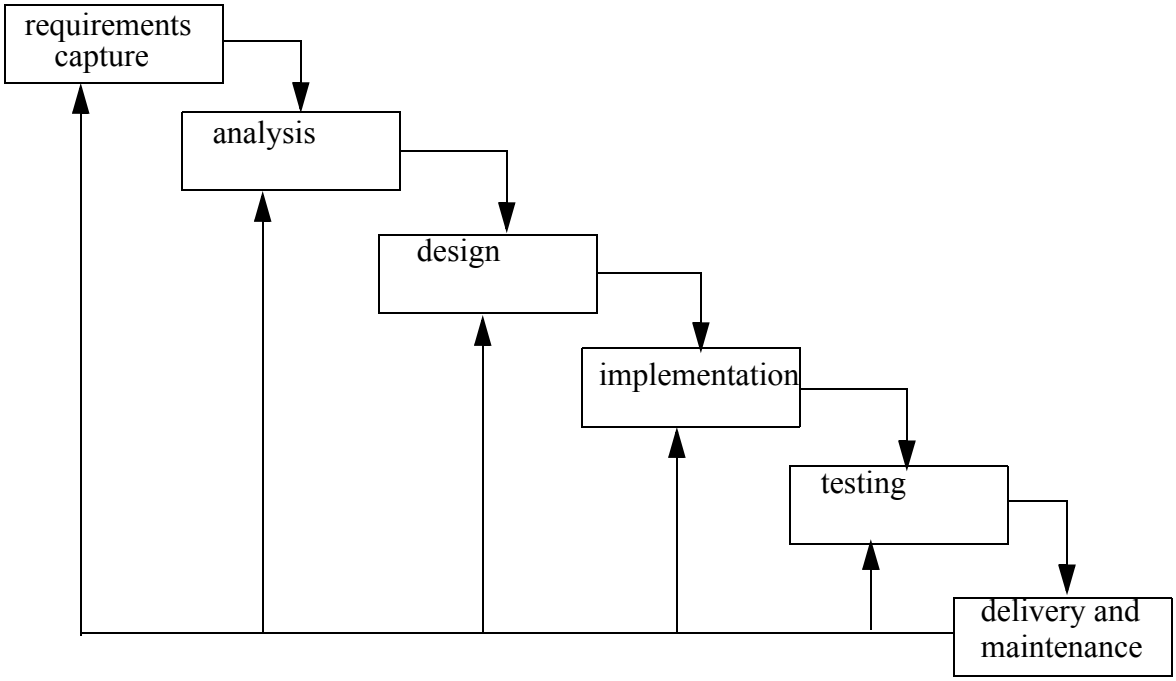


Figure 1. The Waterfall model of software development.

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*.

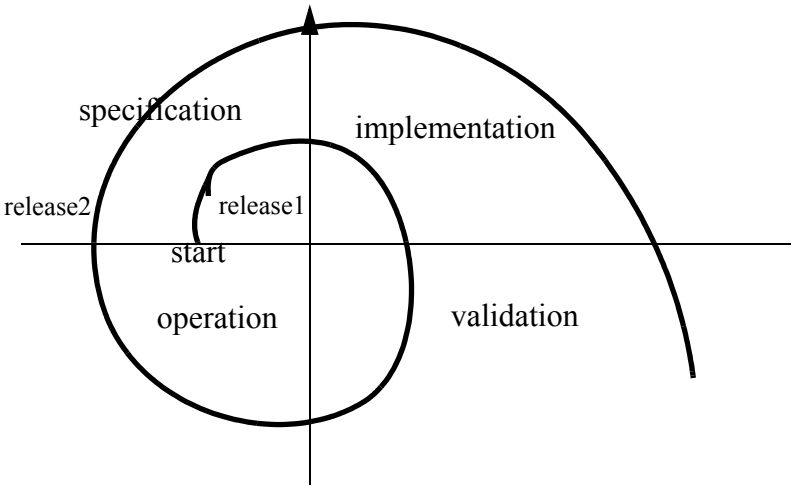


Figure 2. The Spiral model

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.

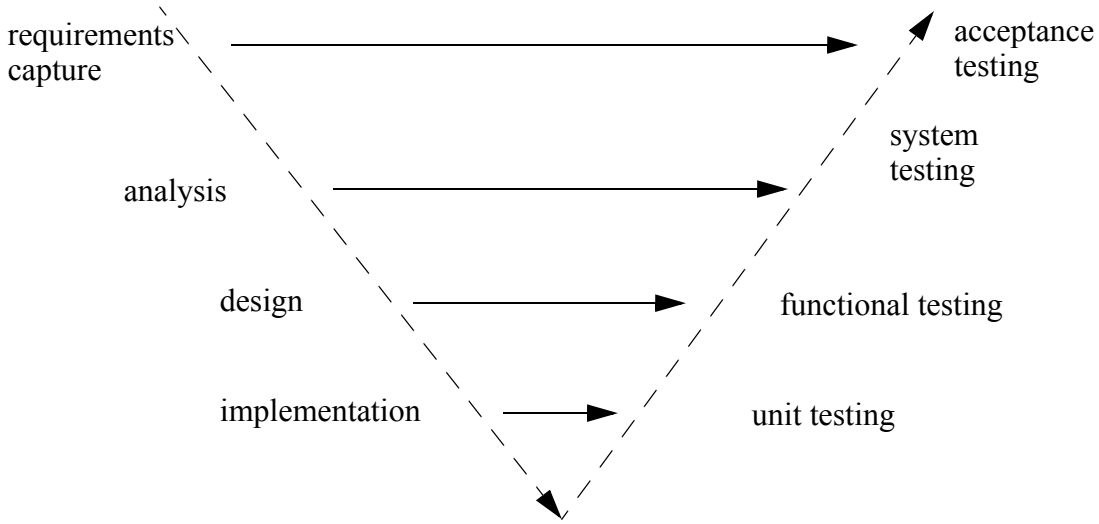


Figure 3. The V model

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.

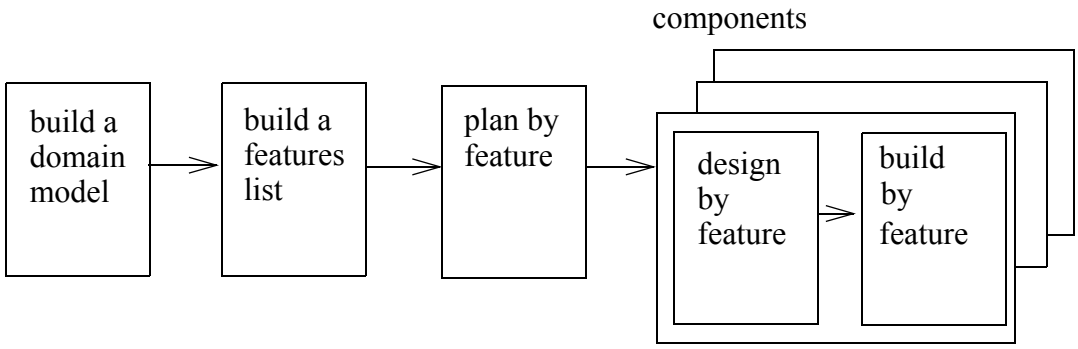


Fig. 2. Feature driven design.

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

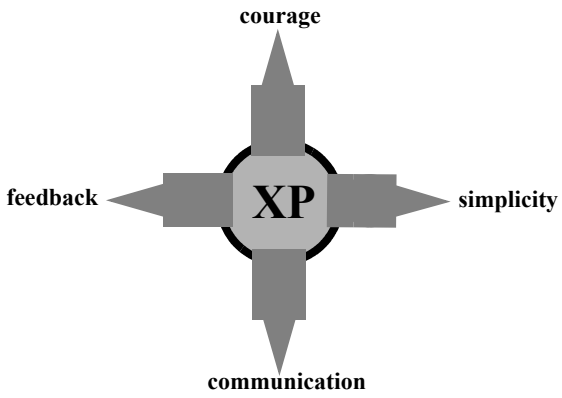
EXTREME PROGRAMMING outlined

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.

We have been trying to find a simple diagrammatic method which shows how the system

hangs together, which illustrates the flow of the processes, is understandable to both developers and customers and can adapt to changing requirements. Our research into the cognitive porocesses involved in design and requirements modelling have shown that the generalised machine model, the X-machine, works extremely well. In our recent industrial projects we have used it extensively and the results have been very encouraging. We will discuss this in chapter 5.

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.

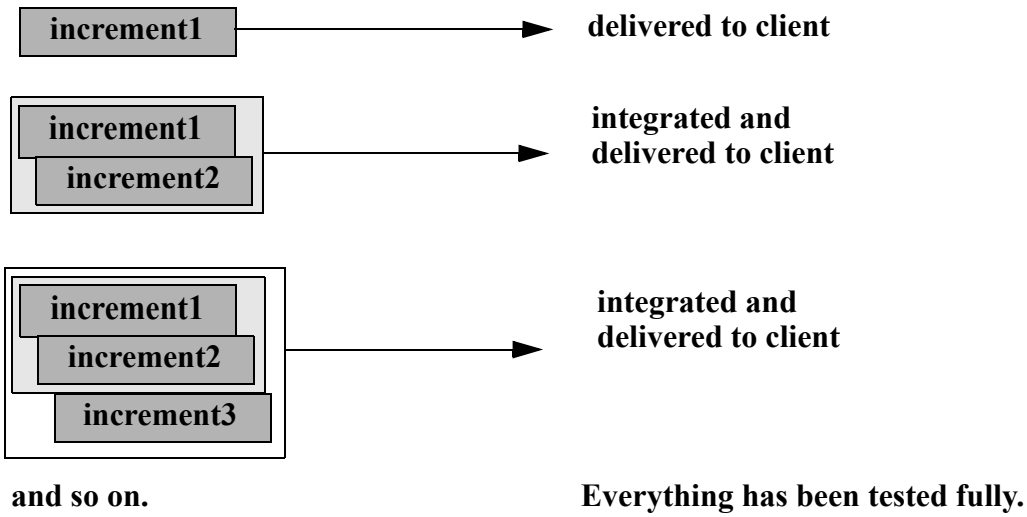


Figure 1. Incremental delivery.

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 new 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 hun-

dreds 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].

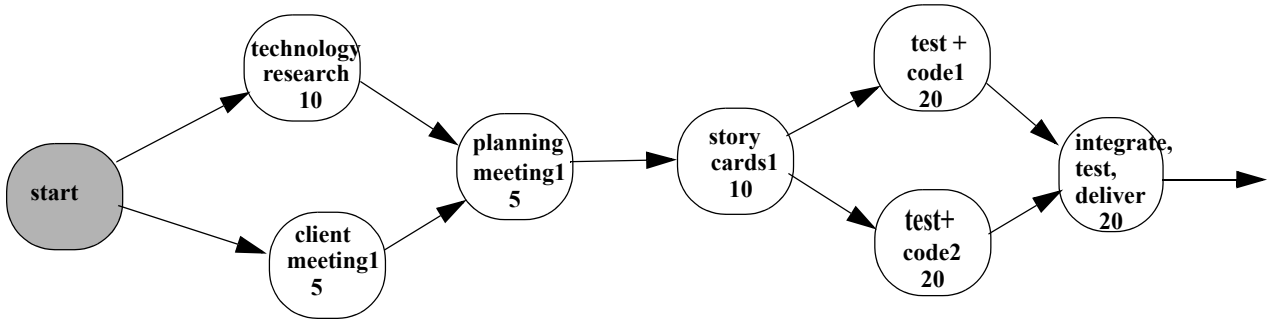


Figure 2. A PERT chart for a project.

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.

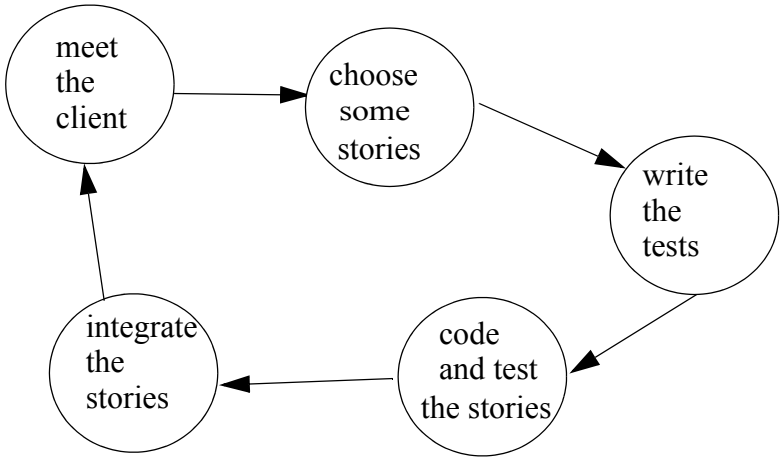


Figure 3. The weekly XP cycle

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.

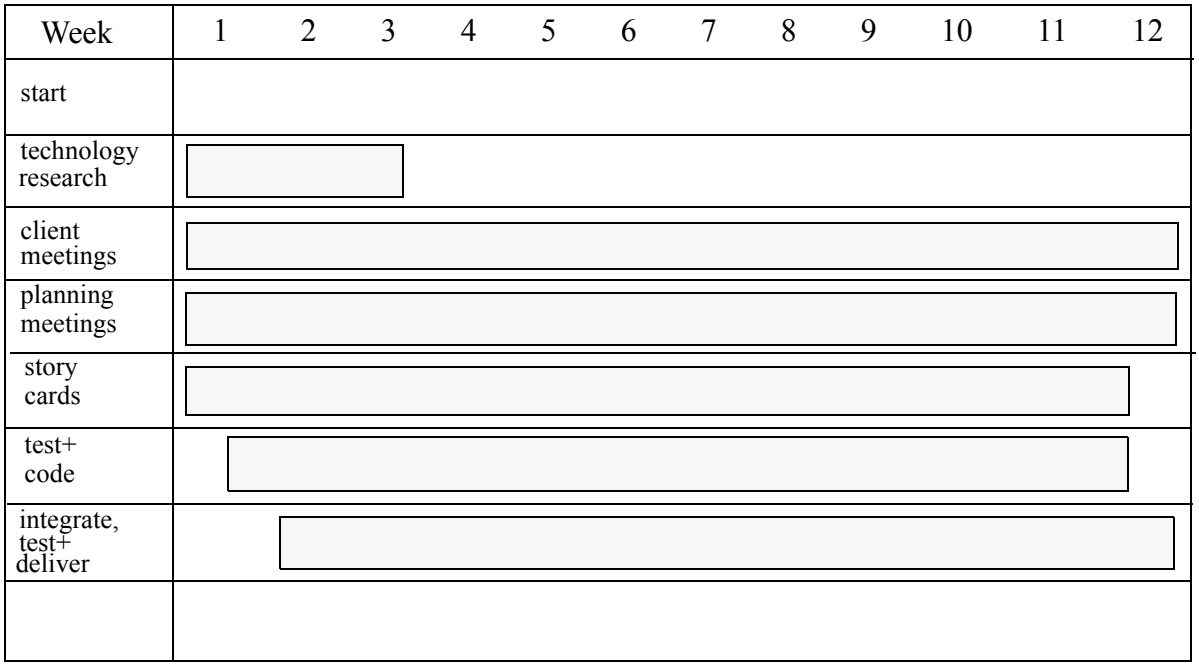


Figure 4. A GANTT chart.

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.

How should you deal with the situation?

For a discussion of this see Chapter 11.

*References***.**

[Boehm1981]. B. Boehm, "*Software engineering economics*", Prentice-Hall, 1981.
[Gilb1988]. T. Gilb, *Principles of software engineering managemen*t, edited by Susannah Finzi. - Wokingham : Addison-Wesley, 1988.
[Humph1996.]. W. S. Humphrey, "*A Discipline for Software Engineering*", Addison-Wesley, 1996.
[Miller2002]. R. Miller, "When pairs disagree, 1-2-3", in D. Wells & L. Williams (eds), *XP/Agile Universe 2002*, Springer-Verlag, Lecture Notes in Computer Science, vol. 2418, pp. 231 - 236, 2002.

Chapter 4

Starting an XP project

Summary: Meeting the client or customer. The first attempt at defining the scope of the project. Some techniques for requirements elicitation. Basic business analysis. Functional and non-functional requirements. Identifying dependencies and constraints. The structure of a traditional requirements document. An example of a real requirements document from a project.

1. Project beginnings.

It is the first class for the project and you have now got a client and a project. Initially, it all seems very daunting and many students are pessimistic about being able to build something that looks very complicated with a technology or method that is unfamiliar. You will almost certainly succeed if the precautions that I have indicated in earlier chapters are taken.[1] If there is a failure in a team it is because of individual failures or a breakdown in communication, it is rarely due to the teams being technically or intellectually unable to cope. It does depend, of course, on the project scope being appropriate, neither too hard or to easy and this requires some judgement and experience on the part of the tutor.

We will assume that you are starting with a requirements which includes a reasonably simple initial phase and you should confirm with your client and/or tutor whether there is a part of the system which is clearly within your capabilities and which can be addressed first in order to gain confidence. In fact, the XP approach of building things in stages and getting them to work properly will soon build up your confidence.

The initial project description may be nothing more than a paragraph and it might seem to be too vague to allow you to start. Remember, however, that this description is just a starting point to you exploring, with the client, the client's business, its needs and possible solutions and so there will be a lot of preliminary work to do to define the scope of the project and what a potential solution might look like.

This is not an easy stage of any project and it is impossible to learn exactly how to do it in books and lectures, there is no substitute for trying it out and reflecting, as you go, on how the process proceeds.

We will look at the initial descriptions of two real projects that were done by students in 2001 and see how one of them progressed to a complete solution using XP.

Project 1. Quizmaster.

Background.

The brief was given to design a questionnaire-generating programme for an organisation that provides training for lawyers. Its main purpose is to allow trainees to answer weekly exercises for specific topics using a computer as an aid. This is in contrast to the approach used previously where exercises are handed out on paper, the student answers the questions and this is marked manually.

Project description: The proposed system should automate this task by removing the need for the lecturer to do any marking or for the students to mark their own work. The student instead answers the exercise on the computer

1. If it is any consolation I have run such projects for a dozen years or so and it always seems to work out.

and a mark is returned immediately. The system also monitors certain statistics on the student's performance on these tests so that lecturers can see how individual students are progressing and if they have been doing the exercises that they should have been doing. The system is for use on the Legal Practice Centre course that typically has 120 students for the duration of a one year long course consisting of two semesters. Each student takes four compulsory topics during Semester One and three chosen topics during the second semester. The chosen topics for the second semester are not known at the start of the year.

The system must allow for different topics and for topics to be changed, so it has to be possible to add and remove topics. Also, since the students on the course will change it will also be necessary for the system to add and remove students, especially at the end of the course when all students will need to be removed. The system must maintain details of topics and their exercises, students and the topics they take and the required statistics for how students have performed on certain topics. This information will change often so it must be easy to change any of the information stored by the system.

Project 2. WASTETEC.

Background.

The client develops solutions for waste problems facing small and medium sized businesses in and around South Yorkshire. They act as a broker between two companies, one with unwanted waste and another requiring waste. The client does not charge for their service; they are part-financed by the European Community Regional Development Fund.

Current manual procedures: If a customer has waste to sell, he/she contacts the client and provides the necessary details. The client then adds this to the existing database. If another company wants to buy some waste, they approach the client with their requirements, and the client searches the database for any matches. If there is a match found, the client puts the two companies in contact with each other.

Project description: The client requires a WWW based search, enabling potential customers wanting to buy waste to get direct access to the relevant details of the database, without contacting the client, and releasing any company details. The client also requires the system to enable potential customers with waste to sell, to advertise their waste. If a customer finds a match for his/her requirements, they should then be able to make enquiries to the client, on-line. The client then follows this up separately, and is no longer a concern of the system.

As mentioned before these were actual projects carried out by 2nd year computer science and software engineering students at the University of Sheffield during the 2nd semester of 2000/2001.[1] We will follow the development of the Quizmaster project and the appendices contain more detailed information about the system that was built for this client.

2. The first meetings with the client.

This might be a session involving all of the teams working on that client's problem and generally involves the client giving a presentation about their business and what they are trying to achieve, their objectives for the system. There will usually be an opportunity to ask general questions relating to the system but these should not be technical computing questions - apart

1.In both cases the students involved in the projects, there were 6 teams of about 5 for each project, competed to build the best solution for the client. For each client 3 teams used XP and the others used the "traditional" UML design-led approach described in most software engineering text books. It was an interesting experiment to see which approach did best, if there was a clear difference. As it happened the best systems, as decided by the clients, were the XP systems. Furthermore the students who used XP found that it was much less stressful and enjoyable than those who used the traditional method. Some of the reasons for this have been discussed before and we will return to a reflection of the XP process in the last chapter. Be prepared to engage with this reflection process, yourself.

from general things such as the sort of network available or to be purchased for the solution. Remember that the client may know very little about Computer Science or programming, that's why they have come to you, you are the experts. Their expertise is in their business.

If you are not sharing your client with other teams then the first meeting will be a more informal one. It is important to prepare for it.

Starting with the initial project description there are a number of things you should do:

Research into your client's business:

have they got a web page?
can you find any other published information about their business?
what do they sell - products or services or what?
what sort of clients do they typically have?
who are their competitors, what can you find out about them?

Preparing carefully for the first meeting will impress the client that you are professionals and will give them the confidence to proceed, don't forget that they are giving up some of their time and this is a cost to their business.

Turn up looking smart, on time and at the right place. Do not chew gum or do any other things that would distract the client's into doubting whether your are worth working with. These first impressions are important, you may think that they are trivial or superficial issues but it is part of the business expectation that the clients will have. In later life you will have to recognise these things, anyway, so it might as well be now!

3. Initial stages of building a requirements document.

Although we will be using stories as the basis for the software development it is important to have a clearly structured list of requirements, both functional and non-functional, so that an overall description of the complete target system is available.

This is developed in discussions with the client. At a suitable point we will develop a system metaphor and extract stories from this requirements and start developing the system.

There are a number of reasons why a requirements document may be important. Your client will often want something that can be shown to his/her superiors in order to provide some indication of what is being developed. Naturally, this document may change during the course of the project. It will be built around the stories together with the non-functional requirements and other contextual information. Some in the XP community do not like the idea of a requirements document, seeing it as an unnecessary creation that is not in sympathy with *pure XP*. We have already commented on how some businesses will require such formal statements in order to approve things like project expenditure. The key thing is to be aware that it will change and to make sure that it is used as a summary of what the current knowledge of the proposed system is.

The process of carrying out a full *requirements analysis* in a traditional software engineering context is often done at the start of the project and only occasionally revisited in any fundamental way later. In XP we will have a more continuous dialogue with the client and so the full re-

quirements document will be a rather fluid document. We emphasise, here the construction of an initial one. It will contain, not only the functional requirements but also details of important *quality attributes* of the system.

Requirements analysis and specification is deceptively difficult since many clients don't know what they really want and they don't know what it costs or how long it will take to deliver. They often fail to recognise how hard it is to create a reliable system and how long it takes. Some might expect it to be done by next week!

Clients express problems naturally in their own words, words that might be unfamiliar to us or used in different ways, don't assume that your understanding of a particular word or term is the same as theirs. We need to identify what the terminology means and to agree on it. The construction of a *glossary* of business and technical terms should be an outcome of this dialogue.

When talking to clients realise that there may be hidden factors at stake: political, historical, geographical. You may need to understand these features of a business organisation in order to understand the reasons for particular requirements. Remember that there are probably more personnel involved in the business, who may have different requirements and different priorities, it is an important but delicate task to ascertain these.

The initial software requirements analysis can be divided into a number of activities:

Problem recognition;
Evaluation and synthesis;
Modelling and metaphor building;
Specification of user stories;
Review and discussion.

In the first week or two of the project, you should be evaluating and synthesising the problem and requirements information from the client. Always write down your thoughts, refer to these at your formal group meetings and put a date on them. Later, you may need to revisit some issue when you have forgotten the details. Although we wish to keep the paperwork to a minimum records of this stage should be saved, carefully.

You should already be modelling aspects of the client's business processes, in an attempt to clarify and make more specific your understanding of these processes. We will suggest a suitable way to help collect your thoughts together in the next Chapter.

You should also be writing code, trying to implement a few basic stories. Before you do this, however, it is necessary to write some simple tests so that you can see how your code works.

By the next week, you should be refining some of your ideas and developing code to give you and the client a better idea of what is needed.

Problem evaluation involves:

defining all external observable relevant business objects;
evaluating the flow and content of relevant information in the business;
defining and elaborating all relevant software functions;
understanding relevant business behaviour (events);

understanding user behaviour (tasks);
establishing systems interface characteristics;
uncovering additional constraints.

All of these activities are difficult and simple techniques that will always work are a chimera..

4. Techniques for requirements elicitation.

There are a number of useful approaches that can be used to elicit user requirements and to gain user involvement. Here are 6 approaches that can be useful:

Interviews;
Structured questionnaires;
Observation - again only successful, if you can do it unobtrusively;
Concurrent protocols - where a user describes his/her tasks whilst performing them;
Card sorting - useful if you want to understand the user's classification of his/her knowledge domain;
Carrying out a user role yourself;

Interviews have to be prepared carefully. In the first meeting, when you know little about the problem, then it is important to ask the client to describe all the key aspects of the system, try to guide them away from the desire to get to intricate detail about what they want when you simply do not understand what they are talking about. As you get immersed in their business context it is important to manage the meetings carefully. Identify what you want to know beforehand and prepare a set of questions that will help you to find out what you need. Once these questions are answered then you can explore further areas. It will often be the case that a question will stimulate the client into telling you some other piece of information, carefully record this. It is best to go to the meeting with all the team but make sure that there is a principal speaker and someone to record what is said. There is nothing more *off putting* for a client than to be faced with people asking questions from all angles on all sorts of disconnected topics. Plan your meeting carefully and try to stick to it. The same advice applies to any other stakeholder you meet, such as a user of the proposed system.

If you are not able to meet the client or the user then leaving a structured written *questionnaire* is another technique. Try to group related questions together. Also try to make your questions clear, unambiguous and relevant. Leave a contact number or email in case the person filling in the form has a query. Make sure that people know where to send the finished questionnaire and try to impress upon them, with tact, of course, that you need it by a specific date if the project is not to be held up.

Sometimes it is possible to visit the client and *observe* the business in action. Here you may be able to observe users in their current work. This is helpful in providing you with a context and a better idea of what the users are like, what they expect or are comfortable with and what sort of system you might be trying to emulate. Pay particular attention to the sort of user interfaces that seem popular. Take care not to disrupt their work too much. Some users are happy to talk their way through their tasks while you are there.

5. Putting your knowledge together.

Gathering all this information is one thing but putting it all together into a coherent model of the business is quite another. There are no simple solutions to this problem. Common sense is the best approach!

Defining all external observable relevant business objects.
We need to look at the sorts of things that are *coherent entities* in the part of the business we are considering. These could include: products, contracts, orders, invoices and such like. Make a proper list of them and try to distinguish between those that are involved with the external activities of the business, for example objects that are apparent to the customers, agents and suppliers of the business, and to those involved in the monitoring of the company such as taxation and other government authorities and the objects that are defined for the convenience of the internal management of the company, these might be: internal orders, memos and planning material, records and archives of company activity etc.

Evaluating the flow and content of relevant information in the business.
Each business process will involve a number of individual processes which take place in an organised way. What is the order in which this information is processed, what type of information is it? Try to get a general picture of what happens and when during typical scenarios of business activity. You will refer to the business objects described above, if you come across one that has not been identified, previously, then it needs to be added to the list. Equally, if you found an object that doesn't seem to feature in any process that you are analysing, eliminate it. It may be that you find some difficulty in modelling things at the right level, there is always the temptation to try to describe things in too much detail. Try to avoid this at this stage. We are looking for a rather "broad brush" description of what is going on.

Defining and elaborating all relevant software functions.
Now we can start imagining what our software is going to do. It might be replacing some existing function, either a manual operation or in some obsolete software, or it might be a new feature that has not been implemented with software before. We will come back to this process in Chapter 6, at the moment it suffices to write it down as clearly and concisely as possible.

Understanding relevant business behaviour (events).
Now we have to try to figure out how these things actually relate to each other. We should try to define some common scenarios which explain the overall operation of the business processes through the medium of identifying the events that cause the scenario to operate. These could be the placing of an order by a customer, here we might need to identify what sort of customer is involved, a new or existing one, trade or retail. The business process involved for each of these may be different and so the system will be expected to behave differently as well. This leads to us identifying the different conditions that must apply for the different cases. Again we need to check that our business objects and processes described above are consistent with this.

Understanding user behaviour with task analysis.
There are many techniques for task analysis which can be used to elicit user requirements relatively easily. Task analysis tends to concentrate on the way users conduct business processes now. It may include user actions which do not involve interaction with a computer. Nevertheless, a task analysis model can form a useful representation for discussion with your users, helping to identify aspects of the task with which users are comfortable and familiar with and which could be incorporated into the structure of interaction with the required system. Alternatively, it can help identify aspects of the task which are currently problematic and could be improved in the required system.

As if requirements capture and analysis were not sufficiently complicated, we must often obtain the views of different users, who are likely to have different stakes in the outcome of the new system. Hence, you may need to identify and resolve stakeholder views. You should ask yourselves, who are your users? They are not necessarily a single, homogeneous group of people with the same tasks, the same goals or the same view of the world who are the clients?

In Checkland's Soft Systems Methodology, [Checkland1990] a distinction is made between clients, who usually commission the system and stand to benefit from its outcomes, and actors.

Who are the actors? Actors are system users, who have to play a part in the system, but who may not directly benefit from it. How are you going to gain these stakeholders' involvement in and commitment to the development process?

At the end of this process you should have identified the dependencies that the solution needs to relate to within the business context as well as any basic assumptions that pertain. You may also have started to think about the constraints that will affect your solution, the available resources you have at your disposal, time, technology and so on. This needs to be clarified, it is no use trying to specify a system that you are not able to build.

A collection of *requirements notes* can be produced which can help to organise ones' thoughts into a more structured form. The aim is to produce a detailed list of requirements which provides a basis for early planning and approval.

The *functional* requirements should be stated, eventually, in a tabular form using simple English statements. These will be derived from the user stories as we will see in the next chapter. In fact, the functional requirements document is really just a summary of the story cards as they exist at the time. Where it is necessary to break a complex functional requirement down into a set of simpler ones, do so but try to preserve the connection between related requirements by grouping and numbering them together.

Looking at the Quizmaster system it is clear that one actor or user is the lecturer and they wish to be able to set tests. Now a test will consist of a number of questions and so the task of setting a test will involve the subtasks of setting a single question a number of times.

The requirement:

n. the lecturer can set a test

could then be restated as:

n.1 the lecturer can set a question

and

n.2 the lecturer can create a test from a number of questions.

If some of the requirements are poorly defined or subject to change identify them and put some measure on their risk of change, even if it is just a number 1...5 (low risk...high risk) which you allocate on the basis of your best guess given the knowledge you have available. We will find this useful, later.

We also classify each requirement as being:

mandatory - it must be present in the final solution;
desirable - it should be present if at all possible;
optional - only implemented if all others are done and there is still time.

Naturally the client will specify these levels and they may change during the course of the project.

We will use user story cards and their implementations as the main mechanism for determining detailed functional requirements. Once we have got a basic understanding of the system needed we can then start showing the client real examples of programs that will be the basis for further discussion at the regular (weekly, if possible) meetings. The organsation of these meetings is discussed in Chapter 5.

6. Specifying and measuring the quality attributes of the system.

We talk about two main types of requirements: *functional* and *non-functional*. Put simply the functional requirement describes *what* the system has to do and the non-functional describes *how well* it is supposed to do it. This is a little simplistic but it will do for a start.

We often put most emphasis on the functional requirements and neglect the non-functional or assume that they are easily dealt with. In fact, identifying the non-functional requirements can be difficult. We need to define them carefully and what is more we need to set some sort of acceptability levels for them and a means of demonstrating compliance with these levels. It is possible to refine the notion of non-functional requirements into two categories: *quality attributes*, which determine how well the system should perform and *resource attributes*, which constrain or limit the possible solutions to your business problem. Unless these are addressed a system may not be successful, even if all the functional requirements are met.

For most software systems some of these attributes will be **critical**, that is, unless each of those attributes achieves some required level then the system will probably not be successful, no matter how well it may meet its functional requirements or meet the goals for its other attributes. Thus it is essential to identify all the attributes, and to identify which ones are critical, and then to ensure that they are all met.

Identifying Attributes

The International Standards Organisation provides a taxonomy of quality attributes in its draft standard[1] for software systems, (ISO 9126). [ISO 9126]. As you read through the following list, based on that standard, make a note of those attributes which you feel could be critical to your project. The list is not exhaustive: you may see other classifications of qualities elsewhere and you may identify critical qualities for your system that do not appear here. Some of the issues that are discussed here may not be relevant to your own project. Think about them and focus on those that seem to be the most critical. Discuss this with your client. The ISO 9120 standard is

1. Some XP people may worry about the bureaucracy implied by the introduction of standards. However, companies purchasing and commissioning software are becoming increasingly aware of the standards issue and some will insist that these standards are addressed.

concerned with the quality of the product. There is another standard, ISO9001, which deals with the engineering process.

***Functionality*:**
Suitability - the presence of an appropriate set of functions for specified tasks
Accuracy - the presence of correct and predictable results from specified input
Interoperability - the ability to interact with other specified systems
Compliance - the adherence to specified standards, laws and regulations
Security - the ability to prevent unauthorised access to programs and data

***Reliability*:**
Maturity - the frequency of faults/ rate of software failure
Fault tolerance - the continuity of software execution in the presence of faults
Recoverability - the ease with which performance and data can be recovered in the case of system failure

***Usability*:**
Understandability - the effort required by users to recognise application concepts and their applicability to user tasks
Learnability - the ease with which an application's functions can be learned
Operability - the effort required by users to operate and control the application

From the ISO 9241 standard for usability in software and hardware design a number of other issues are identified such as:

System efficiency:
Time behaviour - the adequacy of system response and performance times
Resource behaviour - the acceptability of amount (and duration) of resources consumed in performing system functions

***Maintainability*:**
Analysability - the ability to identify and diagnose deficiencies in the system
Changeability - the ability to modify the system, to add new functions, remove faults or adapt to environmental change
Stability - the risk of unexpected effects arising from modifications to the system
Testability - the ease with which correct system functioning can be verified

***Portability*:**
Adaptability - the opportunity afforded to adapt the system to different specified environments
Installability - the effort required to install the software in a specified environment
Replaceability - the effort required to use the software in place of other software in a specified environment

This list of attributes is much larger than you will require. Select the most appropriate and concentrate on these.

Specifying the acceptable level of an attribute.
Having identified critical quality attributes, you need to specify what level or measure of each attribute is acceptable in your system.You should identify at least:

the *worst* acceptable level
the *planned* level - be ambitious, but remain realistic!
the *best* level - just to provide a marker for what might be technically possible but infeasible for you.

It might be useful to identify the present level (if there is an existing system to evaluate). These levels should be specified and measurable in quantitative terms or metrics. It is not good enough to specify that your system will be "very" efficient, "easy to use" or "extremely" adaptable. You must attempt to define *operational*, *measurable* criteria against which your system can be judged. This will lead on to defining a set of tests that will establish whether the attribute has been delivered to the required level. We will look at testing in a later chapter but it will often be important to identify, at least in general terms, what the testing approach will be.

For example, if one of your usability criteria for your system is its suitability for the task,

a measure of its *effectiveness* is the *percentage of user goals achieved in a given time*;
a measure of its *efficiency* is the *time for a type of user to complete a set of tasks*;

A series of experiments (tests) could be organised in which users are asked to carry out some important tasks using the system, we would then be measuring how well these were carried out, how long it took, how many mistakes were made etc. These experiments should be repeated with as many people as possible in order to get a useful result. Alternatively, a measure of *satisfaction,* for example*,* can be gained on a *rating* scale, e.g. a scale of 1-5 by using suitable questionnaires distributed to a selection of users during a trial period of evaluation. If access to real users is not possible in the timescale you could use some of your friends, preferably those with a similar knowledge of computing as the intended users of the system.

For each numbered attribute we will specify a quality level and eventually a test for determining whether it is met in the final, delivered, software.

User characteristics and user interface characteristics
It is worth writing down a description of who the intended users of the systems are expected to be.

Some of the basic principles behind your design of the user interface should also be documented. This does not mean that you should design some specific interface options but some simple diagrams can help the client to visualise how the system might look.

7. The formal requirements document and system metaphor.

XP has, in the past, worked with the set of stories and the system metaphor as the key planning documents for a project it is worth taking a look at how these are being used in projects. There has been much confusion about the System Metaphor and its role. In some industrial companies it is no more than a glossary of terms. In others it includes some simple architectural diagrams and general statements of intent. For a student project with a real client it makes sense to try to use both the stories and the metaphor as a mechanism for describing the current state of the project in ways that the client can understand.

This we propose that these documents are the basis of a requirements document that has some of the attributes of the traditional formal requirements document that has been used in the industry for a long time.

The difference is that our requirements documents is changing, evolving as the project develops, and so we will produce one at regular intervals. It will be a summary or list of the stories so far identified, the non-functional requirements and a glossary.

You must include the following information on any document you produce:

document type - e.g. requirements document
author(s)
version
date

The main body of the requirements document should have the following components:

Introduction and background;
Elementary business model;
User characteristics;
Functional requirements;
Non-functional requirements;
Dependencies and assumptions;
Constraints;
User interface characteristics;
Plan - a schedule of work with milestones, meetings and deliverables.
Glossary of terms (with an index)

The *scope* of the project is described in the Introduction and needs to be clarified with the client as far as possible as soon as possible. The requirements will change as we progress, one of the key points about XP is that it attempts to address this, but if the scope changes significantly we will be in trouble. We will therefore regard the requirements document produced during this phase as an *initial* one which will change as the project unfurls but it is important that the client, as well as the team, have some idea where they are going.

The document should have clearly defined sections and paragraphs, referenced by number and listed on a contents page. This is vitally important for future cross-referencing between other system deliverables and the requirements specification. As Tom Gilb makes clear, thorough cross-checking is necessary for software reliability. It is also vital when we come to testing that each requirement has a suitable test or set of tests associated with it. This then demonstrates that we have met that particular requirement.

In the future, you will only be able to determine whether your designs, your test plan and test cases and your coding are complete and correct, by reference to the sections of your requirements document.

You should discuss with the client whether they could sign off the document as a gesture of good faith. It should be made clear, however, that, in the spirit of XP, the requirements are not totally fixed. Explain to the client how XP works, so that they will expect to work with you on the stories, will help to identify the priorities while you try to estimate the resource implications.

Tell them that major changes, without good business reasons, will threaten the project. Some changes will be feasible, some not and it is important to remember that there will be a fixed time limit to the project. It's better to have a good basic and useful system than an incomplete and useless one.

8. Review.

We have concentrated on the discussions with the client and the formulation of the requirements for the project, both functional and non-functional. It is important that you maintain good communications with the client so that he or she knows what you are thinking about. Later, when we get down to more detail we will need to regularly review progress with the client, establishing the right language and concepts to use is vital. We have considered the structure of a fairly formal requirements document, one which might form the basis of an agreement with the client about what you hope to deliver.

After the first meeting you will have some ideas about the scope of the project. This is important and we will refer to the scope as being an important part of the requirements.

*Exercise***.**

Read the requirements document in Appendix A. *Criticise* it, in particular:

Are all the terms clear?
Are the functional requirements consistent, unambiguous, repetitive at the right level of detail?
Are the non-functional requirements clearly defined, do they have suitable acceptance levels and procedures identified?
How would you deal with any significant change in the requirements introduced by the client?
Would it be easy to maintain?

Conundrum.

Your team is in trouble. The client has not been in touch with her feedback on the proposed system. She doesn't have much experience of IT and only has a rather vague idea of what she wants. There are no similar systems known to you that you can show her. You need to start getting some requirements identified and some initial stories prepared.

Do you:
a) wait until she has thought further about the system she wants?
or
b) build a simple prototype using your imagination and background research in order to show her something that might stimulate her ideas?

References.
[Checkland1990]. P. Checkland and J. Scholes, "*Soft systems methodology in action*", Wiley, 1990.
[Gilb1988]. T. Gilb, *Principles of software engineering managemen*t, edited by Susannah Finzi. - Wokingham : Addison-Wesley, 1988.
[ISO 9126]

Chapter 5

Identifying stories and the planning game

Summary:
Creating stories and learning how to analyse and negotiate with stories. Managing the customer. Identifying functional requirements. Extreme modelling and the system metaphor. Non-functional requirements and quality attributes. Techniques for estimating resources.

1. Looking at the user stories.

In the main the stories will comprise short sentences which describe a sequence of actions and events that are recognisable from the perspective of a user interface. Many users and customers think about their system in this way and it is important to try to identify how a system might work from that viewpoint. However, some stories will involve internal types of processing which has no explicit representation in the user interface.

In a large scale industrial application there may be many teams or departments which collaborate in the development of a large piece of software. Here the relationship might be one whereby a team acts as clients to another team and will develop stories which are of a more technical and specific nature. The principle is the same, try to write down the story in clear language using well defined terminology and if your team is providing the software development effort to engage fully with the clients or customers in discussing the meaning, the relevance, the priorities and the costs of the stories.

In your project, however, you are the only team involved and your client will provide you with the key concepts for their business process and business needs.

We'll take some simple examples of user stories from the *Quizmaster* system to illustrate the process. Note that these can be traced back to the appendix document from the previous chapter.

Our starting point is the requirements document. In this we have identified a number of functional requirements and that is what we need to look at.

Each one of these has the potential to become a story. It depends on the level at which you described the requirement, we will assume that if they are too *high level* then you have broken them down into a series or sequence of simpler activities. For example the decomposition of the requirement to be able to set tests in the Quizmaster system to the sub-activities of setting questions and forming a test from a collection of questions.

The initial analysis of a story is to identify the business process that it refers to. To do this consider two basic things, what is being done and to what. In other words, there is some operation described in the story that is prompted by some intervention - normally a user action but it could be a signal from an external component or system, such as a sensor or something similar. This operation will affect some aspect of the system or its data and will usually produce some observable effect.

Now we create a card for each story, this will provide some basic information about the story and allow us to plan out our work. Recall that we gave a simple tabular description of each story. The columns define the following aspects of the story.

1. Its name.
2. what is the event that begins the story process.
3. what is the internal knowledge that is needed for the story.
4. what is the observable result,
5. how is the internal knowledge updated as a result of the story
6. what is the current priority of the story,
6. what is the estimated cost of the story.
7. what is the likelihood of the story being changed or dropped?

A story card should be created for each of these with the information described. A possible template for a story card is given in Figs. 1 and 2. Some of the topics will be discussed later.

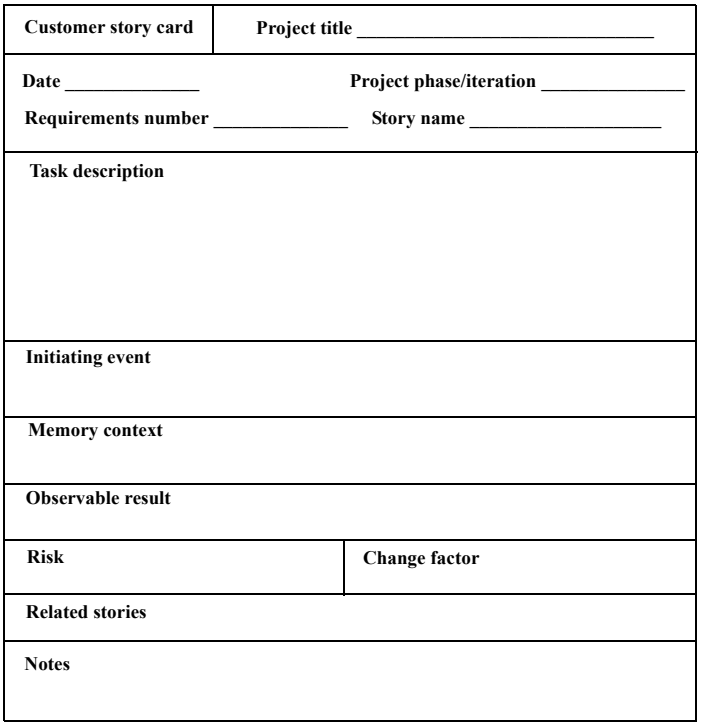


Figure 1 Story card template

Story name _____	customer approval ..... date.....
Resource estimates actual input ☐ output ☐ enquiry ☐ reference file ☐ database ☐ Function/object point total _____	simple ☐ average ☐ complex ☐ Man-hours total _____
Functional tests	
Quality attributes	

Figure 2. Reverse of story card.

Now let's return to the Quizmaster project and consider the first requirement in the document in the appendix, requirement 1, Lecturers can add topics. This will be a story card. Figure 3 shows how a story card might be written, some of the details are missing, in particular the resource estimates which we will return to. Notice that the story card implies the existence of a database of some sort which will contain information about papers and topics, this is extracted through the questions about what memory interactions there are.

Customer story card	Project title Quizmaster
Date March 15th 2001	Project phase Initial
Requirements number 1	Story name Lecturers can add topics.
Task description A function whereby a registered lecturer can define a topic for a paper and this information is stored suitably by the system	
Initiating event A request is made through chosing a menu option from a suitable screen	
Memory context A record of papers and topics exists and will be updated	
Observable result Confirmation of success and the ability to generate a list of papers and topics at any time.	
Risk factor 1 (low)	Change factor 1 (low)
Related stories Lecturers can delete a topic	
Notes Mandatory	

Story name Lecturers can add topics.	customer approval ..... date.....
Resource estimates actual input ■ output ☐ enquiry ☐ reference file ☐ database ☐ Function/object point total 1	simple ■ average ☐ complex ☐ Man-hours total 12
Functional tests 1. Define a topic when a paper is already defined 2. Define a topic when no paper is defined [error] 3. Define an illegal topic type [error] 4. Do nothing and exit the function [cancel]	
Quality attributes The process of defining the topic and receiving confirmation of correct operation should be instantaneous The operation process should be clear from the interface design Trials of this function amomgst representative users should be 99% successful	

Figure 3 A story card for a requirement.

Now we might identify another story which could be, for example:

3	Lecturers can edit details of a topic.

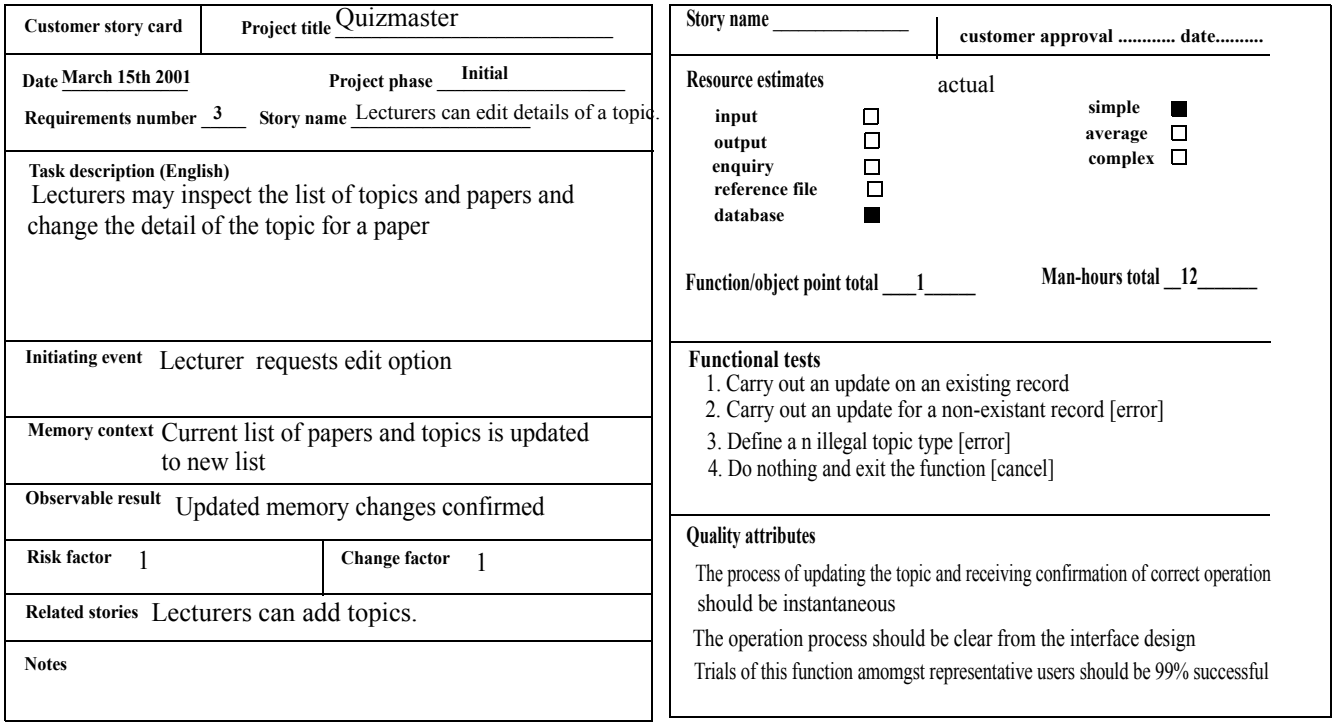


Figure 4. Another story card.

The XP method now tells us to write some tests for a story and then to code the story up. Writing tests, as we will see, is a sophisticated business and one of the weaknesses of much of the literature is that little advice is offered about how the tests are found, there is a lot of information about how to run tests and automate this process, but where the tests come from is somethong that seems to be left to experience - and this is somethig that you may not have!

Take one of your stories and write down some test cases. Don't forget what you are trying to do - to confirm that the code does what it is supposed to do and doesn't do anything else.

Looking at our first example story - *Lectureres can add a topic*, we need to write a class that will allow for the list of *topics* to be updated. The input to the class will be a *topic*, that is a string of characters, the method then needs to check that this topic is not already in the list of topics and then to add it to this list.

Testing this could involve:

an empty topic and an empty list;
a proper topic and an empty list;
a proper topic and a non-empty list;
an improper topic and an empty list;
an improper topic and a non-empty list;
a proper topic and a full list (defined in a suitable way);

Can you think of any more examples of tests? Good testers think awkward, trying out unlikely scenarios and data in an attempt to break the code. If you are to succeed in XP you must all

adopt this attitude. In traditional software engineering programmers tend to be too gentle with their own code, it's a psychological tendency which is hard to overcome, the influence of pair programming in XP is an attempt to prevent this. It's better for the programmers to find the bugs rather than the users!

Now we write the code and apply all the tests, correcting the code until they all pass. Naturally, we do this in pairs.

This is the basic XP process. How long did it take, make a note of the time you spent on writing the tests, writing the code and running the tests. This will give us an indication of what time it might take to write a similar story, although, hopefully, you will get quicker and better as you gain experience.

The story we have just discussed would need to communicate with other parts of the system and it is not worth showing the client this yet. There will be a user interface and a database, in all likelihood, and this class needs to be able to link in with them. While some pairs in the team are writing these basic units the others can be looking at the design of the interface and the database that will power the system. When we are clearer about these we will be able to write some system tests, link them together and see the results of our work, then we can show the client something that he/she would understand.

Although writing tests for simple classes such as this one is not particularly difficult things change when objects start communicating with each other and when a more complete architecture is being puit together, This is where things can go seriously wrong.

The next section details some simple techniques for thinking about system tests. The basic idea has been used in industrial settings and has seen massive improvements (up to 300%) in the effectiveness of the tests compared to the original test method being used.

2. Extreme modelling (XM)

Extreme programming is a movement which tries to reduce the initial analysis and design stages so that the project can evolve without the baggage of lots of documentation and rigid design models. However, its success is dependent on identifying really rigorous system test sets since these are what constrains the software solution to a high quality and correct solution.

It is impossible to define stringent test sets without knowing some details of what the system has to do. This is the basic dilemma that we face and one that has not been fully resolved within the XP movement.

Our approach is to apply an extremely powerful modelling paradigm that seems to be fairly easy to use and has *proven* excellence in the construction of functional test sets.

The task of taking the list of functional requirements or stories and identifying and organising them into a coherent system can be achieved using the technique that we will describe next.

To gain a greater understanding of how all the stories fit together into a coherent system we need to think about how they relate to each other. For example, it may be that one story can only occur after another one has occurred, or it might be that at some point in the business cycle there is a choice between several stories.

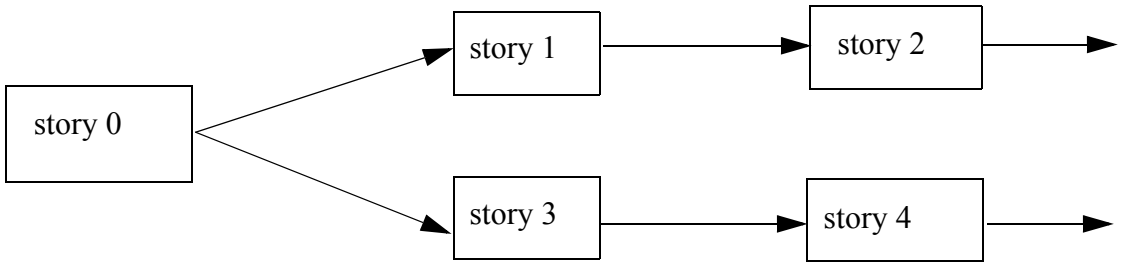


Figure 5. Collections of stories.

In this picture the initial story, story 0 is followed by either story 1 or story 2 (but not both at the same time) and then either story 1 is followed by story 2 or story 3 is followed by story 4. It might be then, that stories 2 and 4 are succeeded by further stories or the system returns back to the initial story.

In many cases each main story is associated with a user interface screen, there may be a whole screen to a given story or there may be many stories that can be driven from that screen. Although it is too early to plan out the detailed graphics of the screen it is still important to identify the key elements of the screen, the components that can be used by the user to instigate the process defined by a story, the extra information needed to be displayed for this and the result of the operation of the story displayed suitably.

We might break down a story into tasks which, when combined, provide a natural way to implement the story. One task might be to paint a screen - eg. a form, another might be to provide a data entry function which will connect to a task that performs some calculation woth the data. This might involve communicating with a database - to check with current data, and then to communicate the result back to them screen. Once these tasks have been programmed and integrated together we have a coherent story to show the client.

It is sometimes a good idea to show the client some of your thoughts, on paper, of how the story relates to your interface ideas before you do much coding. This can then lead to a clearer understanding of what is required.

The system will respond to some *external* stimuli, these will be, for example, users interacting with a screen entering data, choosing options through mouse clicks, ticking boxes etc.; messages from some other system, perhaps the results of a query to a database,

This data is then processed in some way, perhaps it’s just a simple calculation, perhaps the system needs to contact another part of the system, eg. a database, in order to proceed. The results of the computation may then be fed back to the user via a screen, or output in some other way or to another component. It is possible that the database needs to be updated as a result of this interaction.

This we have 4 essential actors in any system - an input actor, a processing actor, a *memory* actor, and an output actor. The memory actor - this could be managing a database - will be involved in reading and writing to the notional memory and commnicating with the other actors. The input actor reads the input from the screen or input device, the output actor deals with the output while the processing actor does the actual core computation.

We will see how this way of looking at things is both useful for planning out a program but also for testing it. It will be the basis for our system metaphor.

A SIMPLE EXAMPLE

Suppose that we are building a simple customer and orders database. We might identify a number of stories such as the following:

1. Customer details are entered customer by customer.

2. Customer details can be edited.

3. Orders are entered by customer

4. Orders can be edited when necessary.

The details of the structure of the customer and orders details are left until later, we try to build an abstract model of the user interface and then refine it. The test approach permits us to generate an abstract high level test strategy and to refine the test cases *in parallel* with the refinement of the system as explained in a later section [Holcombe1998], thus saving enormously in test case size for large examples - a recent case study, involving 3 million transitions, demonstrated this.

Now we try to identify from these stories, what is prompting change (inputs), what internal knowledge is needed (memory), what is the observable result (output) and how the memory changes after the event. We also try to identify the risk that the story will be changed during the course of the project as a means of trying to manage its evolution.

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low

3	quit()	click on return to start button	-	start screen	-	low

Table 1. Requirements table (part)

The table above describes some of the functions from the stories in this form.

Now consider some simple screens which may help us to visualise how it might work in practice.

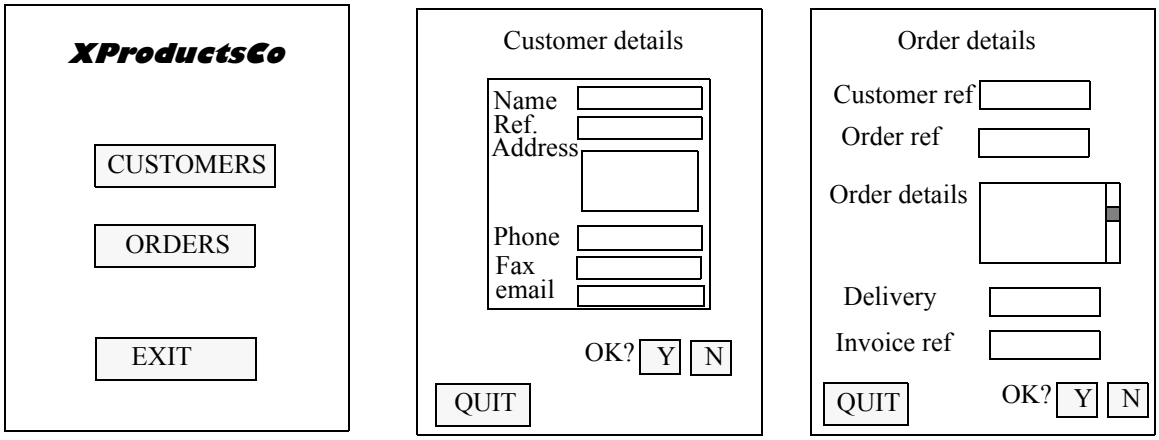


Figure 6. Some simple sample screens.

The confirmation screens are not shown. Notice that we have filled in some of the data details which are not explicitly described in the stories.

The next task is to think about the order in which these screens will be deployed and the tasks that can be done on each screen. This information will be described using a state diagram.

From the diagram (Figure 7) one can see how the basic functions are organised. We build up a simple model like a state machine. Each state has associated with it an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, *refinements* that can be dealt with later and the work we do on test set generation here is built on them. The complete state machine is actually called an *X-machine*, a term first introduced in 1974 (so we use the abbreviation XM to mean both X-machine and eXtreme Modelling). These machines differ from the standard finite state machine and from the statecharts sometimes used in UML in the sense that there is a memory in the machine and the transitions involve functions that manipulate the memory as and when an input is received. This provides an integrated modelling approach which combines in one diagram both processing and data.

This is explained by referring to the requirements table and the story cards where the memory connection is described.

For example, the memory in this case is likely to be the database that contains the record of customers and orders.

The function click(customer) simply navigates between two screens and has no memory implications, but the function enter(customer) will involve some interaction with the database, it might search to see if the supposed new customer is in fact new before proceeding - generating a message if the customer is already on the database. This can be described by the abort(customer) function which has been developed as a result of thinking about the way the system fits together.

The function confirm(customer) actually changes the database by updating the records with the information relating to the new customer.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

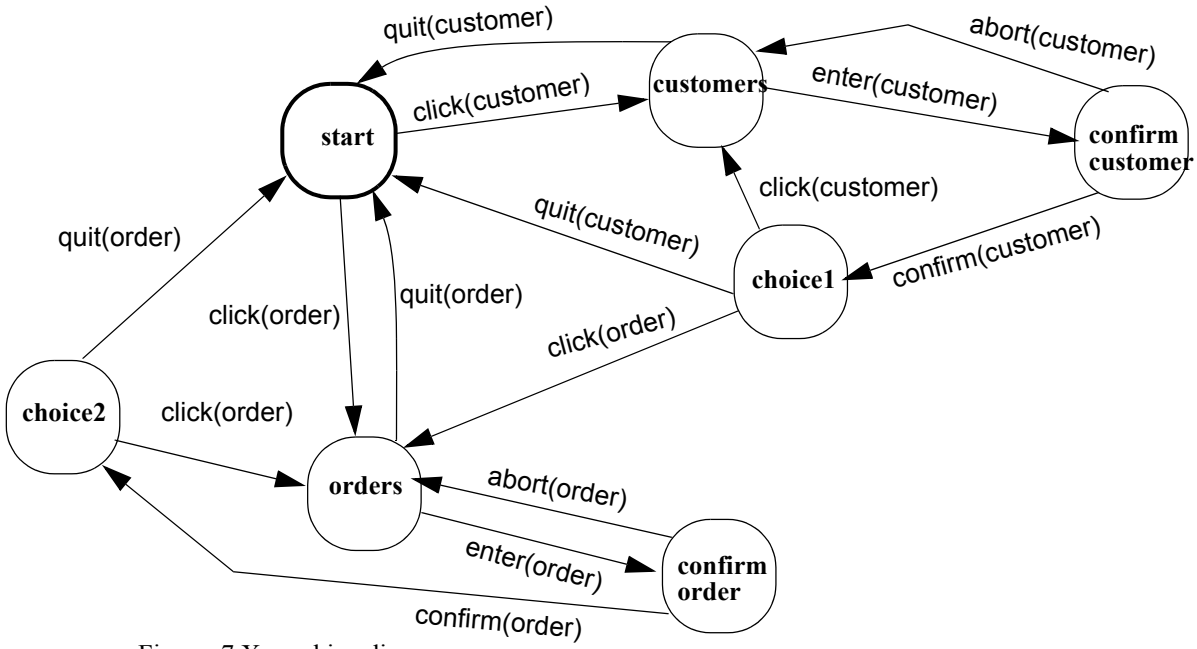


Figure 7 X-machine diagram

Looking at Figure 7 there are a few other things to note. We have created a specific user interface architecture which will constrain the solution in particular ways. For example it is not possible to quit the process of inserting the order details during this process. We can abort *after* completing it, however. This returns us to the **orders** state, then we can quit this screen.

The diagram can be used to define when and under what conditions different processes and stories are available.

Now we are beginning to see how the system might fit together, how it might look to the user and behave. It is worth showing the client some mock-ups of the interface at this stage and work through the scenarios derived from the requirements analysis that u have been doing. Now we can get some feedback to enable us to check if we are going in the right direction. The client can also re-examine his/her ideas and perhaps see new opportunities or problems with their original thoughts.

3. Managing the customer.

At each meeting - in our case weekly meetings are the only practical face to face opportunities - we need to provide the customer/client with a progress report. Because some of our clienrts need to keep their managers informed as well it is best to provide them with a document that summarises the current state of the project and where it is going to. This can be achieved by producing a requirements document.

After a few weeks this can become quite a detailed description of the system being built. It is suggested that a working requirements document is kept updated as the requirements change, in particular, as the collection of stories grows and the system architecture develops this should be recorded in a coherent way and the requirements document is the place for this. Because a student's life is a varied one and many other course and activities will be taking place concurrently with this project it is important to keep everything orghanised so that there is no confusion about what is being done and where one is going. This is why the requirements document is important. In many industrial companies it will be based around a standard template.

A suggested agenda for a regular client meeting/

1. Progress update - what has been achieved since the last meeting;
2. Review of system requirements;
3. Demonstration of new code and interface mock-ups;
4. Changes needed to existing stories/requirements;
5. New stories/requirements;
6. Plans for the coming week;
7. Next meeting - time and place.

4. The requirements document.

The format of the requirements document that we will be presenting to our client was discussed in the previous chapter and an example is given in Appendix A. The important thing about this document is that it should be understandable to the client. It is built from the basic information in the stories together with some outline ideas of the how the system might look and work. Important non-functional requirements need to be specified and clear statements about how these are to be interpreted and tested included. It is no good saying that the system will be fast if we don't say what that means, for example, in a Web based system this might include the maximum acceptable page download times under suitable conditions, etc.

Although we have presented the requirements document and the story cards as two separate things they are very closely related. There will be an interplay between them. We might regard the requirements document as a *summary* of our current understanding of the overall system whereas the stories are a more detailed description of individual aspects of its functionality with enough information to allow us to plan, test and implement each story. The requirements document will have extra and vital information about the proposed system, context statements, assumptions as well as global quality attributes and non-functional requirements. It is these that we turn to next.

Non-Functional Requirements

A non-functional requirement either describes how well the system should perform (a quality attribute) or a constraint or limit that the system must adhere to (a resource attribute). The non-functional requirements were defined in Chapter 4, and can be split into categories like *reliability, usability, efficiency, maintainability* and *portability etc*. Here we give some example statements that might be part of the requirements document.

Reliability:
Examples are:

For a single user, the system should crash no more than once per 10 hours.

The system should produce the correct values for any mathematical expression 100% of the time.

If the system crashes, it should behave perfectly normally when loaded up again with minimal data loss.

Usability:

A user should be able to add a new customer to the system within 1 minute.

A user should be able to add a new order to the system within 1 minute.

A user should be able to edit a customer's details within 5 minutes (will vary with details type).

A user should be able to produce reports and statistics within 1 minute.

Efficiency:

The system should load up within 15 seconds.

The time taken for the system to retrieve data from the server should never exceed 30 seconds.

Maintainability:

The system should be designed in such a way that makes it easy to be modified in the future.

The system should be designed in such a way that makes it easy to be tested.

Portability:

The client system should work on the client's current computer network which is connected to the Internet and has Windows 95 or better.

The system should be easy to install.

These statements need to be refined into a more precise statement in order to make them testable. What, for example, does *easy to install* mean? we will look at this in the next Chapter.

From each story that we have discussed with the client we extract the key functional details. These are grouped in sections with other story lines that are clearly related.

These requirements are categorized on the basis of which are *mandatory, desirable* and *optional*. To do this we need to have an estimate of the time we might take to complete these and this will help us to make these decisions. The next section looks at the process of trying to estimate this.

Naturally we must consult the client on which he/she thinks are mandatory etc. *We have to be realistic, however, you must not promise to do more than you can achieve in the time given.*

4. Estimating resources.

If we have a model like the one above we can use something like the *function point* and *object point method*. Here we try to estimate the amount of effort required to build a story or a screen with its accompanying functionality.

To do this we look at each story and consider the functions contained in it. We then try to categorise what sort of function this is. we can find information which estimates the amount of effort each category of function might require, this data is being collected in industrial organisations and some of it is published. We include some examples here. It is a good practice to try to measure your own efforts for these functions to see if they agree with the estimates and to inform future projects.

Software cost estimation.

Basic questions at the *start* of the project and also at suitable review points *during* the project:

How long will it take?

What resources will it need?
How expensive will it be?

The approach is to use the techniques of Software measurement.

During the course of projects we measure the following parameters:
lines of code (loc) produced over the timescale;
number of observed defects over the timescale;
number of person hours worked over the timescale;
amount of time spent on debugging over the timescale;
amount of time spent on requirements over the timescale;
amount of time spent on design/specification/analysis over the timescale;
amount of time spent on writing documentation over the timescale;
amount of time spent on testing/review over the timescale;
etc.
These are all measures of production volume, product quality and effort. If we have some previous experience and data of this type for old projects we may be able to estimate the effort and time needed for the new project.

From the timesheets and documentation produced we should be able to find the following for the completed project:
lines of code (loc) per person-month (pm);
cost per 1000 lines of code (kloc);
errors per kloc;
defects per kloc;
pages of documentation per kloc;
cost per page of documentation;
number of requirements;
average kloc per requirement.

If we have this data then we might be able to estimate what the next project will need in terms of people and time.

But different types of project will require different amounts of effort, so we need to collect information about the type of project:
product functionality;
product quality;
product complexity;
product reliability requirements;
etc.
These are not always easy to measure, unlike the first set of measures.

We need to describe the software being built on the basis of the requirements in order to estimate the resources needed. There are several techniques, none of which are very precise. If the new project is very similar to the previous ones things are much simpler. If it is a completely new type of project, perhaps involving a new technology, then it is much more difficult.
We will look at the *function point method* (see the Function Point User Group - www.ifpug.org). There is also a more modern method called *object point analysis*.

Function point analysis (FPA) was developed by Albrecht in 1979, [Albrecht1979] for business information systems development.

The principle underlying FPA is that the effort required to construct a software system is a function of the size and complexity of the product. The size is determined by the number and complexity of the requirements not the size of the code.
Method.

1) for each requirement/story we decide if it is one of: *input, output, inquiry, reference file, database.*
2) assign a weight to each requirement: *simple, average, complex*.
3) consider other influencing factors: reusability, adaptability and weigh them according to a suitable scale. This is, to an extent, guesswork but if we have an idea of which requirements are hard to implement and which are easier it will help us to plan.

Below are a list of influencing factors that could either make the requirements easier to implement or harder to implement depending on how much influence these factors have on the system. Each factor is allocated a number range and we try to estimate from this the likely weight of the factor.

criteria	simple	average	complex
number of different data elements	1-5	6-19	>10
input validation	formal	formal, logic	formal, logic, database
ease of use	low	average	high

Table 4: Input data (keyboard, screen, from other applications)

This means that if there are 4 data elements on a form and there is little data validation and ease of use is not an issue then the input data requirement is *simple*. If there had to be some logic based data validation then it would be *average* and if ease of use was a big issue then it would be *complex*

criteria	simple	average	complex
number of columns	1-6	7-15	>15
number of different data elements	1-5	6-10	>10
number of formatted data elements	none	some	many

Table 5: Output data (monitor, printer, to other application)

criteria	simple	average	complex
number of different keys	1	2	>2

criteria	simple	average	complex
ease of use	low	average	high

Table 6: Inquiries (search and display)

criteria	simple	average	complex
tables: number of different data elements	1-5	6-10	>10
tables: dimension	1	2	3
read-only files: number of different data elements	1-5	6-10	>10
read-only files: number of keys/sentence formats	1	2	>2

Table 7: Reference file (tables and read-only data

criteria	simple	average	complex
number of keys/sentence formats	1	2	>2
number of different data elements	1-20	21-40	>40
database already exists?	yes	-	no
new implementation?	no	yes	-

Table 8 : Databases (managed by the application)

Influencing factors.
Having identified the key functionality of the requirements we need to factor in the effects of a variety of important aspects. These include the extent to which the application will interface with ther applications - often a source of interface problems, the organisation of data stores - distributed storage and procesing adds to the complexity of the design, the transaction rates expected - high transaction rates bring special considerations, being able to reuse some parts of an existing system, the amount of data conversion involved - perhaps we have to migrate data from an obsolete system. These issues need to be considered and the following table provides some details of how to do this. For those factors given a scale of 0 - 5 we define these by:

0 = no influence, 1 = occasional influence, 2 = moderate influence, 3 = average influence, 4 = important influence, 5 - strong influence.

For the scale 0 - 10 these points get double weighting.

Factor
1) Interplay with other application systems (0-5)

2) Decentralised data (processing) (0-5)
3) Transaction rate (0-5)
4) Processing
(a) Calculations/arithmetic (0-10)
(b) Control (0-5)
(c) Exceptions (0-10)
(d) Logic (0-5)
5) Re-usability (0-5)
6) Data conversions (0-5)
7) Adaptability (0-5)

Table 9: Influencing factors

Re-usability is rated as follows: 0 = less than 10% reuse, 1 = between 10% and 20%, 2 = between 20% and 30%, 3 = between 30% and 40%, 4 = between 40% and 50%, 5 = above 50%.

We now consider all the requirements and create a table which summarises the function point position: here is an example:

Category	Number	Classification	Weight	SUM(FPs)
Input data	7	simple	3	21
	5	average	4	20
	1	complex	6	6
Inquiries	2	simple	3	6
	0	average	4	0
	0	complex	6	0
Total				53

Table 10: Example function point table - Quizmaster system.

Here there are 7 simple input requirements/stories each given a weighting of 3 making a function point score of 7 * 3 = 21. We do this for all the requirements/stories, only part of the full table is given see Appendix B for the details.

The next stage is to calculate the sum of the contribution of the influencing factors. If the sum of the influencing factors in N then the total function point contribution of the influencing factors is N/100 + 0.7

An example is given in the following table:

Factor	Score
1) Interplay with other application systems (0-5)	0
2) Decentralised data (processing) (0-5)	2
3) Transaction rate (0-5)	2
4) Processing	
(a) Calculations/arithmetic (0-10)	3
(b) Control (0-5)	2
(c) Exceptions (0-10)	4
(d) Logic (0-5)	2
5) Re-usability (0-5)	1
6) Data conversions (0-5)	1
7) Adaptability (0-5)	1

Table 11: Example influence factor table - Quizmaster system)

This influence factor value is then multiplied by the function point total from table 7 to give the final function point value for the list of requirements

The historical data on function point effort can now be used to calculate the resource required to implement a suitable system.

Data is available from a number of sources which can help us to make the time and resource estimates. In an ideal world these would be available from previous student projects but collecting reliable data on person months is always difficult. You may have already implemented a few simple stories and this data might be useful for predicting how much time you will need to spend. The time it takes for a story to be implemented and tested can be used to estimate the *velocity* that the team can work at, however, it is very dangerous to assume that each storywill involve the same amount of effort, we need to ne more subtle and the technique presented here is one way that this can be done. To use it, however, we need some snesible data about the effort required to implements a function point. This has been obtained from industrial surveys but may not translate well into an XP project run by students.

Here is a table of person month data for various function point values derived from some commercial projects in IBM.

Function points	IBM person months
50	2.3
100	5.6
150	9.5
200	13.9
250	18.6
300	23.6
350	28.9
400	34.4
450	40.1
500	46.1

Table 12: Industrial data.

This data is based on averages taken over a variety of projects with many different project teams. It can only be a rough guide until you have established your own data.

It also depends on the programming language used, generally the number of lines of code per function point varies from 128 for C to 22 for Smalltalk. Java is probably around 30.

Given that the number of requirements that we have are more than we can deliver in the time scale we have to decide which ones are *mandatory* or essential, which are *desirable* but not quite essential and which are *optional*. The function point values can then be used to work out what requirements are feasible within the resources and time available. For these types of projects the amount of person months available for each team will vary from 2 - 3. So function point totals of around 50 are feasible.

Object point analysis.
This method was introduced by Banker, Kauffman et al in 1992 [Banker1992].

It is based on:

the number of separate screens - simple = 1 object point, average = 2, complex = 3.
number of reports to be produced - simple = 2 object points, average = 5, complex = 8.
number of modules that must be developed, 10 object points for each module.

A module will be any small coherent part of the system, it could be a screen or a story.

It is easier to calculate this from the high level requirements.

The COCOMO model, [Boehm1995], is another estimation process which is based on industrial data. See any Software engineering text such as [Pressman2000] or Sommerville[2000] for more details.

COSMIC FFP
This is an updated version of the Function Point Analysis.

It is based around a very simple definition of a fiece of software functionality, A function Process is a unque set of data movements : input, output, read and write.

Actors trigger these movements, directly or indirectly. these movements are also identified at the lowest level in terms of the requirements, that is, we do not break them down into smaller functions. Events will trigger these functions and we consider them to run until they complete.

Then we sum up how many of these functions there are, ignoring their type or any other factor. Such a simple method of estimating might be very useful for XP projects, it still requires the collection of data about a team's velocity in order to be useful, though.

Further details can be found at http://www.cosmiccon.com

5. Review.

This chapter has looked at how the planning game and the use of story cards can help us to develop a detailed requirements document. Although the requirements are changing it is important to bring all the changes into a single document at suitable stages through the document. We considered how the different functional requirements can be integrated into a simple (extreme) model which helps us to think through how the system, will work overall. These processes will be repeated as the requirements emerge and change. Chapter 9 will examine some of the issues relating to this evolution of the system and the ways it can be articulated and managed.

Non-functional requirements were identified as a key factor in the success of a system. These need to be thought about very carefully and clear and measurable statements made about them.

Estimating the resources needed to complete a project is notoriously hard and two techniques, function point and object point analysis were described. These can only be rough and ready guides until you get more experienced. Noting down as much detail about your team's performance and the time taken to do things will provide an ongoing and useful archive for future projects.

Conundrum.

The company wanted an *intranet* that provided support for many of their business activities and also their personnel management. The site would contain information about the various company activities, a diary system and templates for administrative tasks such as the submission of illness and absence forms. The users would be able to log on remotely to carry out tasks as well as from within the company offices. The customer was able to maintain a very close relationship with the development team and had a clear idea of what the company needed. There were 3 teams using XP working on this project, each competing with all the

others. Initially all the teams thought that the project would take 10 weeks. It didn't quite work out like that. When the first team delivered their first increment they discovered something important that had not emerged from the planning game. The company had a service agreement with a third party network solutions company which provided the computer system and the internet connection for the customer. This led to a serious problem for some of the teams and resulted in some failing to meet the 10 week deadline despite the careful planning.

What might have been the problem?

Exercises.
1. Read Appendix A and B. These relate to a real project that was successfully implemented using XP.
2. Prepare your own requirements document for your client for submission also to your tutor. The contents are specified below. Use the planning game to create the individual requirements for the document.

Your requirements statement should contain the following sections and paragraphs:

Introduction - a statement of the required system's purpose and objectives

Dependencies and assumptions - things that will be required for your system to meet its specification, but which are outside your control and not your responsibility

Constraints - things which will limit the type of solution you can deliver, e.g. particular data formats, hardware platforms, legal standards

Functional requirements - you are advised to priorities your requirements into those that are:

- mandatory
- desirable
- optional

Non-functional requirements - with accurate definitions and an indication of how they are to be measured and the level required.

User characteristics - who will the users be?

User interface characteristics - some indication of how the interface needs to be structured and its properties.

Plan of action - defining milestones - key points in the project

- deliverables - an indication of when increments will be ready
- times when these events will occur.

Glossary of terms.

Any other information such as important references or data sources etc.

Here is a simple tabular template that could be used for some of the functional and non-functional requirements. It includes a column for trying to set priorities for the individual requirements and to identify the risk of change in the requirement, a difficult thing to estimate but worthwhile for planning purposes.

Number	**Description**	**Mandatory / Optional / Desirable**	**Priority (1-9)**	**Risk (1-9)**	Function point
1.					

References.

[Albrecht1979], A. J. Albrecht, "*Measuring application development productivity*", SHARE/ GUIDE/IBM Application development Symposium, 1979.

[Banker1992], R. Banker, R. Kauffman et al, "*An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment*", J. Management Inf. Systems, 8, 127-150, 1992

[Boehm1995], B. Boehm et al, "*Cost models for future life cycle processes: COCOMO 2* ", Balzer Science, 1995.

[Holcombe1998], M. Holcombe & F. Ipate, "*Correct systems: building a business process solution*", Springer, 1998 available on-line at: http://www.dcs.shef.ac.uk/~wmlh/

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

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

Chapter 6.

Designing the system tests.

Summary:

What are tests? Developing a model to aid test generation. Building the functional test sets for the stories. Documenting the tests and the test results. Design for test. Non-functional testing and testing the quality attributes.

1. Preparing to build the functional test sets.

1.1 Tests and testing.

First, what is a *test*?

A test is an application of some user or system input in a particular state of the system to establish if the system response is what is expected or is faulty in some sense - it might produce an incorrect response or output, no output, or the system might crash. A test must comprise both the *test input* and the *context* that the system must satisfy together with the *expected* output.

Going back to our interface screens (Fig. 4, chapter 5) we might consider the first screen and apply a test by clicking on the *customers* button. The expected result is a new screen, the customer's screen. That is all. We would not want the database to be deleted also.

When in the customer's screen we might wish to test the data entry of the customer name and address. We have to specify the state that the test starts from and the data that we will insert. Now the data entry requirement will be based around some part of the software architecture, some classes or functions that accept user input and do things with it. The extent of the system testing that needs to be done is related to the level and extensiveness of the testing that has been carried out at the unit testing stage. Unit testing will be covered in the next chapter, it is the testing of the classes and components that will be put together to form coherent subsystems which will provide some functionality that we can relate to some of our stories which are integrated together to form some useful business functions.

For example, if there are data integrity checks being carried out then these have to be tested. If there are table look ups, for example the system might check that a specific postal code exists by consulting an official list of these, or it might need to check that the format of the input is correct - letters where letters are expected and numbers also. No control characters etc. It might need to check whether the customer is already registered on the database and this will involve a query of the current database state.

The decision as to where the testing should be done - unit level or system level is not always clear. Obviously the more complete the testing at unit level the better but it is not possible to do all the things that are needed since inter-class communication and communication with databases and tables may not be possible at that point in the project.

We will have to include in our system testing, test cases which will expose faults in the data entry checking. We would do this by having test cases with *draft* input, for example. This might be invalid symbols or no symbols - perhaps just return - and so on.

We should also test the system under different conditions relating to the database, if any. For example, at the beginning the database is empty, we would test the system under these conditions, also when the database has some data in it and when it has a lot in it.

Where data entry has been carried out and hopefully stored in the database we need to establish that the data has been stored correctly. This needs either setting up queries as part of the test or writing a suitable script to pull the data out to check that it is OK.

So the test cases must reflect all of these things - what we put in and what we expect to see, and what we actually see.

The tests that we are talking about here are tests that relate to coherent pieces of a functioning system and for them to be carried out we will need some code to test. In an XP project we will have implemented and tested some stories and integrated them into such a subsystem. We can then apply the tests developed according to the methods in this chapter. The stories will be built from units of code and how these units are tested will be the subject of the Chapter 7

1.2 Testing from a model.

The machine based model we started to develop in the last chapter can be very useful when it comes to finding good system test sets.

If we can build a machine diagram like fig.1 and relate all the main requirements to it we can then create some very powerful test cases.

We developed our model for two main reasons. Firstly, it was to try to understand, from the point of view of the behaviour of the system, how it all fitted together and mapped onto the requirements. Secondly, we will use it to generate test sets that will be fundamental to how we will establish that the system works.

Our model was based on identifying a set of states and the operations (functions) that operated between the states.

Recall the diagram (fig. 5) from Chapter 5:

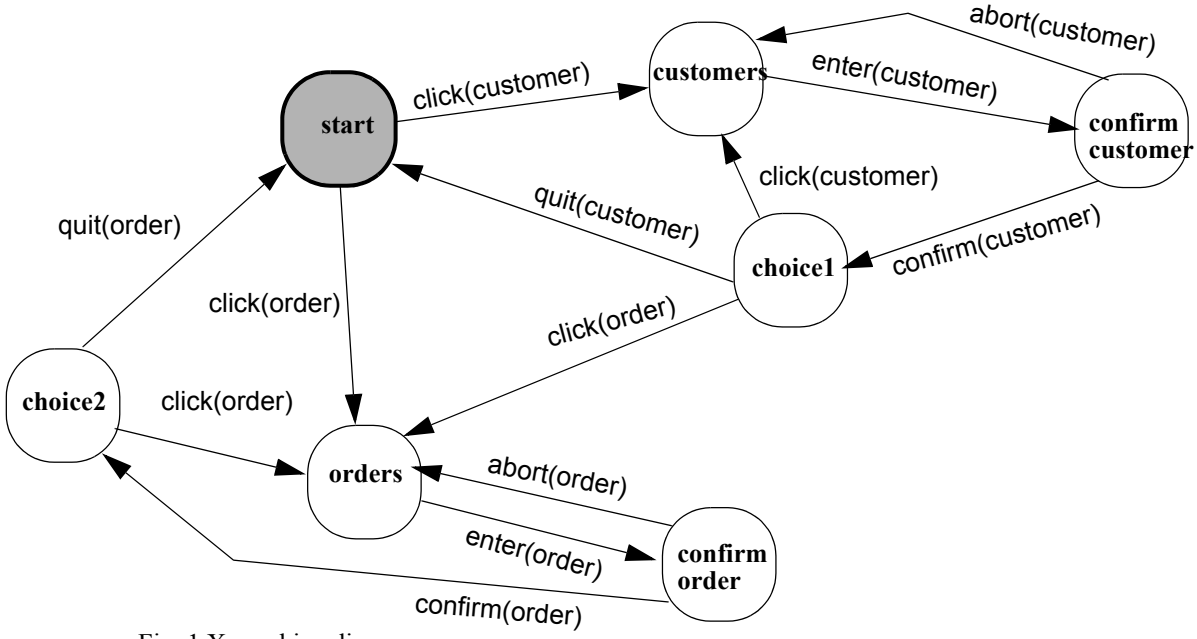


Fig. 1 X-machine diagram

we will consider how to create test sets which will systematically exercise the system.

An obvious starting point is to try to check out the paths through this system (machine), this means looking for the conditions and activities that will force the system through paths made up of sequences or arrows. This we will do and then we will consider how such *path sets* could be turned into test sets.

Suppose that we start with the system in the start state. Recall that our system also contains an internal memory value which needs to be considered. Let's assume that we are starting the system with some initial memory value, perhaps the set of records in the database is empty - we are awaiting the insertion of the details about our first customer. We will call this the *initial memory value*.

The first thing we have to do is to make the click(customer) function operate as one might expect this is achieved by clicking on the **customer** button in the initial menu page. If this part of the system has been built properly we should expect to move to a new state, the screen for customer entry. We need to exercise the enter(customer) function next. This will be achieved by filling in the customer entry screen and submitting it. The result should be the move to the customer confirm state and so on.

A simple notation to describe these sequences of operations is to use the sequence operator, **;** well known to those who have studied functional programming or formal methods. Thus:

```
click(customer) ; enter(customer)
```

describes the first two operations.

We continue in this way forming sequences of legitimate operators to represent possible paths through the system. Each sequence will define a test that has to be carried out in a systematic way and the results observed and compared with what the model and requirements state.

If we find a problem then it has to be investigated and fixed.

Another set of tests would be carried out by following the paths:

click(order)
click(order) ; enter(order)
click(order) ; enter(order) ; confirm(order)
click(order) ; enter(order) ; abort(order)
click(order) ; enter(order) ; confirm(order) ; enter(order)

and so on.

This approach to finding test sets, or paths through the model, is one of the simplest is called the *transition tour*.

In a transition tour we choose a number of paths beginning at **start** and try to visit as many states as we can.

Some further examples are:
1. click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; abort(customer)

2. click(customer) ; enter(customer) ; confirm(customer) ; quit(customer)

3. click(order) ; enter(order) ; confirm(order) ; click(order) ; enter(order) ; abort(order)

4. click(order) ; enter(order) ; confirm(order) ; quit(order)

To turn these sequences of transitions into a set of test inputs we need to choose the various inputs that cause these sequences of transitions to operate. It's easy here, we look at the screens and identify either suitable buttons to press or insert data in appropriate places to make it all happen.

We use the notation <`enter`("ThingsRUs")> to indicate a test input in terms of the function we are executing and the data values that are supplied to it.

brackets like: < **** > tell us that this is a test input,

the phrase: `enter`("ThingsRUs") indicates that there is a function applied, this function is `enter` which requires an input parameter or data value, here this data value is "ThingsRUs".

So, the notation for these sorts of test inputs is:

< `function`("data_value")>

There is no widely used standard notation for describing test inputs, this seems to work in most cases that are likely to arise in this sort of project.

Thus the first test input sequence, corresponding to tour 1 might be:

T_1 = <button_click(customer)> ; <enter("ThingsRUs")> ; <enter("Tolpuddle Estate,"> ; <enter("Utilityville,")> ; <enter("00123 4567")> ; <enter("00123 4568")> ; < enter("TRU.com")> ; <button_click(Yes)> ; <button_click(customer)> ; <enter("Serendipity products")> ; <enter("Towchester Road,"> ; <enter("Norfolk City,")> ; <enter("00555 4321")> ; <enter("00555 4322")> ; < enter("Serendipity.co.uk")> ; <button_click(Yes)> ; <button_click(cancel)>

For this to represent a proper test we need to describe the context under which it should be applied and what the expected output is. A tabular method for doing this and for recording results is discussed later in this chapter.

Such test sets will provide a good basis for testing the functionality of the overall system but they can be improved. There are a number of faults, for example, that such a test strategy may not reveal, including extra states, some faulty transitions etc.

These tests will tell us if the system is doing the things that we know we want it to do. However, they will not tell us if the system is doing anything else, possibly something undesirable. The key to good testing is to know enough about what it is you want and to test that this is what you get and *you don't get anything else*. Thus we need to try to see if any *illegal* operations are possible. We need to do this throughout. The simple strategy is to take a path through the system to some state and then to introduce tests which test whether any functions that are not defined at the state are actually present. So we send the system to the target state and then try to apply all the functions that are *not* supposed to be defined there. This should cause an *error* and we want to see that happen. Otherwise there is something going on in that state when this *supposedly absent* function is called.

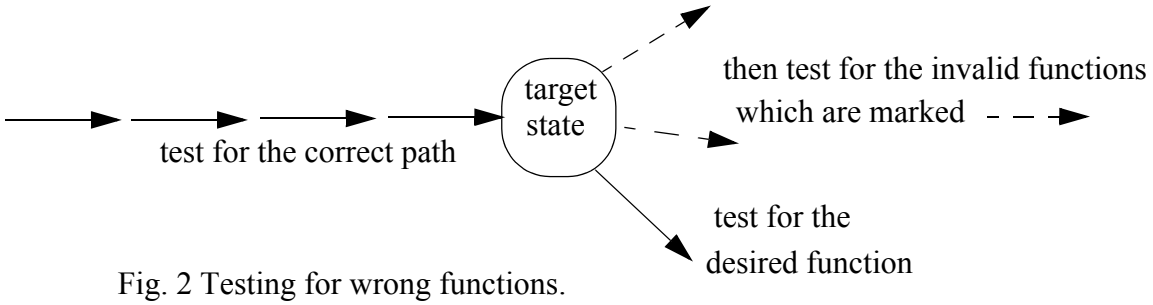


Fig. 2 Testing for wrong functions.

1.3. Developing the model.

Figure 2 illustrates part of a path through the state diagram that is being exercised by a test case. From the diagram (Fig.2) one can see how the basic functions are organised. We build up a simple model like a state machine. Each state has associated with it an appropriate screen with buttons, text fields etc. Of course, the model is simple and crude, there is no distinction between entering a new customer's details and editing an existing one but it is enough to explain the method. These refinements are, what they say they are, *refinements* that can be dealt with later and the work we do on test set generation here is built on them. The complete state machine is actually called an *X-machine*, a term first introduced in 1974! (See [Eilenberg74]). These machines differ from the standard finite state machine in the sense that

there is a memory in the machine and the transitions involve functions that manipulate the memory as and when an input is received.

A good way to think about these machines is to draw the state diagram and remember that the transitions that act between each state represent system functions that are triggered (usually) by an external event (user actions) and which carry out processing over some database or a global or local memory store. In many cases the functions can be described very simply. One could use the formal notation Z[1] or VDM[2] for this but I prefer a simple functional notation.

This memory will be derived from the requirements in a natural way so we refer to the requirements table where the memory connection is described.

For example, the memory in this example is likely to be the database that contains the record of customers and orders.

The function click(customer) simply navigates between two screens and has no memory implications, but the function enter(customer) will involve some interaction with the database, it might search to see if the supposed new customer is in fact new before proceeding - generating a message if the customer is already on the database. The function will involve moving between the different slots in the form filling in the required details. It shouldn't matter which order we do this in so the testing of the enter(customer) function will involve typing in suitable data values into these slots in various orders. We may also have a requirement that the user can cancel a data entry at suitable places in the interaction and this can be described by the abort(customer) function which has been developed as a result of thinking about the way the system fits together and the needs of users.

The function confirm(customer) actually changes the database by updating the records with the information relating to the new customer.

The memory structure now needs to be discussed. Essentially we need to think about this in terms of what basic types of memory structure is relevant at the different levels. At the top level, for example we could represent it as a small vector or array of compound types of the form:

customer_details × order_details

filling in the actual details later. It may be, for example, that these will represent part of a structured database with a set of special fields which relate to the design of the screens associated with these operations. So customer_details would involve name, address etc. which would be represented as some lower level compound data structure, perhaps and there would be basic functions which insert values into the database table after testing for validity etc.

Now we can describe some of the functions from the diagram. First note that the memory is just a set of records with 2 main fields, the first structured into:

customer_details : name, address, postcode, phone, fax, email

and the second into:

order_details : customer_ref, order_ref, order_parts, delivery, invoice_ref

1. Z is a mathematical notation for specifying simple systems. [See Spivey1992]
2. VDM is a similar notation, see [Jones1986]

Then the set of all *current* customer details is given by:

Customer_details

so Customer_details might be {customer1, customer2, customer3} after 3 customers have been put in,

where customer1 = [name1, address1, postcode1, phone1, fax1, email1] and so on.

The notation using the braces, {, }, is a way of describing all the members of a collection of data elements that we want to refer to as a whole.

The set of all possible sets of customer details that there ever could be can be defined as CUSTOMER_RECORDS.

So Customer_details ∈ CUSTOMER_RECORDS

For those who have not attended a Discrete Mathematics course this is simply read as:

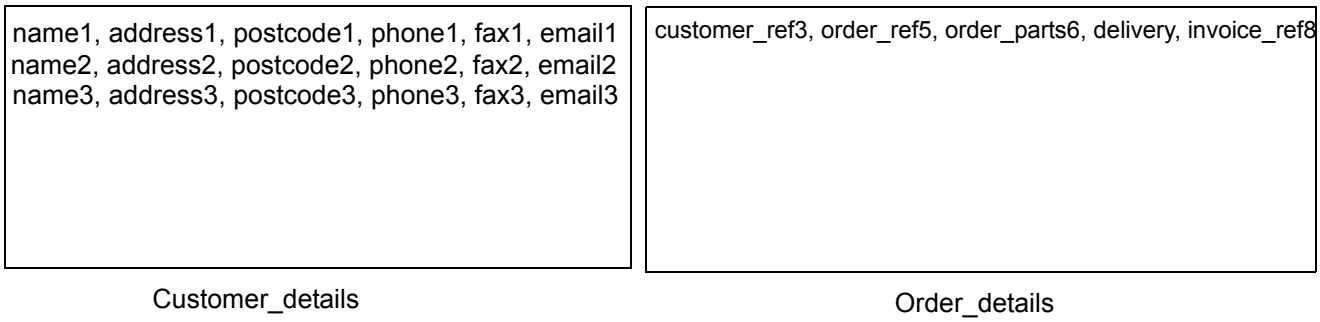
Customer_details is a typical member of the collection of all CUSTOMER_RECORDS

And the current order details is {order1} after one order has been put in, eg.

Order_details = {order1}

where order = [customer_ref3, order_ref5, order_parts6, delivery, invoice_ref8]

Thus Order_details ∈ ORDER_RECORDS



The state of the memory after 3 customers have been inserted and one order.

Let the set of current memory values is given by:

Customer_details × Order_details

A typical element of this memory set is

([name, address, postcode, phone, fax, email] ,[customer_ref, order_ref, order_parts, delivery, invoice_ref])

The function definition for enter(customer) would now look like:

enter(customer) ([name, address, postcode, phone, fax, email] , (Customer_details × Order_details)) =
((Customer_details × Order_details) ∪ {([name, address, postcode, phone, fax, email], -)}, display)

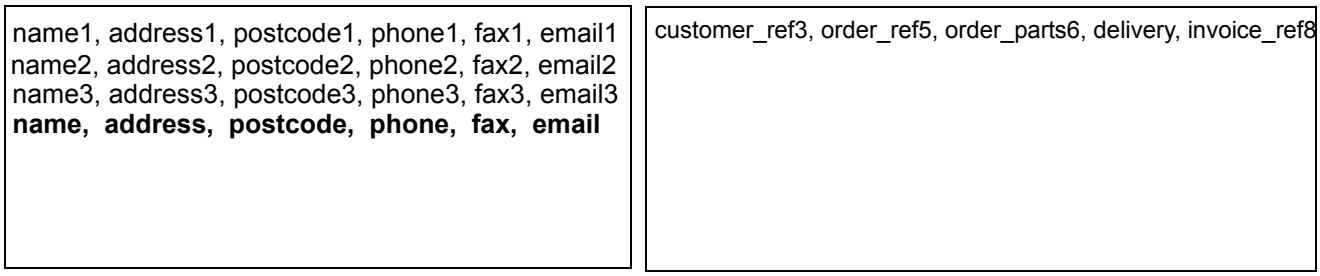
if [name, address, postcode, phone, fax, email] ∉ Customer_details
else error message : "customer already present"

The precondition is that these details are not already in the database. The Order_details part will not be changed by this operation. The display element is the visible screen message asking for confirmation of the input data (the screen for state **confirmcustomer**) or an error message.

The function's type would be:

$$\text{enter(customer)} : \text{INPUT} \times \text{CUSTOMER\_RECORDS} \times \text{ORDER\_RECORDS} \rightarrow \text{CUSTOMER\_RECORDS} \times \text{ORDER\_RECORDS} \times \text{OUTPUT}$$

Here INPUT is the set of all possible inputs/data entry and OUTPUT is the set of all possible displays. Note that this function is a *partial* function. It can only operate if certain pre-conditions hold such as the input is of the correct type.



This is now the state of the memory after the enter(customer) function has been applied with the new customer data: name, address, postcode, phone, fax, email

All the other functions in the diagram can be defined in a similar way.

All types of systems can be described in this way - we need to identify the functions involved, the data and memory they use and the states that sort out which function is valid and when. Such X-machines are extremely powerful - as powerful as Turing machines.

We are going to use this state diagram to define how we can build a set of functional system tests that will link to the requirements and be extremely effective.

Most testing is *ad hoc* in the sense that the creation of the tests is left to the tester's common sense. This may not be a very effective way of finding faults - and this is what testing is all about.

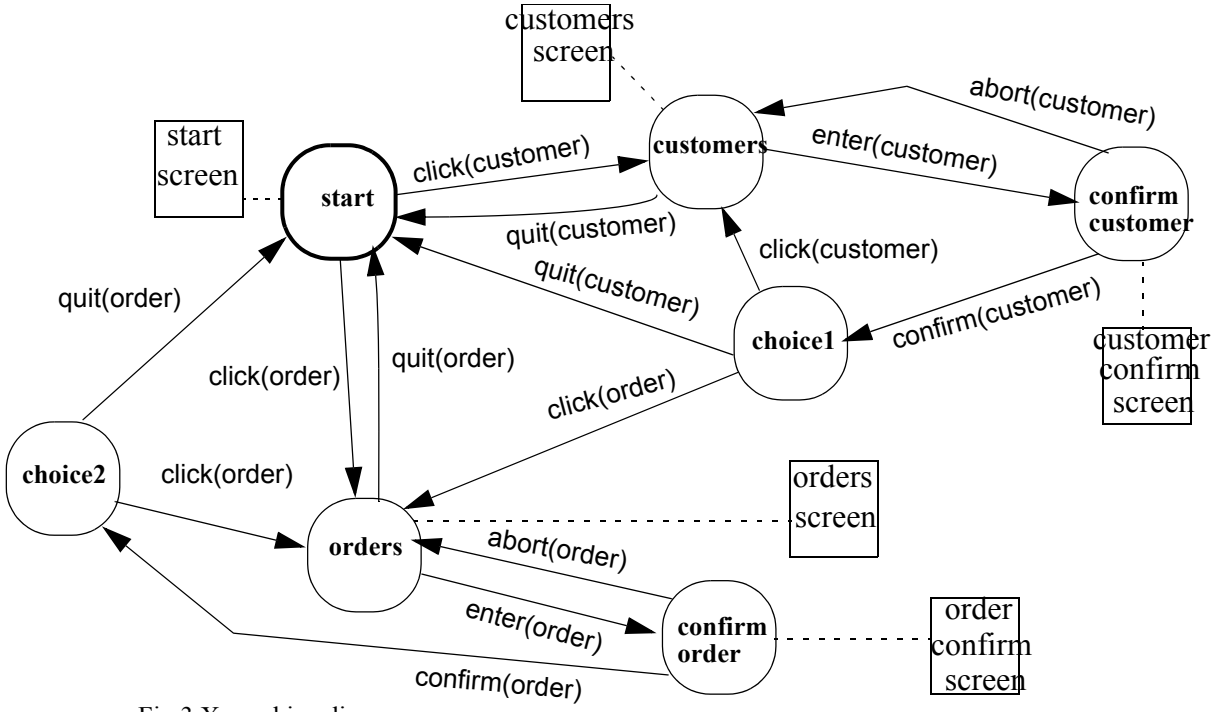


Fig.3 X-machine diagram

2. The functional testing strategy.

Each requirement in the requirements document should be traced to a test or tests. Since our requirements have been numbered and defined on story cards it is important that each test should have a number which identifies which requirement it is testing for. The requirements have been integrated into a dynamic machine based model which defines the operational relationships between them. this model will now provide us with the system level test sets. It is important to identify these at this stage. The testing of the individual requirements or the units/classes that implement them is covered in a later Chapter.

We identify paths from the start state and derive a test for each path. However, we do more than that. Notice that the paths through the machine involve driving the system between the states by carrying out the various functions that are available at each state.

As we have seen we start at the state **start** with the initial state of the internal memory, probably in some basic initialised state, and the aim is to visit every state in turn. When we have reached a state we need to confirm that it is the correct state and this is done by following more paths from that state until we get outputs that tell us, unambiguously, what the state was. Then we repeat the path to that state and check what happens if we try to apply every basic function from that state, some will have succeeded but some should fail. Have the correct ones passed and failed? This is then repeated for every state.

So we could consider the path obtained by operating the following functions:

click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; confirm(customer) ; click(order) ; enter(order) ; abort(order) ; quit(order)

At each point in this sequence we will be submitting data values or mouse clicks and we will have to choose these to enable the test to be carried out. We expect certain things to happen and these have to be recorded and the test will be evaluated in respect of detecting what actually happened and seeing if this matches the expected behaviour. Did the buttons work, do the correct screens get displayed, was the correct data put into the database, were the correct error messages displayed (if appropriate) and the correct screen displayed subsequently etc.?

This test tests what should be there. However it often happens, particularly with OO programs that the system can do unexpected things which were not planned for. We need also to test that *the functions that are not supposed to be are not there!*

As an example, once we have reached state **customers** it should not be possible to use any of the order functions that are available for the **orders** part of the system. we could do this by trying to see if we can make these functions work as part of a test. So we ought to test that the data we entered for the customer does not also get put into some other part of the database dealing with orders.

Thus we can assemble a set of test cases based on paths of various lengths through the machine diagram and tests of the non-availability of functions in certain states. Here are some more examples.

```
click(customer)
click(customer) ; enter(order)
.
.
click(customer) ; enter(customer)
click(customer) ; enter(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer)
click(customer) ; enter(customer) ; confirm(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer)
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; enter(order)
.
.
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; abort(customer)
click(customer) ; enter(customer) ; confirm(customer) ; click(customer) ; enter(customer) ; abort(customer) ;
enter(order)
.
.
etc. etc.
```

Recall that the enter(order) test will look at the orders database to see that nothing has changed. The other functions of the system would also feature in a similar way.

Clearly this will lead to a lot of tests and automation is required to manage the size of the test set. In industry, sometimes very expensive, test tools and environments are available to generate tests, to apply tests and to analyse test results. Some of these can be found on the Internet and we have some test tools that support this approach to test set generation.

Another, rather draconian, general test is to reboot at an arbitrary point in the program, this is important since some users may panic and do this, we need to ensure that the minimum data is lost in this situation.

3. The full system test process.

We assemble the full test set in stages.

Firstly we return to the X-machine diagram, Figure 3. We will look at a part of the diagram to explain the process.

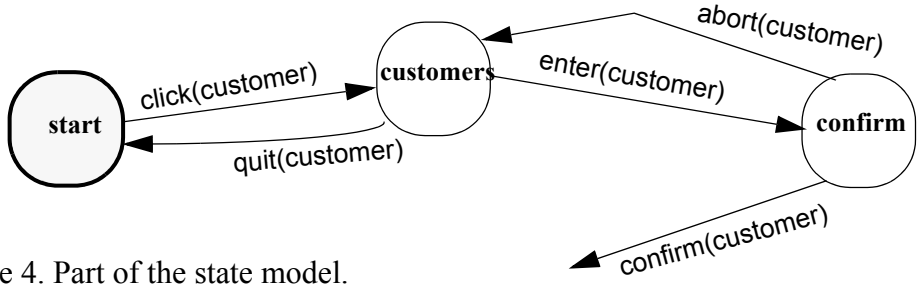


Figure 4. Part of the state model.

What is our basic strategy for testing? If you look at the diagram, which represents what we want our software to behave like then there are a number of ways in which we could have faults. We could find that a transition does not operate from the desired start or source state to the desired target state, for example, perhaps the event click(customer) leads us to the orders state, or to some other state. The output we were expecting was the **customer** state with its screen. Perhaps the function enter(customer) fails to correctly accept the customer details - maybe they are just lost when we enter them. Perhaps there is no **confirm** state and the transitions that are supposed to go there go somewhere else. Perhaps there are states that we did not intend to exist within our software. Some extra states which cause the system to behave wrongly. In the diagram there is a **bad_state** state which should not exist, it is unclear how we get into this state but the confirm(customer) function can only operate from this state and not from the correct state, **confirm.**

To summarise, the software can differ from the machine model in a number of ways:

1). there are *too few* states;
2). there are *too many* states;
3) there are transitions going *from* an incorrect state;
4). there are transitions going *to* the wrong state;
5). there are transitions that carry out the *wrong function*.

Our tests will expose all these faults.

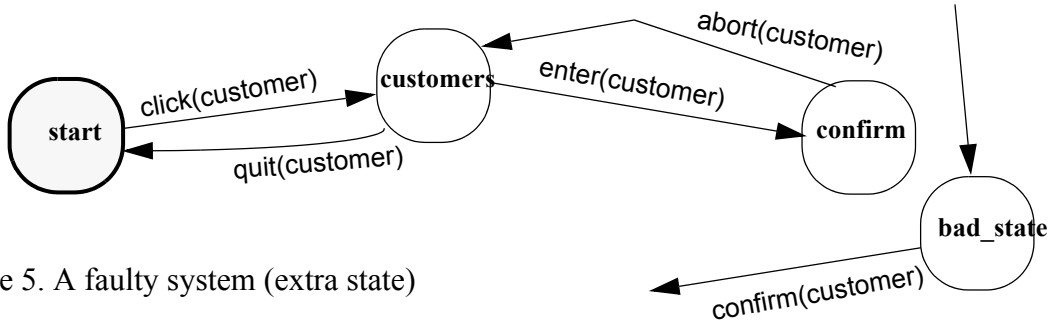


Figure 5. A faulty system (extra state)

We are going to build a number of sets of function sequences as we did above but in a systematic manner.

The first set is called the *transition cover*. What this consists of is a set of sequences that systematically work though the state space of the machine from the initial state. We start with the shortest sequences and extend them by trying out ALL the functions from the state we get to in turn. Then we take each of these sequences and for those that should lead to another state we then extend them by all the defined functions. We will look at the outputs from the software when we apply these sequences to see what happens, does it produce the right results?

Let's look at part of the machine in order to understand the process.

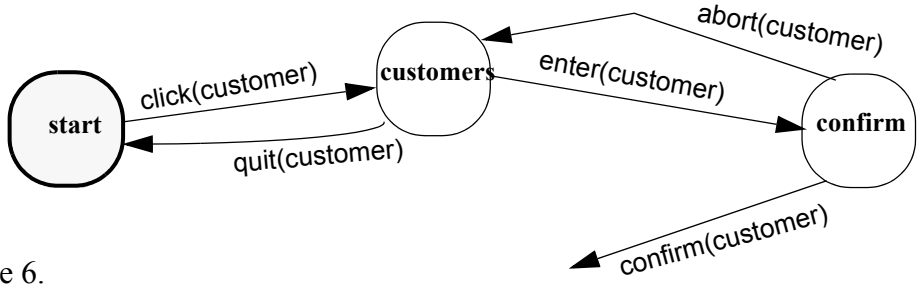


Figure 6.

We start at the state **start**. The first test is the click(customer) event which should take us to the **customers** state. Now, how do we know that we have reached the **customers** state rather than some other state? We will have to test for this separately. We should have already tested the function beforehand, this would be part of out unit testing process which will be described later. If the function on the transition is more complicated than this it might require a more complex use of this testing technique, this is not discussed here, see [Holcombe 1998] for further details. Having observed the results of this simple test we now introduce some more. These consist of applying ALL of the possible transition functions after we have reached the **customers** state.

e.g.. click(customer)
click(customer) : quit(customer).
click(customer) ; enter(customer)
click(customer) ; abort(customer)
click(customer) ; confirm(customer)
click(customer) ; enter(order)
click(customer) ; confirm(order)
click(customer) ; abort(order)
.
.
.
etc.

Only the first 3 test sequences are legitimate, the rest should generate failures during the testing. We need to check these out because it is important that the software does not do anything unexpected, checking that it does what it is supposed to do is only half the story. We also need to show that it doesn't do what it shouldn't do!

Now we consider the position from the **customers** state. We know how to get to this state and we have to check that only the expected transitions operate from it and these have the right behaviour. So we will try sequences such as:

```
click(customer) ; enter(customer) ; confirm(customer)
click(customer) ; enter(customer) ; confirm(customer) ; enter(order)
.
.
etc.
```

We would only expect the first set to work the others should fail.

An algorithm for determining the transition cover.
Much of the process for generating and applying test sets to real cases can be automated. We consider the case of constructing the transition cover.

We first build a *testing tree* with states as node labels and inputs as arc labels.

From each node there are arcs leaving for each possible input value. The root is labelled with the start state. This is level 0.

We now examine the nodes at level m from left to right;

if the label at the node is a repeat of an earlier node then terminate the branch;
if the node is labelled "undefined" then terminate that branch.
if the label at the node is a state such that an input *s* is not defined then an arc is drawn, labelled by *s*, to a node labelled "undefined".
if an input *s* leads to a state q' then insert an arc, labelled by *s*, to a node labelled q'.

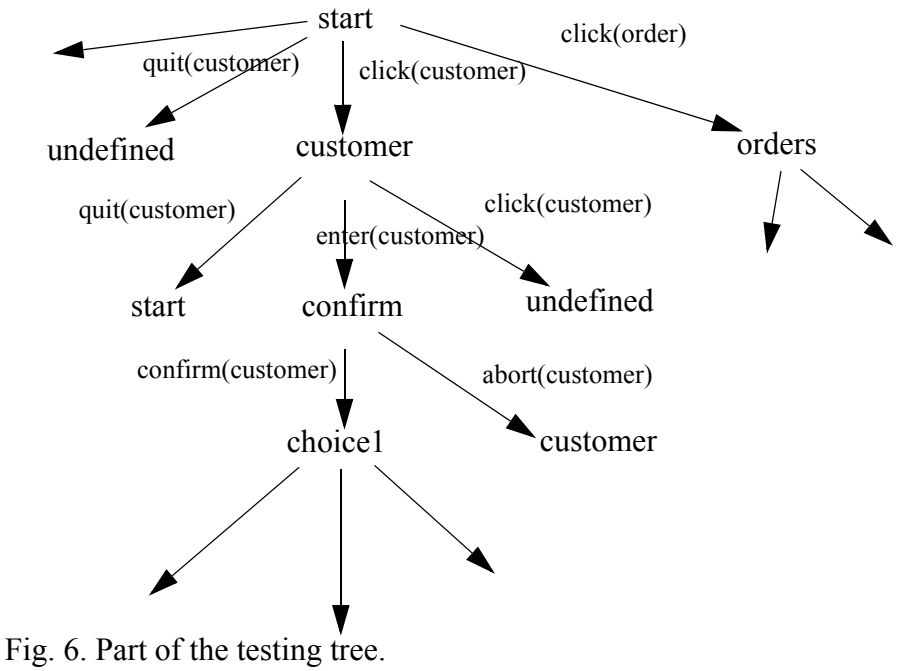


Fig. 6. Part of the testing tree.

The test sequences we need can be read off as labels of the various paths through the tree.

This process continues. It will detect many of the faults in the software but there are still things we need to do. We need to check that the state that we have reached at the end of the test sequence that we have applied is the correct state. Unfortunately we cannot just look to see what state we are in, the software is like a black box, we can only see what goes into it and what comes out. We need to add some more operations at the end of our test sequences in order to ascertain the state we have reached, and thus know whether the software is behaving correctly or not.

The next set we need to work out is called the *characterisation* set. This will consist of a set of sequences that will enable us to distinguish between any two states in the intended system.

To work out this set we need to look at the machine diagram.

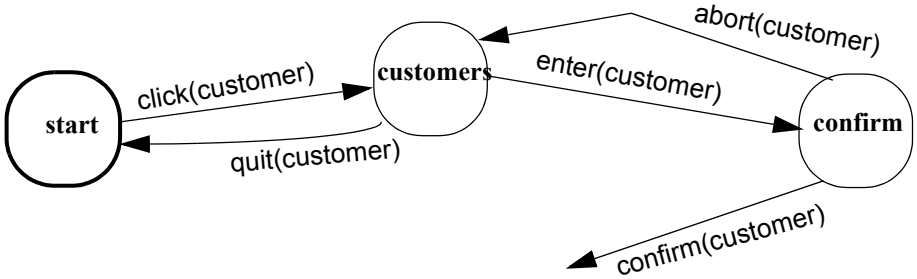


Figure 7. Part of the machine model.

Consider the states **start** and **customers**, the functions click(customer), enter(customer) produce different observable outputs from the two states, in the first case the first function should lead to a customers screen and the same function should have no effect on the customers screen. using the second function in the two states will result in the confirm screen in the case of the state **customers** and nothing in the case of **start**.

We choose for our characterisation set a collection of functions (transitions) that can distinguish between the states.

Having reached a particular state we then apply values from the characterisation set, the results will confirm what the state was that we reached.

We now need to estimate how many more states there are in the implementation, than in the specification. Let us assume that there are **k** more states.

The shorthand *A* is used for the collection of all possible transitions in the machine model. Let *W* be a characterisation set, it consists of a number of short sequences of transitions.

Now choose any transition from the machine and apply that followed by one of the transitions from the characterisation set, *W*. This will provide a sequence of 2 transitions, one from *A* and one from *W*. We do this for all possible combinations of transitions from *A* and transitions from *W*.

Thus we have moved in the machine from the start state to another state using the first transition and followed it up with an element from the characterisation set. This will tell us what

state we have reached - if the software is faulty we may have taken the transition but gone to the wrong state.

This is the key idea. Try all transitions from all states and then try to see where we have got to. The assumption about the number of possible extra states in the implementation is used in the following way. Suppose that this number of possible extra states is **k**. We then apply all possible sequences of transitions of length **k** each followed by transitions from the characterisation set. This is part of our full test set and can be described in the following mathematical formula.

$$Z = A^k W \cup A^{k-1} W \cup ....\cup A^1 W \cup W,$$

that is we form the set of sequences obtained by using all input sequences of length **k** followed by sequences from *W*, then add to this collection the sequences formed using input sequences of length **k-1** followed by sequences from *W*, continue building up a set of sequences in this way.

The *final test set* is *T Z* where *T* is a transition cover. This set consists of any sequence from the set of tests in *T* followed by any sequence from the set of tests in *Z*. We do this for all possible combinations.

Clearly this will lead to a lot of tests and automation is required to manage the size of the test set. There are some test tools that support this approach to test set generation. [see http://www.dcs.shef.ac.uk/~wmlh]

This particular approach to testing provides us with an extremely powerful set of tests, tests that will find almost every fault that could exist in the software. The exercise to this Chapter takes a more detailed look at a specific example.

4.Test documentation.

It is vital that all the tests are properly documented so that testing can be carried out systematically and effectively. We also need to keep a record of the results so that the quality assurance can be convincing. Maintenance will also require information about the testing results.

For each requirement, which should be properly numbered in the requirements document, we will generate a set of tests. The details should be kept in a suitably designed spreadsheet.

Here is an example:

Requirement	Test reference	Test purpose	Test input	Constraints/ prerequisites	Expected output	final state	comments
1.1.1	1.1.1	test front page	load program	browser open	page loads	start	
1.1.2	1.1.2.1	load customers page	click(customers)	start page open	customers page displayed	customers	
	1.1.2.2	load customers page	type random keyboard characters	start page open	no change in display	start	invalid input

Require ment	Test reference	Test purpose	Test input	Constraints/ prerequisites	Expected output	final state	comments
	1.1.2.3	load custom-ers page	reboot	start page open	close down, data-base unaffected	-	invalid input
1.1.3	1.1.3.1	enter cus-tomer details	standard data entry1*	customers page open	data displayed	confirm	
	1.1.3.2	enter cus-tomer details	standard data entry2*	customers page open	data displayed	confirm	
	1.1.3.3	enter cus-tomer details	standard data entry3*	customers page open	data displayed	confirm	
	1.1.3.4	enter cus-tomer details	empty data entry	customers page open	error message	customers	invalid input

Table 1. Systems/acceptance test definitions

* The definitions of standard data entry1, standard data entry2, standard data entry3 need to be made somewhere in an appendix to this table.

The next stage is to try to automate the testing as far as possible. We need to create a file of

Test ref.	Function sequence/path	Test sequence	Expected output	Final state
.....				
2.3.2.1	click(customer) ; enter(customer); enter(order)	<click(customer)>; <enter("standard_data_entry1")>; <enter("order_ details")>	no change to d'base	confirm customer
.....				

Table 2. Test data file

test inputs, one set of inputs for each test. These could be kept in a spreadsheet, the test data file - Table 2 - and a script written to extract these inputs and put them into a standard text file, one line per test. Another script would extract each input sequence and apply it to the code. This is sometimes easier to do in some cases than others. With GUI front ends it is sometimes difficult to access the key parameters/events from inside like this and anyway one would want to test the overall programme as well. Test software is available, at a price, to automate a lot of the interface interactions but for university projects it may be necessary to rely on manual techniques.

Test ref.	Date/ personnel	Result pass/ fail	Fault	Action	Comments
1.1.2.1	12/3/02 Pete	P	-	-	-
1.1.2.2	12/3/02 Pete	F	system crash	debug	Jane alerted (13/3/02)
1.1.2.3	12/3/02 Pete	P	-	-	system closes - no losses

Table 3. Test results table - system version 1.0

The test results file - Table 3 - is a vital resource which will have to kept up to date during testing. It describes what has been done, what has been fixed and what remains to be done.

5. Design for test.

Sometimes, it is hard to test a program because it has not been designed to make testing easy. This will usually result in a poor quality program since testing is very expensive - as you will find - and many software developers will stop testing, not when the system is suitable for release or delivery, but when they run out of money in the test budget. Often they take a risk that the cost of fixing the client's bugs later, or of supplying patches, is cheaper than continuing testing in-house. The client ends up doing some of the testing and they may not appreciate it!

In order to make the testing easier we introduce two strategies that will help. These are called *design for test* principles.

Design For Test Principle 1 :*Controllability.*

This amounts to designing the X-machine of the system so that you can access any state with any value in the memory. It can be achieved by using a special test input to do this.

This issue mainly arises when there are functions that exist in several states of the machine. We wish to send data (inputs) directly to them without going through intermediate states which may change the internal memory in ways that will not allow the functions to be fully tested, for example preventing the preconditions to be satisfied or violated in some way.

See the diagram below:

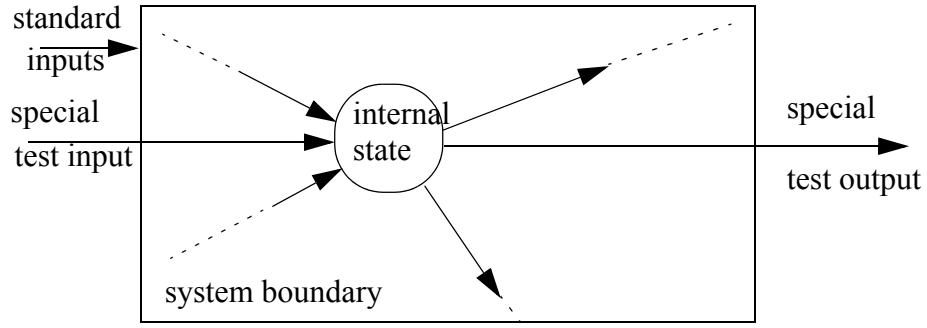


Fig. 4 Illustrating how to achieve design for test compliance.

We write special code to access this function under the conditions that we need, setting, for example, internal variables to suitable values.

Design For Test Principle 1: *Observability.*

This problem arises when we have carried out a test but we are not sure which function has operated and what it has done. The outputs might have been masked by other activity. The solution here is to define a special output value which is used to determine if the test has run properly. We therefore write some extra code which will print out, for example, some critical variable values, messages which will tell us what has happened etc. This is a common practice in programming where you often interrogate a variable to see what its value is etc. during debugging.

In both of these cases we have code in the implementation that is used only for testing and is not part of the original requirements. We can either leave it there or remove it - comment it out, for example - but whatever you do it needs to be done with care or else it might break the system!

6. Non-functional testing

Although the principal purpose of the system test is to confirm that the functional requirements have been met it is also necessary to consider the non-functional requirements and quality attributes. We will establish compliance with these also through suitable types of testing. This is done prior to final delivery of any version. We can regard the testing of the non-functional requirements together with the testing of the functional requirements as playing the role of the acceptance tests for the software. This needs the active involvement and the agreement of the customer.

Let us look at some of the non-functional requirements mentioned in the previous chapter.

Reliability:

For a single user, the system should crash no more than once per 10 hours.

For the first requirement there is very little alternative to just running the system and logging any problems where functionality is lost. Other approaches would be to examine the technology in use, age and type of work stations and servers, type of software technology used, in particular

how stable it is and what is currently known about its reliability. Demonstrating compliance with this requirement will be difficult within the constraints of this type of project.

> The system should produce the correct values for any mathematical expression 100% of the time.

Showing that the calculations, if any, are always correct is pretty well impossible, one can log errors if they arise during final testing but there is very little more that can be done in a practical way.

> If the system crashes, it should behave perfectly normally when loaded up again with minimal data loss.

It is easy enough to crash the system, carrying out a reboot for example, and this can be the basis for this type of test. what is meant by *minimal information loss* needs to be thought about. A bare minimum would be no loss of any data that has been committed to the database. If some temporary recovery files can be developed this would be better but probably beyond the scope of the project.

Usability:

> A user should be able to add a new customer to the system within 1 minute.
> A user should be able to add a new order to the system within 1 minute.
> A user should be able to edit a customer's details within 5 minutes (will vary with details type).

We need to define a user. It might be best to consider the sorts of qualifications and experience that a typical user might possess. For example - left school at 16, successfully completed an initial secretarial and office course, 3 years experience with MS Office and so on. The test would then be to find a number of people, perhaps some of your friends and relations, and to get them to try these tasks on the systems a few times. What we are looking for is the number of mistakes in carrying out the task, the time it takes and any apparent confusion observed during the session. This could indicate that there are problems with your user interface.

> A user should be able to produce reports and statistics within 1 minute.

For this requirement we need to specify what sort of reports and statistics are meant. Then we can ask a user to see if they can do the task.

Efficiency:

> The system should load up within 15 seconds.
> The time taken for the system to retrieve data from the server should never exceed more than 30 seconds.

These requirements can be checked directly by measuring the time for these activities to complete. They should be tested on a number of occasions and under a number of conditions - database containing a few entries to one with many to approximate to the intended operational context of the software. To do this it is best if the data that is loaded into the database is similar

in nature to the client's intended data. If this is not available then you should write a script to generate suitable data.

Portability:

The client system should work on the client's current computer network which is connected to the Internet and has got at least Windows 95.

This may not be so easy to test as it seems, it depends on whether you have access to a system similar to your client's. It is very easy to find that the software works perfectly on one system but not on an apparently similar one. This is particularly true of PCs and Windows, of MS Office based products using for example Visual Basic and for Java programs. It is important that all the ancillary files and directories are availabel and in the right place on the client's system.

The system should be easy to install.

The definition of this needs some elaboration. The install process must be defined. It might mean inserting a CD and following simple on-screen instructions. If this is the case then it has to be carried out on a number of occasions by a number of people to see that it does work.

A final area where we need to test is the User Manual. We will describe this in more detail later but mention it here to emphasise that it will need careful thought and someone needs to review it, preferably not someone who wrote it. In the spirit of XP it could be the client but any drafts should be checked by the team beforehand.

7. Testing internet applications and web sites.

There are many issues relating to testing these types of applications.

Users of the web site could be using one of many different types of platform, for example, PC, Mac, Unix as well as different browsers, Netscape, Internet Explorer etc. It is best if the system interface can be tested under all these combinations of platform and browser. It is surprising how different some web pages can look under different circumstances.

Where the users are can also be a factor, not just their geographical location but how they connect to the internet.

The number of potential users is also an important issue. Your client may have an Internet Service provider offering a service is this sufficient for their needs when the system is up and running?

Among the load measures that affect the operation of the site are:

static: hits per day, page views per day, unique visitors per day.

dynamic: transactions per second, MB per second, number of concurrent users, number of session initiations per hour.

Load profiles need to be estimated based on the profiles of the potential users as well as dealing just with the volume of users. Some transactions occur more frequently than others and a test script should acknowledge this. Browsing is more frequent than buying. Thinking time is also a factor. Users arrive and leave at random. The rates are not related, the time it takes a site to respond can affect subsequent behaviour with customers abandoning slow sites in favour of faster ones. Downloading large graphics files over a slow connection can be a disaster. Graphics files should be optimised to suit the conditions. Huge swings in usage are often found. For e-commerce browsing peaks in early evening, purchase commitment and validation peaks at lunchtime. Time zones also affect things.

There are a number of client network connection options - multiple connections open (Netscape), buffer size options etc. all affect performance. The use of http v 1.1 over http v1 is also significant. Client preferences can affect behaviour and the configuration of the browser comes into play here for example is javascript on/off, graphics on/off, cookies on/off, cache sizes, encryption etc.?

The service provision ISP companies is organised in tiers:- Tier 1 ISP (e.g. AT&T), Tier 2 ISP (e.g. AOL), Tier 3 ISP (e.g. local ISPs) - each further from the backbone. How many Tier 1, 2, 3 users are amongst the user profile? Your client may have to investigate this with his/her business and marketing advisers.

Background noise can also affect performance - client virus detectors, intruder detectors, e-mail etc.all take up processing resource and may slow some sites down on some machines. Geographical locations - response times vary around the world.

All of these variables leave us with a real problem of modelling the load and testing for it.

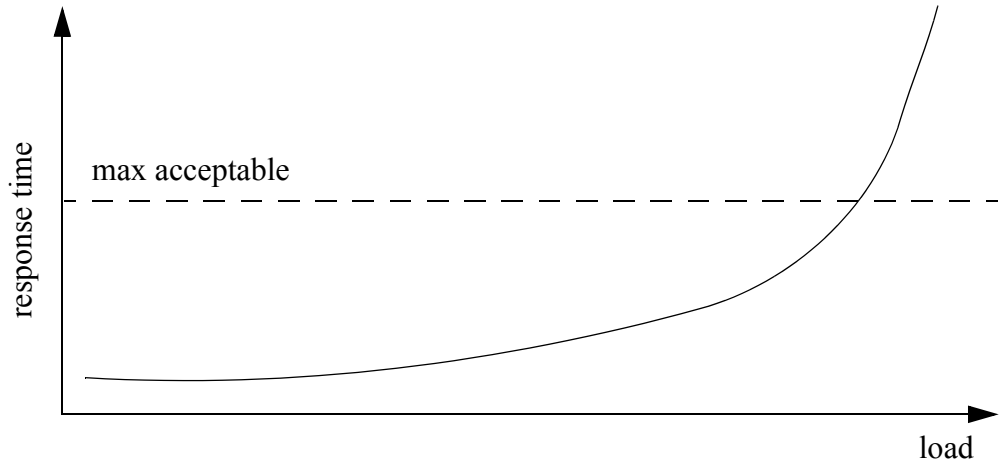


Fig. 5. A typical response/load graph

Fig. 5. provides a simple picture of how the response time is affected by the load on the servers. In order to test a system we could try to identify the worst set of parameter values to define a user profile and the best case. These give extreme performances values as in Fig. 6.

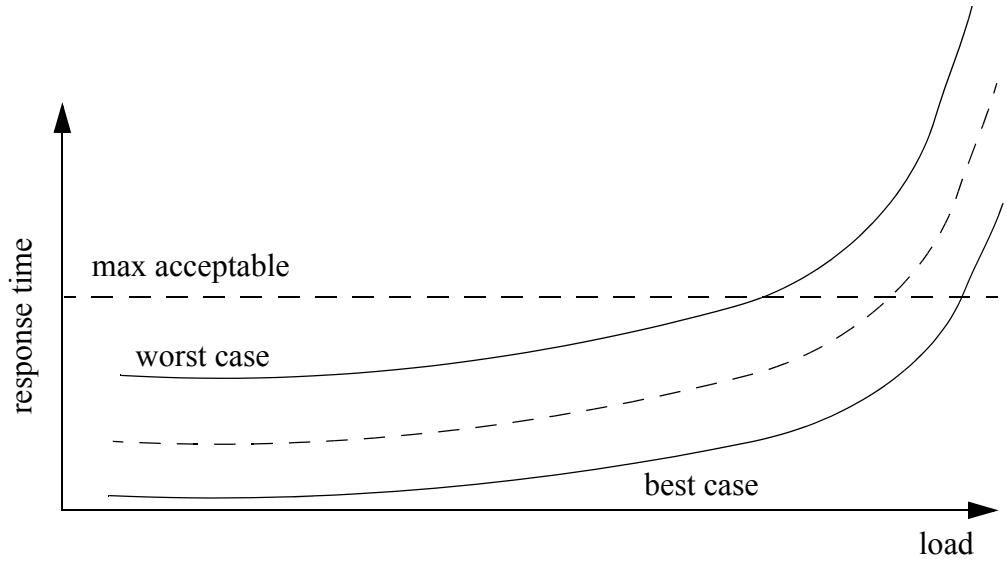


Fig. 6. Typical response/load graph with best and worst case profiles.

We might decide where in this region we wish to establish our typical user mix is and test this for compliance with our desired performance requirements.

This is rather a specialist area and may be beyond the scope of the project. However, it is useful to be aware of some of the issues.

Building an e-commerce site introduces a number of risks for businesses. It allows for possible connections to internal company systems, accounting, customers, orders and other confidential and critical content. This can be attacked, stolen etc. If WWW users can access part of the company network then it is important that suitable security checks are in place. Internal hackers/trojan horses are the single biggest threat. all businesses should be aware of this and you may like to bring this to the attention of your clients if you think that it could be an issue for them. They will need to seek professional advice.

8. Review.

There are many aspects to testing, we have only just scratched the surface. Later we will look at unit testing and testing for non-functional requirements. For further information about the type of testing described here consult the following book: [Holcombe1998].

A final series of tests that could be carried out on a completed system is done by repeatedly trying out arbitrary input values and arbitrary mouse clicks at all stages of the operation of the system. It is a type of random testing that seeks to break the system by creating unusual combinations of events. It can be quite effective.

Testing non-functional requirements is also vital. If the system is too slow or too hard to use it will be a failure and that is not what we want.

Some projects will involve the building of a web site, perhaps with a database back end. Whereas we can test an in house system reasonably well it is much harder to test if it is to be

available on the internet. The testing of such web sites is a specialist activity and requires a lot of understanding of the technology and of the key issues at both the client end and at the server. For critical e-commerce business there are many security threats also. You have to know what you are doing. We will consider this later in this Chapter.

Conundrum.

Two leading supermarket chains introduced their first internet ordering system at around the same time. Their e-commerce sites, although superficially looking similar, fared rather differently. One saw a much greater growth in business than the other. Yet the technology used, the warehousing and delivery systems were very comparable. Customers just didn't like using one of the sites.

What could have been the differences between the two user interfaces that made this happen? (It was nothing to do with the look and feel of the web pages or the way that the orders were managed or the price of the goods.)

Exercise.

This exercise works through a simple test generation example in the form of a learning exercise. You may wish to refresh your knowledge of the mathematical notation by referring to a book on discrete mathematics or formal languages and machines.

Consider a simple machine with 4 states. There are 2 functions: a and b.

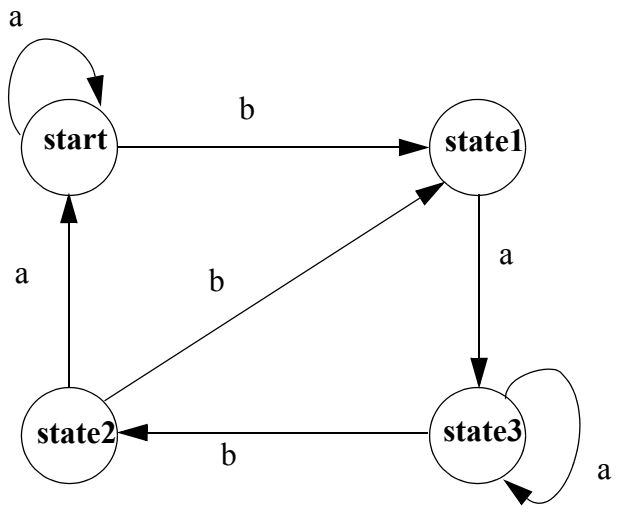


Fig. 5. A simple finite state machine

Putting in a stream of functions, say ababab the result is a transversal of the diagram to **state1**. If there is a state where a given function fails to operate then the machine will halt, e.g. abb starting from start halts in state **state1** after the second function is applied since function b is not defined in this state.

Constructing a test set.

The test generation process proceeds by examining the state diagram, minimizing it (a standard procedure) and then constructing a set of sequences of functions. This set of sequences is constructed from certain preliminary sets.

We require some basic definitions, these apply to any finite state machine.

Distinguishability. Let *L* be a set of function sequences, and q, q' two states then *L* is said to distinguish between q and q' if there is a sequence *k* in the set *L* such that the output obtained when *k* is applied to the machine in state q is different to the output obtained when *k* is applied when it is in state q'.

Minimality. A machine is minimal if it doesn't contain redundant states. There are algorithms that produce a minimal machine from any given machine - the minimal machine has the same behaviour in terms of input-output as the original.

Example.

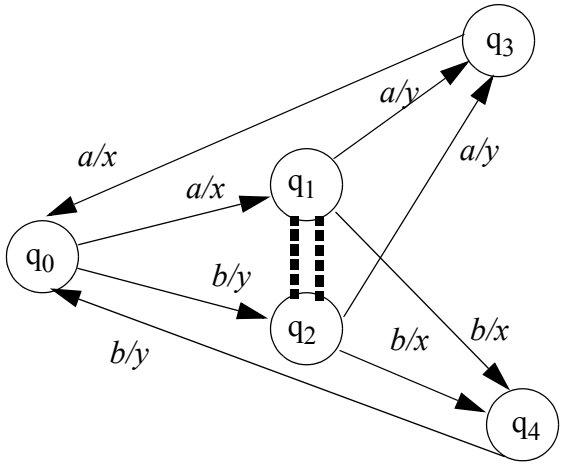


Fig. 5. A simple finite state machine which is not minimal.

In this machine states q_1 and q_2 can be merged to form a machine with fewer states and the same input-output behaviour.

Let us consider, from now on, a minimal finite state machine.

A set of input sequences, W, is called a characterisation set if it can distinguish between any two pairs of states in the machine

Example - in the first machine $W = \{ a, b \}$ is a characterisation set (the machine is minimal).

A state cover is a set of input sequences *L* such that we can find an element from *L* to get into any desired state from the initial state start.

$L = \{ 1, b, ba, bab \}$ is a state cover for the first machine, 1 represents the null input.

A transition cover for a minimal machine is a set of input sequences, *T*, which is a state cover and is closed under right composition with the set of inputs *Input*, so we can get to any state from start by using a suitable sequence *t* from *T* and for any function *a* in *Input* the sequence *ta*

is also in *T*.

Here *T*= { 1, *a, b, ba, bb, baa, bab, baba, babb* } is a transition cover for the example.

Generating a test set.

We first need to estimate how many more states there are in the implementation, than in the specification. Let us assume that there are **k** more states. Let *W* be a characterisation set:

We construct the set $Z = A^{\mathbf{k}}W \cup A^{k-1}W \cup ....\cup A^{1}W \cup W$, that is we form the set of sequences obtained by using all input sequences of length **k** followed by sequences from *W*, then add to this collection the sequences formed using input sequences of length **k**-1 followed by sequences from *W*, continue building up a set of sequences in this way.

The final *test set* is: *TZ* where *T* is a transition cover.

Example.

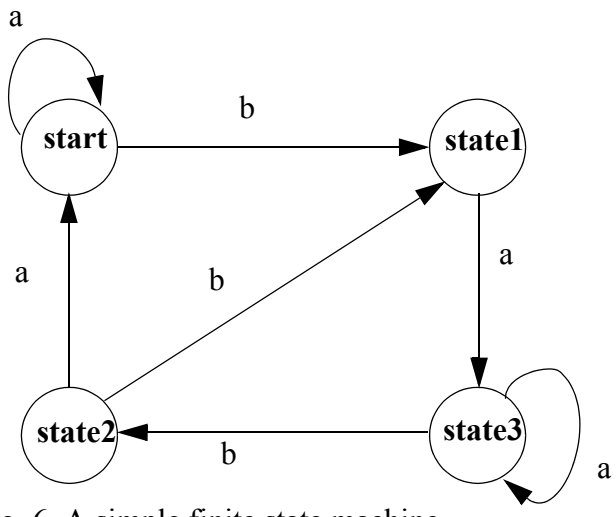


Fig. 6. A simple finite state machine.

The following diagram represents an implementation with one extra state, a missing transition and a faulty transition label (a' instead of a).

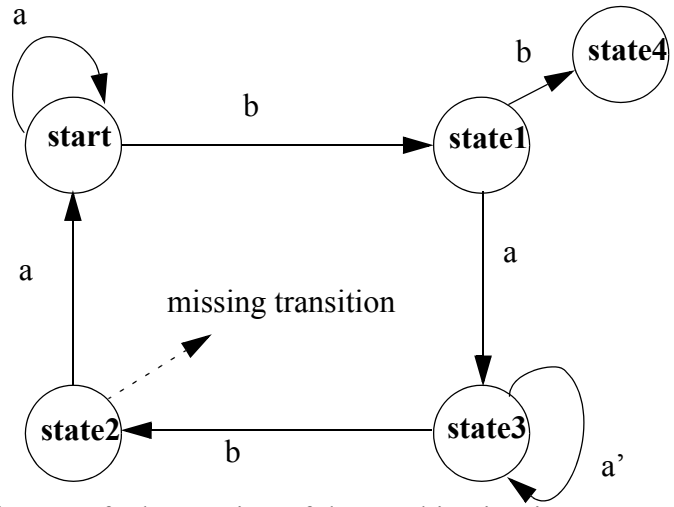


Fig. 7. A faulty version of the machine in Fig. 5

The value of **k** is assumed to be 1 for this example, the set $Z = AW \cup W$ = { *aa, ab, ba, bb* } and the test set *TZ* is thus

$$TZ = \{ 1, a, b, ba, bb, baa, bab, baba, babb \} . \{ aa, ab, ba, bb \}$$

This means the following tests:

aa, ab, ba, bb,
aaa, aab, aba, abb,
baa, bab, bba, bbb,
baaa, baab, baba, babb,
.
.
.
.
babbaa, babbab, babbba, babbbb

The extra transition is exposed by the input *bb* which produces a different output in the implementation than in the specification where no effect should be observed for the second *b*; the missing transition is exposed by *babb* and the faulty transition by *baa*.

The transition cover ensures that all the states and transitions of the specification are present in the implementation and the set *Z* ensures that the implementation is in the same state as the specification after each transition is used. The parameter **k** ensures that all the extra states in the implementation are visited.

Reference.

[Eilenberg74] S. Eilenberg, "*Automata, machines and languages*", Volume A, Academic Press, 1974.
[Holcombe1998]. M. Holcombe & F. Ipate, "*Correct systems - building a business process solution*", 1988, Springer Verlag. (available on-line at http://www.dcs.shef.ac.uk/~wmlh/correct).

[Spivey1992] J. M. Spivey, "*The Z notation: A reference manual*", (2nd. Edition). Prentice Hall, 1992.
[Jones1986] C. B. Jones, "*Systematic software development using VDM*", Prentice Hall, 1986.

Chapter 7.

Establishing the system metaphor.

Summary:

Finding the right initial architecture for the application. Three tier architectures. Deriving architecture information from the model. The Model, View, Control architecture. User interface design.

1. What is a metaphor?

Essentially we need to provide some framework within which we can discuss the software being developed in order to relate what we are building to what problem we are trying to solve. This can be achieved in many ways. It depends on what the application is.

A metaphor may consist of some of the following elements:

> an exemplar system with a similar purpose, this might be a well known package or some existing bespoke software used in the client's organisation, we will call this a metaphor *exemplar*;
>
> a general architecture of the solution expressed in terms of components or layers of software organised to communicate in particular ways, for example a *client-server architecture,* this will be an example of a metaphor *architecture*;
>
> a high level model which describes in some suitable language the essential features of some aspect of the software, it might be the structure of the user interface, communication protocols, database structure and so on, we will call this a metaphor *model*;
>
> it could be a class diagram or some similar organisation of the software components into a structured and hierarchical representation.

The metaphor also needs to consider the system boundary, where the software system meets the rest of the business, the business personnel, not just the system users, the customers and all the stakeholders. There are dangers at these interfaces if critical aspects are ignored. We need to think about some of the issues carefully.

2. A simple common metaphor.

If we think about the system that we have been considering in the previous chapters, a simple customer and orders database then it is possible to identify some basic components. We can envisage it in the form of a classic *three layer structure*, (Fig.1).

The user interface is represented by the machine state space and the business logic by the definitions of the transition functions. The interaction with the database is carried out through the business logic layer responding to and supplying output to the interface. For an e-commerce system the user interface is probably accessed through a web browser and the client server communication is made through the internet.

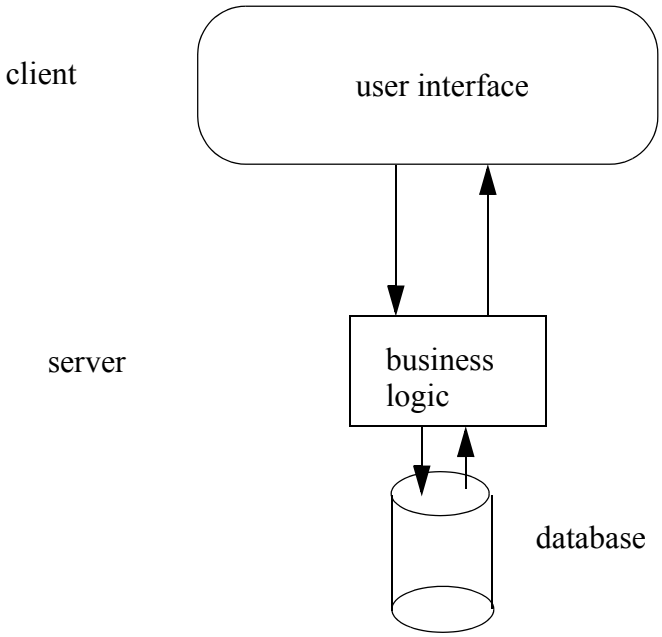


Fig. 1 Classical three layer architecture.

If we look back at the way that we developed the requirements we note that the role of the story cards was central. If we consider these then the way in which the requirements were described in terms of the inputs, the internal memory and the outputs then we can capitalise on this information in developing the system metaphor and the class architecture.

Consider, for example, as story such as:

```
enter(customer)
```

then we note that there is a series of interface events:

```
customer details entered
```

which are involved at the user interface. Thus information is communicated between the user interface and the business logic server.This carries out a number of checks on the data supplied, for example, checking that the rules defined for the data are satisfied, making sure that text is supplied when text is needed, perhaps checking Zip or postal codes are correctly formed and so on.

On confirmation of the details presented at the interface through the report:

```
confirmation details screen
```

the data is committed to the database using a suitable object database connection technology if you are using a relational database (Access, MySQL etc.) or another suitable technology if you are using XML as a database.

In a similar way, querying will involve building up a query through the user interface, this will be validated at the server and the database queried, the results being presented through the user interface in a format defined at the server.

So how do we design the interface, the server and the database?

The basic principle is to separate out the different areas of concern into layers and to ensure that the separation is clean and efficient. The Sun java J2EE model is a typical example of this idea.

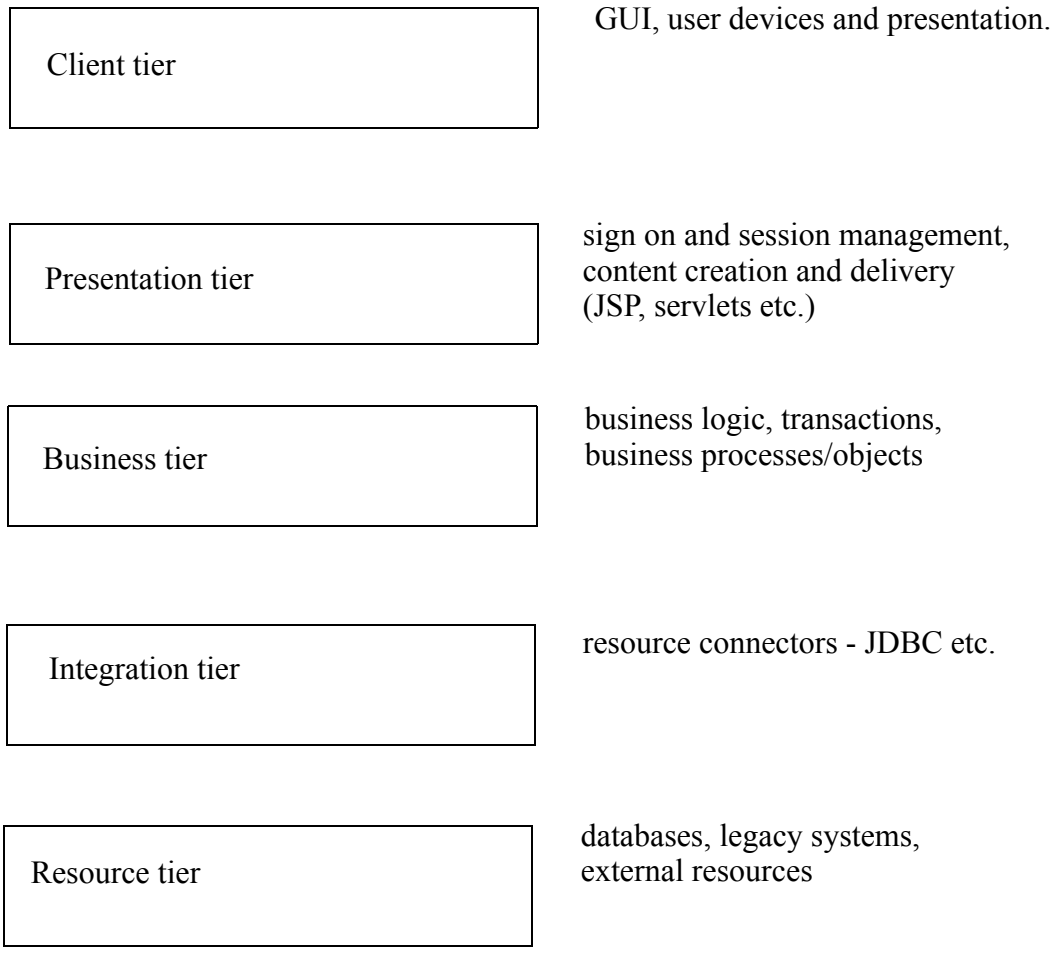


Figure 2 E-commerce tier architecture (after Sun Microsystems)

Patterns such as the above are created to permit a clear separation beyween the different technolgies needed to develop such a system and each technology has its own standards and construction methods.

Such systems should be easier to maintain since there is a clear separation of concerns, roles, behaviour and code.

In commercial systems the lower tiers of the system will service many different types of upper tiers. These upper tiers are also more likely to change with new applications. The lower levels are more stable but when they do have to be re-engineered it is expensive.

3. A simple 3 tier architecture.

For this approach we concentrate on building a database to service the application, a business logic layer containing the business rules and a presentation layer which will be based around a web browser.

For the database any suitable and available relational database system is chosen. this might be MS Access or MySQL etc.

The business rules and management system is coded in PHP, alternative approaches are ASP, JSP and servlets.

The interface is developed in html using applets perhaps in suitable ways.

Such an approach is not necessarily the most sophisticated and there may be some security issues which could need to be addressed, depending on thse application context.

For larger systems there are significant disadvantages in this simple model. The business logic component becomes very complex since it is combining a number of separate concerns which should be separated. There will be a lot of problems when it comes to maintenance in large systems. Also many large e-commerce sites involve a number of separate applications which are built upon the database and more structure is needed to prevent confusion and operational problems. This is a more advanced topic, however, we will briefly look at it later in this chapter..

4. Building the architecture to suit the application.

We have developed a model of the system and the next stage is to try and map that onto the architecture.

The best way is by referring to the example we considered earlier.

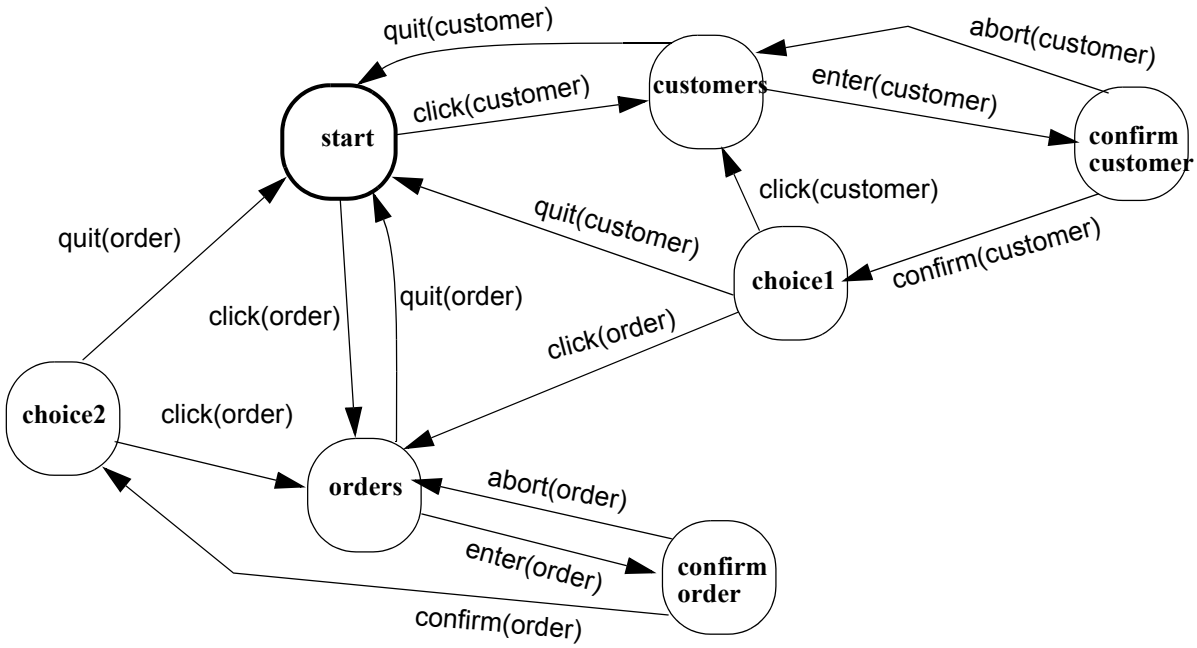


Fig. 3 X-machine diagram

where the functions are derived from the table:

story	function	input	current memory	output	updated memory	change risk
1	click(customer)	customer button click	-	new customer screen	-	low
1	enter(customer)	customer details entered	current customer database	confirmation details screen	-	medium (nature of details liable to change)
1	confirm(customer)	customer confirm button clicked	(current customer database)	OK message and start screen button	updated customer database	low
3	click(order)	orders button clicked	-	new orders screen	-	low
3	enter(order)	new order details entered	current orders database	confirmation orders screen	-	high (nature of details of orders liable to change)
3	confirm(order)	orders confirm button clicked	(current orders database)	Ok message and start screen button	updated orders database	low
3	quit()	click on return to start button	-	start screen	-	low

Table 1. Some of the X-machine functions.

The first stage is to consider the data requirements and construct the underlying database. To do this we examine the memory elements. Some of these are described at a high level and will need decomposing. For example there are customers and these have been defined in terms of their names, addresses, phone numbers etc. This is clearly a suitable structure of a database table with an appropriate customer key.

Table 1:

Customer_ID	Customer _name	Address_line1	Address_line2	Address_line3	ZIP/postcode	Phone	e-mail
ABC123	WidgetCity	345 Biz Park	Some Road	Big City	BC1 4 AS	564 583	Widg.com
ABC124	ThingArama	Corp House	Little Street	Townsville	TV2 7BB	530 986	ThingAr-ama.co.uk

Table 2. A simple customers' table

Another table is defined for the orders component and so on.

One problem with databases is that they are often hard to maintain in the sense that, if our customer comes along with some fundamental new requirements and these involve substantial changes to the tables and their relationships, then we could be in trouble requiring a major re-engineering of the database. There is no easy solution to this, relational tables are highly optimised for performance and integrity and the price that is paid for this is in their inherent inflexibility in the light of changing requirements.

Increasingly, databases are being developed using the langauge XML [Bray2000] which is much more flexible. There is technology available to connect application programmes to these XML databases. The main API's for providing an interface for XML documents in Java are JAXP, the Document Object Model (DOM), the Simple API for XML (SAX) and JDOM. All these API's offer advantages and disadvantage with no single API standing out as the standard, although JDOM is becoming popular in Java based applications due to its efficiency and ease of use, [JDOM2001]. Other techno.logy includes the Extensible Stylesheet Language (XSL), [ADLER 2001]

Now we consider the Business logic layer. the machine will provide the top level control for this section. It involves identifying the individual functions that label the transitions and which correspond to the different operational behaviours of the system. These may be at rather a high level and in that case we need to refine them to a more suitable level for implementation. For example, the function that edits customer's details involves a number of separate data entry and checking activities. We could visualise this as a submachine in the following way:

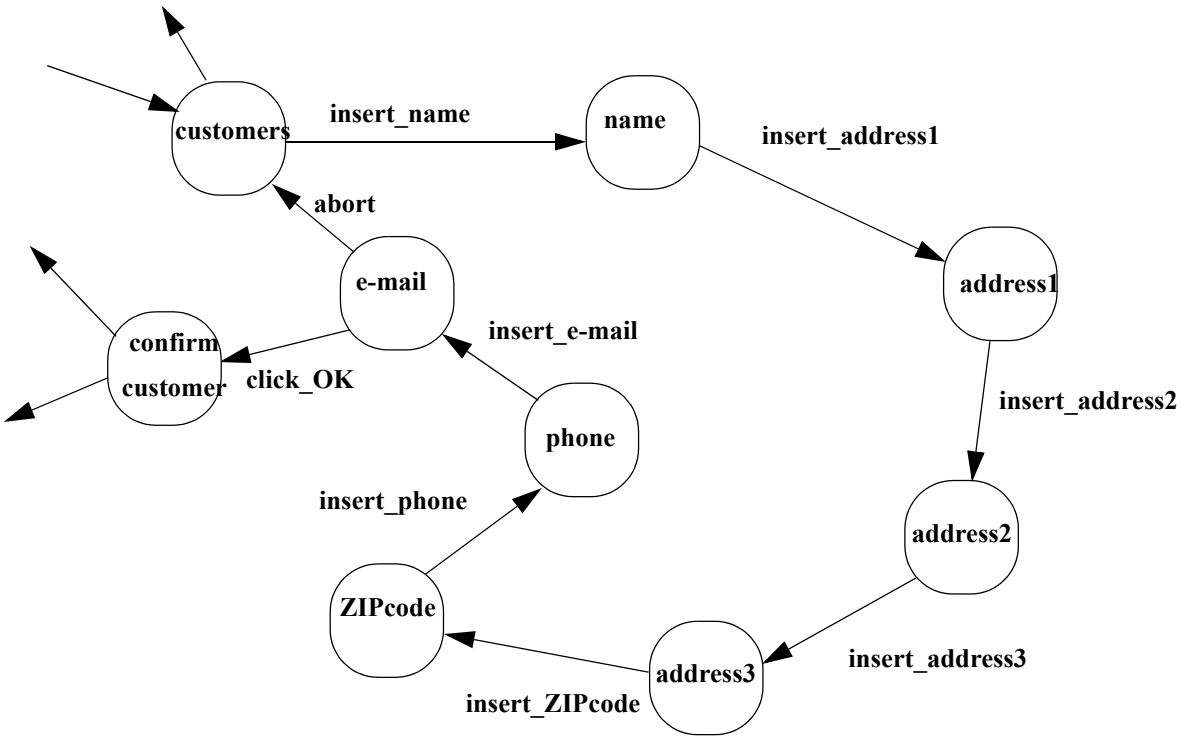


Figure 3. A refined function - enter(customer)

We have included just one **abort** function but the same function may be available from all the internal states of the function.

The next stage is to see how we can organise the implementation of these functions. We consider a number of different cases.

The simplest case is a direct interaction between the User Interface and the Business Logic layer which does not require the involvement of the database. The functions to control the slider bar, to resize the screen and to migrate between screens usning mouse clicks and mouse movements are of this nature. The general model is:

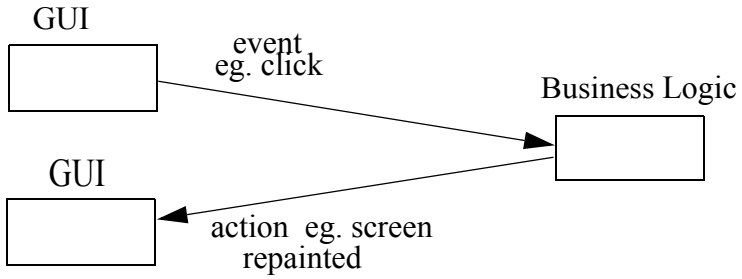


Figure 4 A simple interface action

A simple class in the Business Layer will handle this sort of function. It will listen for the specified event and change the interface accordingly.

A more complex situation will arise if the interface view has to reflect some other factor, for example to disable or “grey out” a button or data field becuse it is not valid to have access to the function beneath it. An example might be to grey out the orders button if there are no customers in the database - hence there can be no orders.

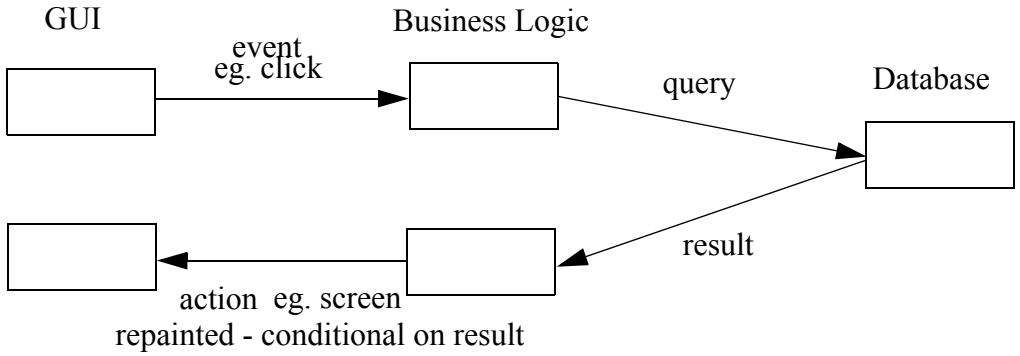


Figure 5 A conditional interface action.

Suppose that we want to describe a function that submits data, carries out a check with the database and then allows a commit action with reflection of success onto the screen.

This could be described as follows:

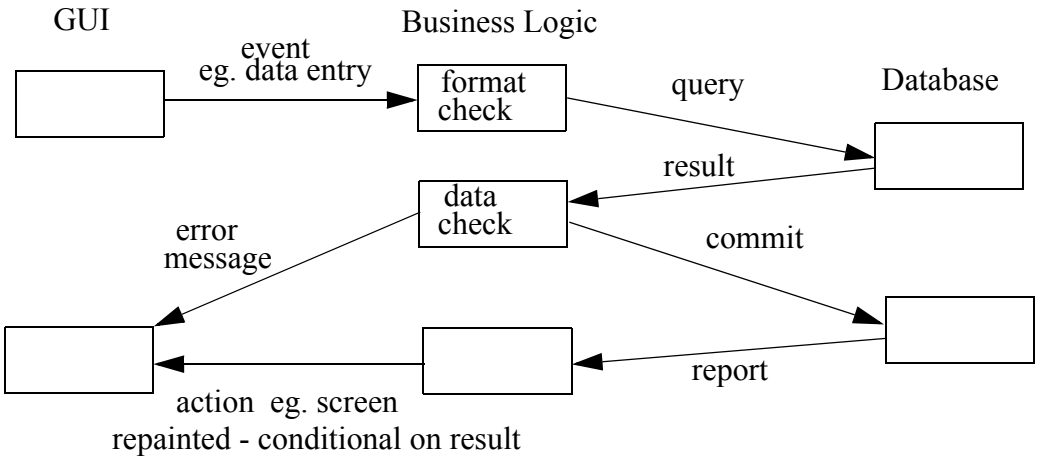


Figure 6 A data entry interface action.

The business logic needs to check that the input data is in a valid format, for example a string of characters of length less than 30, a correct date format, etc. If necessary the Business Logic will query the datbase to see if there is already data there in this field, depending on the outcome of this query the logic will either commit the data or report an error to the user interface. Following successful database update a confirmatory message may be sent to the user.

The analysis of these events and actions will provide us with some guidance on how the programme should be structured and what classes will need to be developed.

If there is a number of places where there has to be a check that the entry data submitted satisfies some predefined format, for example a text srtring of length less than 30 then it makes sense to write a class with a method that does just this. This class is then available to the other parts of the programme that requires this check to be made. This is better than embedding this check separately in all the methods that need it.

The overall machine perspective is useful for the development of a simple system metaphor. In the next chapter we will look at the way in which we might derive the class structure from it.

Since projects vary enormously it is hard to provide examples of useful metaphors that will be applicable to them all. In the next section we look at an area that provides an increasing number of projects, that of the e-commerce application.

5. Model, View and Controller - a paradigm for e-commerce.

In this approach we first discuss the *model* for the system. Ths will include the description of the state of the data and its methods independently of how it is implemented or viewed.

The *view* deals with how the component is viewed and it may be that there are several views of the same component depending on the technology (operating system) and the application. There

could be several applications sharing the same model using different views. There could be a single application using different views of the same model.

The *controller* manages the way components react to events and sends suitable information to the model to update it.

An example is the window slide bar. The slide bar is presented on the screen by the view component. The user actions of clicking and dragging the mouse over the bar will be dealt with by the controller which passes this information to the model so that the attributes chosen to represent the slide bar up updated, this will include the values chosen to represent the end points of the bar and its current position.

The MVC system approach can be explained by looking at a typical sequence of events. The user sends a request through the interface which is handled by the controller. The controller may respond by instructing the viewer to provide information to the user - such as moving the slide bar on the screen display. The controller then informs the model of the nature of the request. The model may have to interrogate the database or other resource and request a reply. This reply is then passed on to the viewer for presentation to the user.

In some examples the model is a set of Java beans, the viewer is written in JSP and the controller is a servlet. Other technologies, however are available and the decision as to which to use should be based on finding the most approriate match between the team's knowledge and experience and the needs of the client. Some problems can arise if the client has a limited technology option, perhaps because their Internet Service Provider can only support certain types of technology or their own IT administration and servers are unfamiliar with modern approaches. These issues must be sorted out before things go to far.

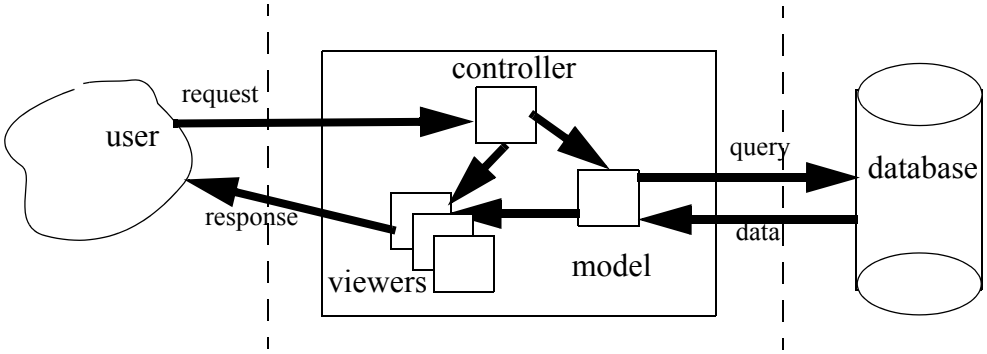


Figure 7 The MVC model in action.

Some open source resources are available to support this type of architecture. STRUTS is an example [STRUTS2001] and the book Professional JSP, [Brown2001]

Thinking back to the X-machine model it now becomes clear that this structure is entirely in keeping with what is going on in the Business Logic tier of the simpler architecture in sections 3 and 4. What we can do is separate out the different parts of the Business Logic layer in terms of what communication it is expected to carry out. The X-machine will define the overall structure of the controller, the model will reflect the current state of the system separated out from the underlying database.

Thus the model will represent the specific screens and internal variables that are of importance to the operation of the system. The viewer will act as an intermediary between the model and the GUI and will also deal with basic communication between the controller and the GUI, for example the direct interaction involving lower level unconditional events such as slide bar moves, button clicks etc.. These will cause the updating of the model so that it has a precise record of what is going on and will carry out any screen activities directly.

The more sophisticated events involving checks with the database (or the model) will be handled slightly differently. here the model needs to communicate with the database to obtain the relevant information.

Typically the controller, possibly implemented in servlets would process events and call event handlers to process the body of the request. It would be efficient to create an *eventhandler* class which could provide the basic event handling capability. Individual events would inherit the *eventhandler* and deal with specific event requirements. The events will communicate with, possibly, both the model and the view components.

The structure of the controller will be determined by the structure of the X-machine and would represent a number of *if-then-else* statements. There are design patterns which could provide this functionality (eg. request despatcher pattern).

The controller will also deal with security and error handling functions.

The true business logic is now encapsulated within the Model component, which might be written using java beans - restricted and carefuilly defined classes. These will deal with the explicit read/write functions to the database.

The model must also keep state information needed by the controller so as to maintain the overall coherence of the system.

The view layer, which might be written in JSP, is involved with the presentation to the interface and is a collection of read only functions.

6. User interfaces.

So far, the concepts that we have discussed are oriented towards the needs of the developers, when it comes to communication with the customer, however, it is essential that we use ideas that he/she can understand. Many people look upon a software sytstem from the perspective of how it presents itself to them. In other words *the system is the inteface!*

People are all different and differ greatly in the way they think and behave when using a software system. The designers of a user interface would seem, therefore, to have an almost impossible task when it comes to trying to satisfy every possible user of the system..

There is now a considerable amount of research and experience when it comes to this area. We will briefly review some of the commonly proposed principles that are recommended for the design of good graphical user interfaces (GUIs). Note, however, that if your client has some special factors, perhaps some of the users are handicapped in some way or have

other special needs, than these will have to be investigated carefully and may result in some specialist features being incorporated in the interface.

A useful general reference on user inteface design is [Schneiderman1998]

Most user interfaces consist of a collection of windows and screens. These have two main purposes, one is to present information to the use, the other is to permit the user to carry out some tasks. Naturally, many windows are a combination of both types.

If we are presenting information then there are some important principles that should be followed:

i) the information presented should not be confusing, contradictory or misleading;
ii) the words, icons and other visual metaphors used should be clear and understandable, the use of obscure technical jargon should be avoided;
iii) the screens should not be cluttered, full of irrelevant and distracting images and text, it should focus on the task in hand;
iv) the information should be up to date and presented in a consistent manner;
v) if the user is expected to do something it should be made clear what.

If the window is designed to allow the user to carry out some task then other important characteristics are desirable:

i) the action required to carry out the task should be clear, help should be given if appropriate;
ii) similar tasks under similar conditions should require similar actions;
iii) feedback should be given, if the operation was successful then this should be clear to the user;
iv) it should be possible to recover if the action was not successful, make sure that nerror message are clear and helpful;
v) too many alternative ways to do the same thing can be confusing;
vi) similar actions should be broadly consistent, so don't use radically different techniques for actions which are very similar but taking place in slightly different states.

How these windows are organised is crucial. Many simple interfaces can be modelled, as we have seen, by using an X-machine. This is well worth doing as it will relate easily to the user stories and tasks that we have been thinking about earlier. There is a balance between the desire to provide lots of information and the need to keep it clear and simple.

We also have to decide how many windows to use, too many and users find things getting tedious, too few and they may get confused. The correct level can only be found by extensive consultation and trials with prospective users or their proxies.

Since we are using XP we can expect to show our customer examples of the sort of interfaces we are thinking of using, this is an opportunity for some useful feedback. Remember, that many inexperienced custmers and users often think that the system *is* the interface.

XP stresses the need for simplicity but do not interpret this to mean that the interfaces must be very simple, they should be good but that may not mean the same. Interface design is a sophisticated skill, do not underestimate how hard it is. Test out your ideas as much as

possible on potential users or on others with a similar background. Some student friends from other departments and schools could be helpful in this respect. The more experiments you do with people the better will be the result. Don't forget that people may have very different opinions about the same interface. Set up questionnaires to get some evaluation from anyone who uses it, getting them to evaluate it on the basis of how easy it is to learn, how easy it was to carry out the key tasks, how well it kept them informed about what it had done and what needed to be done next, and whether it worked without crashing or failing in other ways. Ask users to rate the key features on a 1(poor) ... 5 (excellent) scale. Check with the non-functional requirements identified earlier.

The system doesn't stop with the interface. The system will be situated within an overall enterprise and work flows and interactions in the company may be involved with it. Some of the tasks will be manual ones and the introduction of a new system may influence these and perhaps change them. Customers should be aware of the implications of introducing the new system and should plan for it properly.

It is sometimes tempting, when designing an interface, to want to use whatever the latest technical feature that you have learnt how to implement. This will be a bad idea in many cases. Only use appropriate technoilogy, not technology for technology's sake. Adding complexity, whether from a programming point of view or from the users', will threaten rather than enhance the system.

Ask the customer or the users which input techniques they want to use in the different contexts. Do they prefer selecting from a drop down menu, clicking on radio buttons, filling in forms, using hot keys etc.? Study the sort of systems that the users are currently using and are familiar with. Keep things similar if at all possible.

Don't forget the help system, this might be a key feature for some users. It should provide some basic support to enable users to get back to a point where they can then continue. Think about the key tasks that are identified from the user stories. Perhaps use each one as the basis for a help system.

An on-line manual is also a good idea. This is discussed in Chapter 10.

7. Review.

The concept of the system metaphor is a rather inprecise one and this is both good and bad. It allows teams to develop their own approach relevant to their application domain but it then raises the issue of whether this is very helpful for maintenance when a different team may be involved. What is important is that the approach taken and the representations used are clearly explained and documented.

We have described a number of approaches to structuring the architecture of the system that makes change more manageable. The thing to avoid is an unstructured system where changes made in one area can have unpredictable consequences elsewhere. Various *layered* approaches are common these days and the separation of the data, the business logic and the presentation layers seems to make sense.

The user interface is a key component, spend as much time as you can on getting this right.

Conundrum.

A local retailer specializing in luxury goods commissioned an e-commerce system. This was completed and installed satisfactorily. The shop gave the job of printing out the internet orders and processing them through the orders system to one of the sales assistant to do at the end of their shift. This worked well at the beginning as the numbers of orders gradually grew.

After a few months of steady growth the sales figures of orders placed through the Web suddenly collapsed to nothing. Initially it was thought to be a software fault but no problems were found when we investigated.

What could have gone wrong?

Exercise.

1. Look at several different web sites and try to establish if their user interfaces meet the criteria spelt out in section 6.

References.

[Adler2001], S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles. *Extensible Stylesheet Language (XSL) Version 1.0*. http://www.w3.org/TR/xsl/.
[Bray2000], T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. http://www.w3.org/TR/REC-xml
[Brown2001]. S. Brown, R. Burdick, J. Falkner, B. Galbraith, et al, "*Professional JSP*", 2nd Edition", 2001.
[JDOM 2001], http://www.jdom.org.
[NIELSEN 02] http://www.useit.com/papers/heuristic/heuristic_list.html.
[Schneiderman1998], B. Schneiderman, *Designing the User Interface*, Addison-Wesley, 1998.
[STRUTS2001], http://www.struts.com

Chapter 8.

Units and their tests.

Summary: From stories to tasks to classes and methods. Finding the unit tests. Running the tests. Documenting the test results.

1. Basic considerations.

The nature of the system architecture and implementation languages are dependent on the application, the resources available and the knowledge of the team. Having said that, however, such a project as this is an excellent vehicle for developing one's programming knowledge and understanding, even in a new language. In fact, once one has some programming experience in any language it is usually possible to develop skills in another rapidly.

If there is a need to learn a new language then it is vital to go about it in a sensible way. One approach is to gather elementary information, introductory texts, teaching aids from the web and so on. Some members of the team should take the responsibility for organising the collection and organisation of this information and someone could then plan some presentations, demonstrations and discussion sessions with the rest of the team. It is surprising how quickly progress can be made, especially if you have a real and specific target system to develop. You may not be able to get detailed technical support from your tutors and professors but that is not critical. Any computing degree should have as one of its objectives the development of the skill to learn new things - technologies, languages, processes - and this is where this can be done most effectively. At the end of the project you will be technical experts in areas that your teachers may not know very much about!

As part of some languages there are development environments, libraries and other supporting material. It is important that these are exploited where this is feasible. It is also important to look around for examples of similar applications and to examine how these are organised. It may be that you can use this information with your own project.

2. Identifying the units.

Each project will be different, there will be different stories, different programming languages used, different operating environments and so on. Furthermore, the programming courses that have been taken may have approached the issue of breaking down a high level story requirement into "bite sized" pieces of code in different ways. It is therefore impossible top provide a definitive method that will enable the programmers to create a framework of units, classes within which the programming can be set.

One approach is to take a story and to try to identify a series of chunks of functionality or *tasks* that need to be defined and which could form the basis of some suitable units.

Consider the story from Chapter 5.

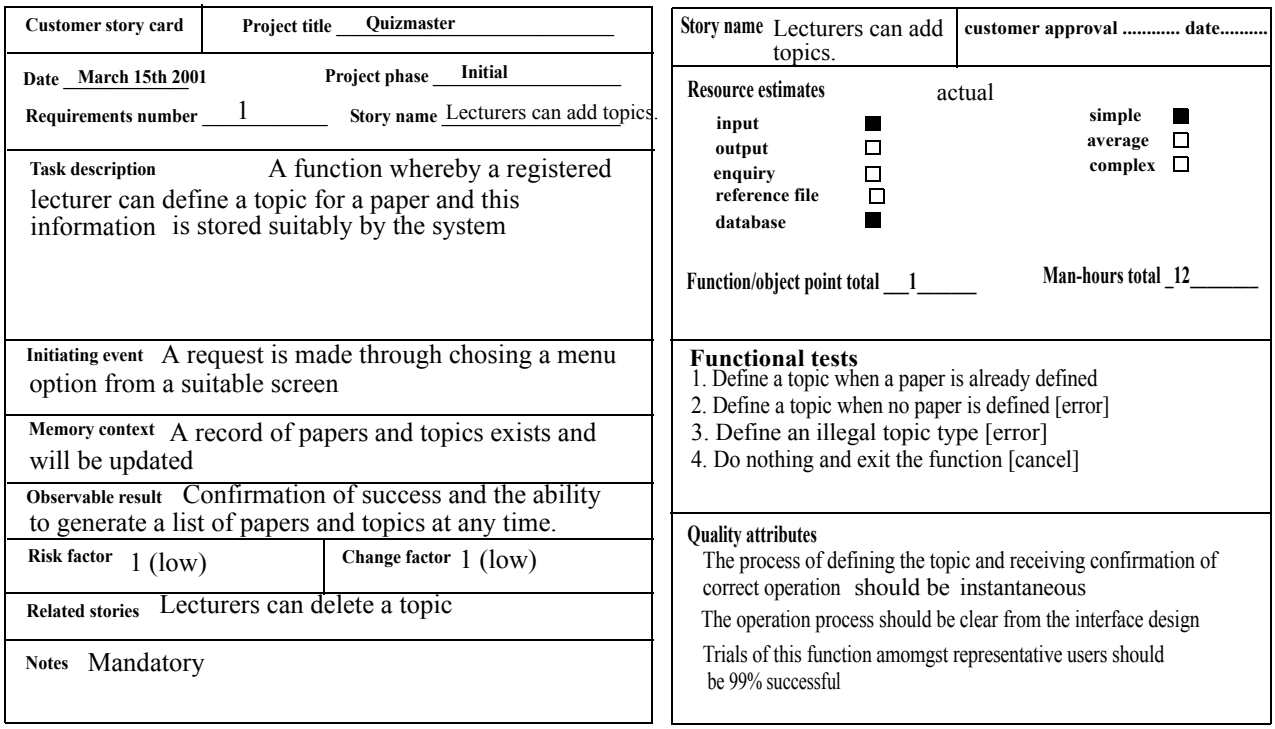


Figure 1. A story card.

The story begins with the task of requesting an option by clicking on a screen button.

The list of papers should be displayed and one chosen.

The next task is to display a suitable window with simple edit facilities to allow the user to input some simple text, namely the topic name for a paper.

The information supplied needs to be validated. In this case, is the same topic already declared on this paper? This will involve a query to the database. The functional tests defined on the card provide guidance as to the checking required. Be prepared, however, to identify other things that may need to be considered, there is no guarantee that all the special and awkward cases have been identified at this stage. Always try to think "*What if*".

If the validation fails then a warning message should be given and a repeat try of the previous task enabled.

Finally confirmation should be given to the user that the operations was successful.

These tasks can then be the basis for a series of units that will provide the functionality required. If an OO language such as Java is being used then it should be possible to define a simple set of class diagrams which will contain the main class outlines, variables, attributes and outline methods.

We will need to build a database model which will have a class to handle data access functions.

Recalling what we said about keeping the user interface, the business rules and the database as separate layers we should organise our classes to respect that principle.

Their will be a screen class to provide the initial user interface for the start of the story, which we will call the *homescreen*. The button will be provided with an adaptor/listener method to enable the story to be started. The next figure reproduces Fig. 4 from Chapter 7.

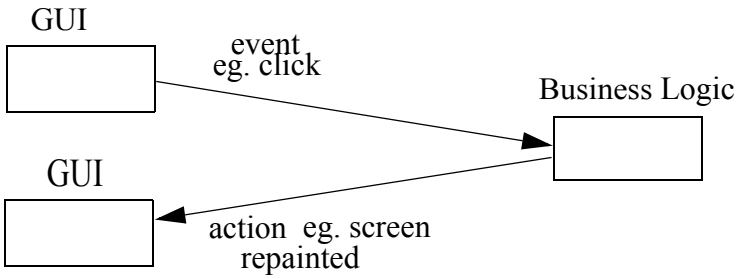


Figure 2. A simple interface action

Class *homescreen* will handle this. The button click will cause an event that calls the *addtopic*, a class that provides the basic interface for this story.

Thus a new object will be created giving a screen with edit facilities. This will include a button for data submission together with checking and recovery methods. To capture and report data entry errors an *error* class is included.

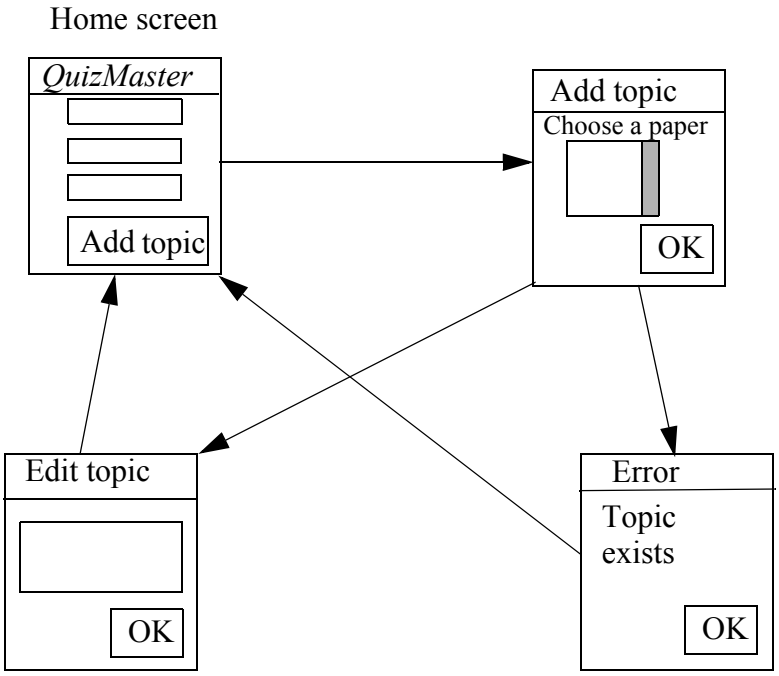


Fig. 3. Some screens and their relationships.

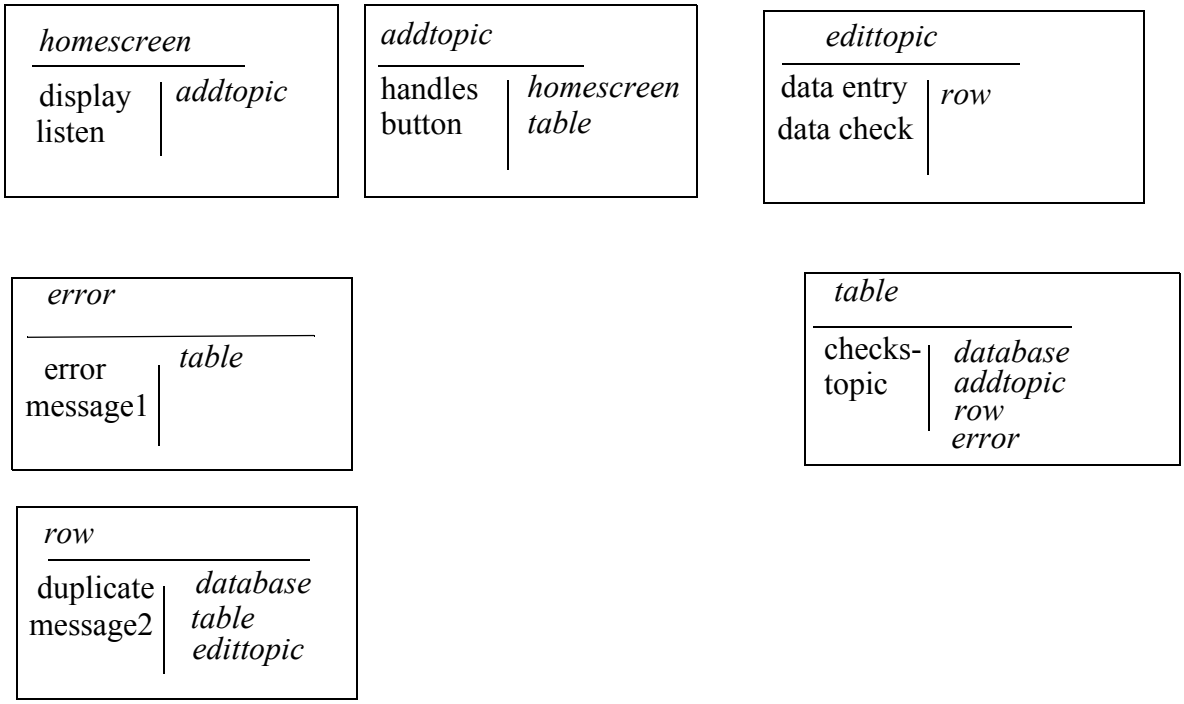


Fig. 4. Some classes involved in the story, presented as CRC cards.

The *topicscreen* class will need information from the database, so a *table* class is used. The *row* class will write to the database.

The results will be displayed in a list box using a suitable component by the *addtopic* class.

Figure 4 illustrates a possible collection of classes for this story, displayed as Class - Responsibility - Collaborator (CRC) cards. [Cunningham1986], and Figure 4 shows how the three layers are organised.

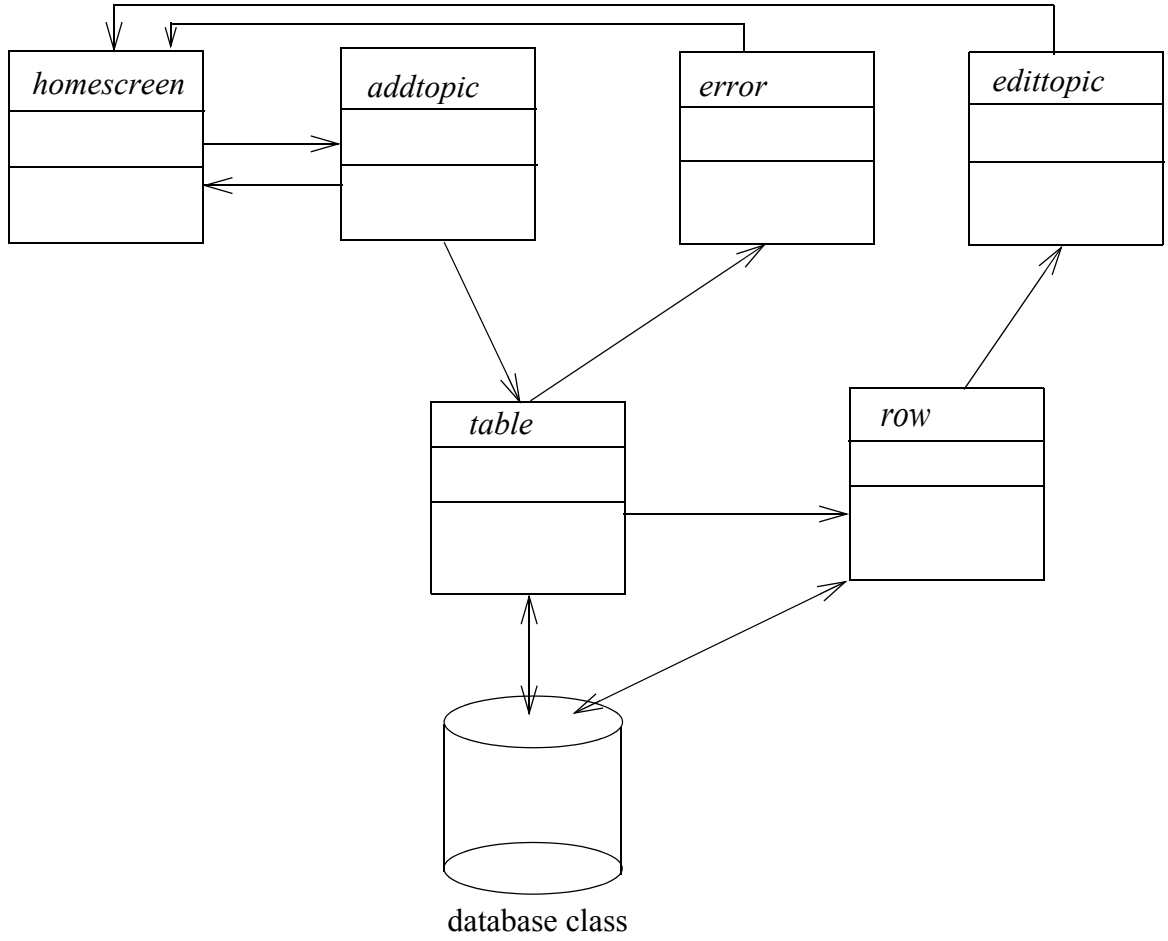


Figure 5. Task sequences, classes and their interactions.

3. Unit testing.

For each class we are building thought should be given to how the class is to be tested. As we have seen the eXtreme Programming approach suggests that test sets should be created before any coding starts. This is not as simple as it seems because at the start of the coding of a unit it may not be entirely clear how it will be written and some important tests may not be easily defined. Furthermore, many of the popular types of testing such as the white box testing techniques are based on the structure of the code. But we have no code as yet so this won't work. The lack of discussion of this point is one of the weaknesses of some treatments of XP.

What is important here is that a basic framework for testing the unit is defined and this will be developed into a more detailed set of tests in tandem with the coding. At the end of the initial exploratory coding stage a complete set of tests should then be available so that thorough testing of the class is possible.

Given the outline description or structure of a class we have to identify two important things:

a) what are the ways in which the method will be accessed and what, if any, are the preconditions on the data that is supplied to it?

b) what are the ranges of values that need to be provided for the methods?

Once we have identified these the expected outputs have to be considered and, in particular, action taken to ensure that the output information of interest can be read or displayed in an appropriate form.

We will be writing some test scripts which will be used in conjunction with the class code to establish whether it is behaving in a desired manner. These scripts, themselves forming classes or modules in the language concerned, will provide the basis for automating many of the tests but it is unlikely that all the tests can be done automatically.

The test scripts will have to provide the information needed to prepare the class for testing and this will involve identifying the entry points to the method and supplying suitable data to make the test work.

In any method that we want to test there will be some data input values needed from a defined data structure or type. It is important to ensure that the data selected for this purpose is sufficiently varied to expose the method to all possible types of failure as well as success. We are trying to do two things during testing - gain some confidence that the method works and at the same time trying to break it. Only then can we be sure that the class is trustworthy enough to be considered for integration into our existing working system.

Most values of data will be defined in the context of limits or boundaries which describe their validity so that, for example, we may have taken the decision earlier that a particular data value that is a string must be between 1 and 30 characters long and that falling outside that range will cause an error and some suitable recovery - perhaps inviting a user to try again if it is a data input through some user interface. Numerical values might also be restricted and it is useful to be proactive in this respect and not rely on the system to deal with *out of range* values.

When choosing numeric data values for using in unit testing it is useful to consider the following simple categories of data values, where we are assuming that there are upper and lower boundaries on the values:

a value below the lower boundary;
a value equal to or at the lower boundary;
a mid range value;
a value equal to or at the higher boundary;
a value above the higher boundary;
an value in an incorrect format;
a null value or no input.

If we are dealing with the type of a string of literals which must be of length between 1 and 30 then we could generate the following distinct tests:

<return>
a
abcdef
abcdefghijklmnopqrstuvwxyz1234
fkdiorufberk5486jfkjfkdlk
309475bfbldflkjslkj

abcdefghijklmnopqrstuvwxyz12346
4onkfkdpkfmlk8e3;im65687^^7E@@cmei;pd
%`¬&*
nul_input

If the algorithm used in the method needs to deal with some valid range data differently then tests with all the types of data that will exercise all the paths through the program graph of the method should be used.

If the input data to the method consists of several different types of values for different parameters in the method then all combinations must be considered. It is possible that some combinations should not be valid during the operation of the method in the software overall. It is a false economy, at this stage, to ignore these. Such combinations can cause problems when the code is integrated if there are undetected errors that cannot be found easily during integration. It will help debugging if care is taken at the unit testing stage to create tests that will report the results in a suitable way.

Suppose that a method is required to take as an input a string of literals of length between 1 and 30 together with a boolean flag which describes how the data is to be treated, true being the prompt for a message to be sent to the interface that the data is already present - perhaps a customer's details are already in the database and the false flag determines that the details should be checked for valid format and submitted to the database for insertion.

The method should then be tested using pairs of input parameters in the following form:

test number	input name (string)	boolean flag	expected result	comments
1	*<return>*	*true*	error "no proper input"	to GUI
2	*<return>*	*false*	error "no proper input"	to GUI
3	*a*	*true*	screen message "already present"	to GUI
4	*a*	*false*	"submit to database" message	needs to connect with database
5	*abcdef*	*true*	screen message "already present"	to GUI
6	*abcdef*	*false*	"submit to database" message	needs to connect with database
7	*abcdefghijklmnopqrstuvwxyz1234*	*true*	screen message "already present"	to GUI
8	*abcdefghijklmnopqrstuvwxyz1234*	*false*	"submit to database" message	needs to connect with database

test number	input name (string)	boolean flag	expected result	comments
9	*abcdefghijklmnopqrstuvwxyz12346*	*true*	error “input too long”	to GUI
n	*abcdefghijklmnopqrstuvwxyz12346*	*false*	error “input too long”	to GUI
n+1	*%`¬*&*	*true*	error “invalid input type”	to GUI
n+2	*%`¬*&*	*false*	error “invalid input type”	to GUI
n+3	*null_input*	*true*	error “null input”	to GUI/system
n+4	*null_input*	*false*	error “null input”	to GUI/system
...				

Table 1. Test planning table.

The comments column is there to provide some reminders of the possible interactions that the class might need to undertake or which need to be considered by the programmers during and after the testing of this class.

The important thing about designing test is to *think awkward* - try to invent combinations of inputs and values that are unusual as well as the obvious ones. One common adage is “if it can go wrong it will go wrong” - there is no such thing as a perfect system the best we can do is to minimise the impact of any failure or error in our code.

4. More complex units.

Not all the classes developed will fit into the simple pattern of a few independent methods that can be tested independently. More complex structures are likely and we need to identify how these might be dealt with. Luckily we can capitalise on our earlier modelling and test generation ideas.

We will consider some case studies, these are due to Jing Yuan.

Each class has its own life cycle; its operations have specified active sequences (during correct use) that must be obeyed by any user or client class. On the other hand, a class cannot control the access sequences of its clients. The operations are driven by event; it is never known when an operation will be called. In such cases, to assure the system’s correctness the programmer must use suitable error handling to deal with incorrect or unexpected use.

The active sequences of the operations could be represented in an X Machine, the input alphabet of this X-Machine includes all input parameters of *construct* and *operation* methods, the output alphabet includes all output values of the operation methods, the transitions represent

class operation methods, the memory being the data values and the output of *access* methods.

Using an X-Machine to represent the class activity can help to generate the test set more easily and completely and potentially automatically.

4.1 Case Example 1. Due to Jing Yuan [Yuan2002]

This Example illustrates how to use the X Machine to express the active sequences for a class. The CFile class is a basic file I/O class which implements the basic non buffer file *read* and *write* operation. The class members are list below:

CFile Class Members

Data Members

m_hFile	Usually contains the operating-system file handle.

Construction

CFile	CFile()	Constructs a **CFile** object
Abort	Void Abort()	Closes a file ignoring all warnings and errors.
Open	virtual BOOL Open(LPCTSTR *lpszFileName*, UINT *nOpenFlags*, CFileException* *pError* = NULL);	Safely opens a file with an error-testing option.
Close	virtual void Close( ); throw(CFileException);	Closes a file and deletes the object.

Input/Output

Read	virtual UINT Read(void* lpBuf, UINT nCount); throw(CFileException);	Reads (unbuffered) data from a file at the current file position.
Write	virtual void Write(const void* lpBuf, UINT nCount); throw(CFileException);	Writes (unbuffered) data in a file to the current file position.

Status

GetPosition	virtual DWORD GetPosition( ) const; throw(CFileException);	Retrieves the current file pointer.
GetStatus	BOOL GetStatus(CFileStatus& *rStatus*) const	Retrieves the status of this open file.
GetFileName	virtual CString GetFileName( ) const;	Retrieves the filename of the selected file.
GetFileTitle	virtual CString GetFileTitle( ) const;	Retrieves the title of the selected file.
GetFilePath	virtual CString GetFilePath( ) const;	Retrieves the full file path of the selected file.

The operation active sequence diagram is showed below:

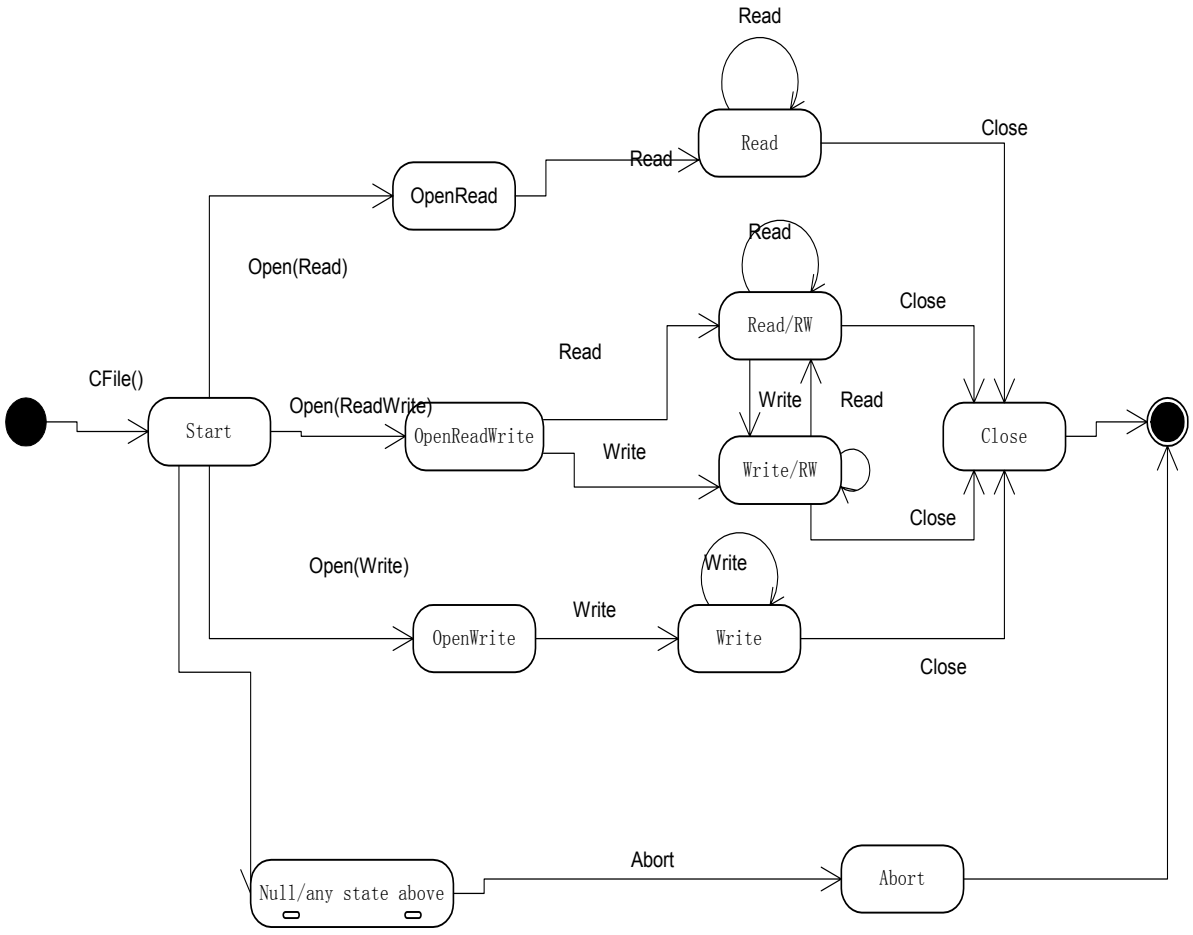


Figure 6. An X-machine model of the CFile buffer class.

Some sample test sequences are given in the following table:

CFile();Open(Read);Open(Read)
CFile();Open(Read);Open(Read/Write)
CFile();Open(Read);Open(Write)
CFile();Open(Read);Read()
CFile();Open(Read);Write()
CFile();Open(Read);Close()
CFile();Open(Read);Abort()
CFile();Open(Read/Write);Open(Read)
etc.

4.2. Case study 2. Due to Jing Yuan [Yuan2002]

In many case, it is difficult to test a class independently, we must put the collaborative classes in a group to test, or create many instances of the class to complete the test. This example shows how to represent such a case in X Machines.

Merging the two classes into one X-Machine will product too many redundant tests, the simpler way is to represent them separately and to test them one by one. But we need to test thesynchronization to establish proper collaboration of the peer class. In such a case, the two X-Machines are chopped before the synchronizing transition, and the chopped small parts are tested first, the communication points are tested after that, the whole machine is tested last.

The example is a simple web examination system. For every course in every level, there will be multiple different exam papers. The exam server will maintain the table of applicants and the table of scores in the database, and respond to the requests for papers, according to the course and level, choosing one paper from the paper library and trying to give different requesters different papers to use in the same time period, then issue the paper to the requester. When it receives the SubmitPaper message, it will check the result and calculate the score for the paper, store the exam result and send the score to the requester.

The class structure is shown below:

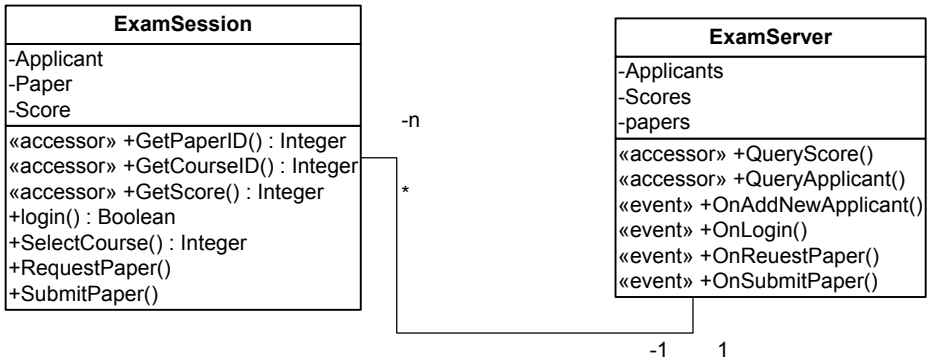


Figure 7. Class diagram.

For the ExamServer class, each method is almost independent from others, we could test

them separately. But for the ExamSession class, its methods have some active sequences and some of the methods have a synchronizing relationship with the ExamServer class. We can use an X-Machine to represent its operation methods sequence, but in order to make the test process at a synchronizing point, we need to test the synchronizing relationship first, this could be achieved by choosing the reachable sequences from initial state to the synchronizing point for the ExamSession class, but for the ExamServer class, its methods are passive, so what we need to do is to create an instance and check that the methods execute the correct results, this could be achieved by using the previous methods.

Some examples of tests from the test set for synchronizing the communication is listed below:

ExamServer; ExamSession::login; (test the method *login* as well as *OnLogin* in ExamServer)

What this means is that we create an instance of the ExamServer and of the ExamSession and then test the login method.

ExamServer; ExamSession::login;SelectCourse;RequestPaper; (test the method *RequestPaper* as well as *OnRequestPaper* in ExamServer)

ExamServer;ExamSession::login;SelectCourse;RequestPaper;AnswerPaper;SubmitPaper;
(test the method *SubmitPaper* as well as *OnSubmitPaper* in ExamServer)

After this test passed, do the test set generated by automation tools (and pass by which has tested before):

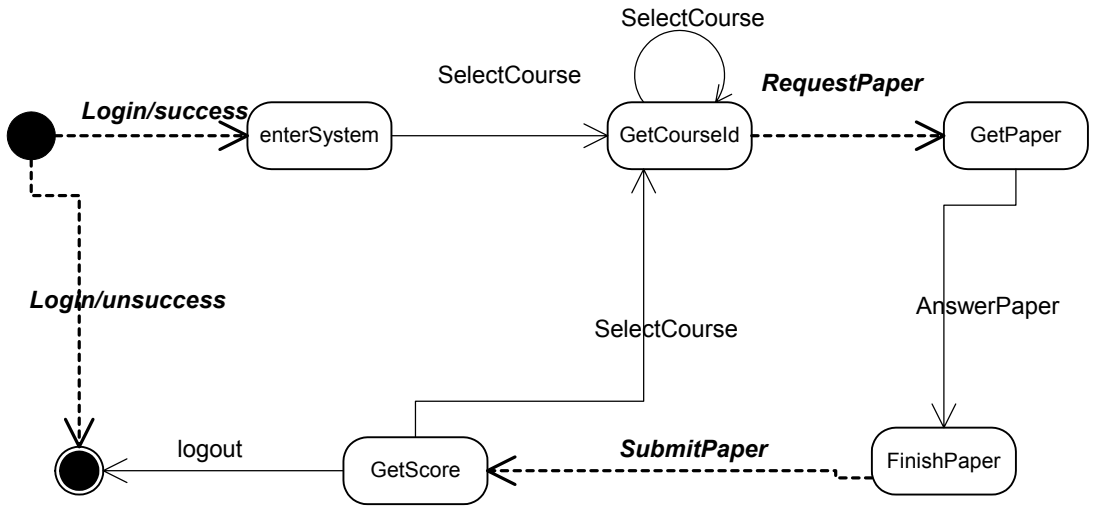


Figure 8 State diagrams.

Some more of the test set sequences are listed below:

login/success; login/success;
login/success; login/unsuccess;
...
login/success; SelectCourse; login/success;

login/success; SelectCourse; login/unsuccess;
login/success; SelectCourse; SubmitPaper;
login/success; SelectCourse; logout;
...
login/success; SelectCourse; RequestPaper; login/success;
login/success; SelectCourse; RequestPaper; login/unsuccess;
...
login/success; SelectCourse; RequestPaper; AnswerPaper; login/success;
login/success; SelectCourse; RequestPaper; AnswerPaper; login/unsuccess;
...
login/success; SelectCourse; RequestPaper; AnswerPaper ;SubmitPaper; login/success;
login/success; SelectCourse; RequestPaper; AnswerPaper ;SubmitPaper; login/unsuccess;

If we can set up a "State Representation Table" to define the states in terms of the system variables involved as follows:

State Representation Table	
State	Represent
EnterSystem	login ==True
GetCourseID	getCourseID()>0
GetPaper	getPaper !=NULL
FinishPaper	getPaper.finish==True
GetScore	getScore>=0

Table 2. A state representation table.

5. Automating unit tests.

If a fundamental part of eXtreme Programming is to test continuously and to test everything then it is important that we make sure that testing is easy - or at least that the application of well thought out test sets is easy.

There are a number of ways that this can be achieved through automating parts of the process.

The first thing to say, however, is that there is no easy solution to the problem of automating testing. Even when the test cases have been defined and the tests created there is still a lot to be done. There are many tools that are available, some commercial and very expensive, others are public domain applications that are widely used. We will mention some of the latter but it will depend on the type of project being undertaken as how useful these are.

Many projects will have a graphical user interface and the way in which information, results etc. are displayed on the screen will be important. It is difficult to completely automate the testing

of this type of application. There will need to be some human intervention even if it is just to evaluate the appearance of the screen output according to some pre-defined requirement.

Nevertheless, at the class level there are sensible things that can be done.

One approach, and a popular one amongst the XP community, is to use a tool such as JUnit (Beck, URL: www.XProgramming.com). This allows you to create a test class around the class under test and to submit tests to the code in a simple and effective way. It is highly recommended that you look at this tool.

Similar tools are freely available for other programming languages, see Appendix D for VBUnit and PHPUnit and the web site mentioned previously for other languages.

It is not always appropriate to use such a tool in some projects and in this case it is possible to write one's own scripts to automate the testing of methods.

In any case it is important to organise one's test data in a sensible way. For example, creating a number of text files with the data in some suitable pre-defined format. A comma separated value (csv) file is one sensible approach.

The test script program will then take the test file and parse it suitably, then insert the appropriate values into the method and execute it. The results will either be collected together in a suitable output file for analysis or the results passed onto some other code for display through a suitable interface component.

Depending on the application context this output file is either examined manually to see if it "looks right" or some automated checks on the output data are performed using suitably written scripts, involving evaluating, for example whether the result is equal to a pre-determined output value or not. In the latter case all of this can be done within the context of a single test program for the unit.

5.1 Writing Unit Tests in JUnit[1]

JUnit is one of a family of unit testing frameworks available. Most of the unit testing frameworks available for other languages have been written to emulate the JUnit style, although language constraints often prohibit a direct correspondence. They are often referred to as XUnit - where X represents the particular programming language.

The easiest way to structure unit tests using JUnit is as follows:

A unit test in JUnit can be taken to mean a test of a class. If there's a class called `Vector` in the package `mypackage.util`, there should be a class that tests it called `TestVector` in the package `mypackagetest.util`. For ease of reference, the directory structure of this test package should match that of the main package.

1. Notes on JUnit prepared by Dave Carrington of Genesys Solutions from material available on www.XProgramming.com

The template structure for a test class is as follows:

```
package mypackagetest;

import junit.framework.*;//the junit testing framework
import mypackage.*;

public class VectorTest extends TestCase //Must extend Test-
Case
{
    private Vector empty, full; //Just some vectors we can
test on

    public VectorTest(String name) {//Standard constructor,
cut & paste
         super(name);
    }

    public static Test suite() {//This is used later in col-
lecting tests
         return new TestSuite(VectorTest.class);//standard
structure
    }

    /**
    * In here we can set up the variables we will be using in
our tests.
    * This method is run immediately before every individual
test method.
    */
    public void setUp() {
         full = new Vector(2);
         full.add(“element1”);
         full.add(“element2”);
         empty = new Vector();
    }

    //--- Now some actual tests ---
    public testAdd() {
         assert(empty.add(“1”).size() != 0);
         //...
    }

    public testRemove() { //if Remove removes the last ele-
```

```
ment
        assert(full.remove().size() == 1);
        assert(empty.size()  ==  0);  //setup()  is  called
before each test method
        assert(empty.remove().size() == 0);
        //...
    }
}
```

Each test method is essentially just a list of assert statements, which will raise an exception if passed `false` as an argument. JUnit automatically collates all methods whose name starts with "test" to add to the set of tests, so if we were to add a method named `testInsert()` to the above code, JUnit would automatically detect it. *(Note: Other unit testing frameworks require you to explicitly name all individual test methods)*.

Once we've got some of these test classes, we'll want to be able to run them. To do this we must first create a test collection class in the following format:

```
package mypackagetest;

import junit.framework.*;

public class AllTests {
    public static Test suite() {
        TestSuite suite = new TestSuite("All of the mypack-
age tests");

        suite.addTest(VectorTest.suite());
        suite.addTest(StringTest.suite());
        suite.addTest(WhateverTest.suite());

       return suite;
    }
}
```

The above is self-explanatory – we just collate all of our test classes in a suite, and return it. We can now run the JUnit test-suite runner program to run all of the tests in our suite. This is done in the following way: either

```
java junit.swingui.TestRunner mypackagetest.AllTests
```

or

```
java junit.textui.TestRunner mypackagetest.AllTests
```

The test-runner tells us of the first assert statement that fails, giving the exception stack trace so that the assert statement can be identified.

The unit testing frameworks available for other languages work on the same principles, i.e. a function is written for each test, which contains a list of assert statements. The way they differ is in the management of these test functions – JUnit can locate individual test methods automatically using Java's reflection mechanism. Other unit testing frameworks usually don't have this luxury.

4.2 Managing Tests

A possible structure for recording is as follows:

Ref	Test	Expected Outcome
1.1	…	…
1.2	…	…
1.2.1	…	…
1.2.2	…	…
1.2.2.1	…	…

Table 3. Test results table.

The first digit in the Ref column is the story number. The second digit is a test number, and is unordered (i.e. test 1.2 can be carried out before test 1.1). All subsequent digits are test order numbers and indicate that a test x.y.z must be performed after x.y. Tests at the same level are still unordered, so x.y.a is independent of x.y.b.

So in the above, test 1.2.2 must be performed after 1.2, but is independent of 1.2.1, for example.

6. Documenting unit test results.

As we saw in Chapter 6 it is vital to maintain a reliable record of what has been done and what needs to be done. This applies as much to testing and debugging as anything. A spreadsheet style record on the tests applied and the debugging done should be a natural part of the project. These details must be available in some shared part of the group's filestore and needs to be kept up to date. The project plan will also need to reflect the progress on the testing and fixing of bugs in the unit code.

To summarise a table like the following is useful for planning the test cases:

Method name	Input	Prerequisites	Expected output	Actual Output/ Action	Status
PrintAction (constructor)	*name* of type **String** *printTitle* of type **String**	parameters given are initialised	sets up variables of the action	Constructor sets up variables correctly. Print button shows as required.	Tested Jane (12/03/ 02)
actionPer-formed	*event* of type **event**	**ActionEvent** occurs in the **JIn-ternalFrame** returned by click-ing Print button	shows a *print dia-log*		To Do
PrintTable-Action (constructor)	*name* of type **string** *printTitle* of type **string** *frameIn* of type **Component** *tatbletoPrintIn* of type **JTable**	All input types valid	input parameters stored locally	The JButton displays OK and all variables stored.	Tested and debugged Bill (12/03/02)
actionPer-formed	*evt* of type **Action-Event**	**ActionEvent** occurs in the **JInternalFrame** returned by click-ing the *Print* but-ton	The printer job is created and a print dialog hows. If *OK* then table is printed, if *cancel* then no print.		To Do

Table 4. Unit: PrintAction (extends AbstractAction)

As the project continues it may be necessary to revisit some of these units and their test details. Perhaps there has been a slight change in the requirements, for example, the details of the data to be entered into a method might have been changed subtly in order to achieve some other objective. This may then require the re-testing of the unit under slightly different parameter values. These test descriptions will allow you to keep track of these changes. The tables of test cases and results must be updated to reflect the new requirements.

7. Review.

The identification of the classes and their implementation will form a major part of the project. If we proceed in the true XP way we will have written the unit tests first and run these against our code on a very regular basis until we have convinced ourselves that they work as required. this isn't so easy as it sounds.

A principled approach to building unit test sets has been described but this is still an active area of research. The approach is rather different to the structural or white box technique often used in traditional software development. We have seen that it is not appropriate here

since we have to write the tests *before* we write the code. One technique often iused in traditional white box testing is to estimate something called *test coverage*. This is a figure that describes how much of the code has been exercised by the tests, it might be defined in terms of what percentage of decision points the test set has exercised, the percentage of branches traversed during testing etc. Sadly, such measures do not tell us much about how well the testing has been done. It merely measures the amount of effort that has been applied to testing. The testing techniques described here can provide complete fault detection if the basic assumptions and design for test conditions are satisfied.

Exercises.
1. Set up the JUnit system and try it out for a simple program. If you are not using Java use an appropriate alternative to JUnit. Think up a few test values to apply to the program.
2. Build a simple model of the program written in question 2. Now generate some tests from the model and compare with your earlier list.
3. Apply the techniques to a more complex model with communication and synchronisation features.

Conundrum.

About 20 years ago the UK Government purchased a system to deal with Air Traffic Control over London. It was based on a number of similar systems that had recently been installed in the USA. Unlike the American systems there were serious problems with the London system. Planes flying over London would suddenly disappear from the screen. Equally alarmingly, planes would suddenly appear as if from nowhere. Extensive testing was carried out, especially on the component of the system that was fed the radar information and dealt with the display of the positions of the planes on the screen. No defects were found, everything was exactly as the requirements demanded.

Why did the system work in the USA but not in London?

References.

[Beck1999], Kent Beck, "*Extreme Programming Explained*", Addison-Wesley, 1999.
[Cunningham1986], W. Cunningham & K. Beck, "A diagram for Object-oriented Programs", *Proceedings OOPSLA-86.*
[Fowler2000], M. Fowler, "*Refactoring - Improving the design of existing code*", Addison Wesley, 2000.
[Yuan2002], J. Yuan, M. Holcombe & M. Gheorghe, "Where do unit tests come from?", *Submitted to XP2003*.

Web sites.
<http://www.XProgramming.com>
<http://www.junit.org>

Chapter 9.

Evolving the system.

Summary: Dealing with change. Changing requirements. Changing test sets. Changing code. Refactoring the requirements, the tests and the code, working with the client, integrating the releases.

1. Requirements change.

Change to the requirements is bound to occur and the way that we deal with it will be a vital aspect of a successful project.

There are a number of different manifestations of requirements change, some are more serious than others. We have been trying to identify those areas of the system that might be subject to change from an early stage in our requirements capture and analysis. Hopefully, we will not be too far wrong but you can never tell.

We will consider several types of change and how to deal with them. Some are serious and will involve us in redoing a lot of our previous work, some are more easily dealt with and won't affect the project outcome too much. There is always a price to pay if the change is significant and the client should realise this. The XP approach is to be agile and adaptable as well as to be honest with the client and to talk to him/her frequently. That way we may see the changes coming and prepare for them. We should also explain to the client the costs of the changes, the delays that may occur, the reduction in quality if it isn't thought through properly and so on. Ask the question: do you really need this change? If the change is fundamental to the way the client's business or organisation is evolving then we need to embrace it with enthusiasm.

Changes can occur during the discussions about the system, about the business processes and during demonstrations of software, whether delivered or not. One benefit of an incremental delivery - it might not actually be delivered just demonstrated to the client depending on the context of the relationship between the customers and the developers - is that it gives customers a chance to see what they may be getting and to identify any changes that they may like, including new features. Also, if their business needs have changed they can then discuss the impact that this may have on your project, in terms of priorities, functionality etc. Not all customers will want to install an increment of the system in their premises. This can be a cause of problems if their computer system differs significantly from that of the development team.

They may also comment on the *look and feel*, on the GUI and how acceptable that is to their organisation and its workers. This may result in a significant change to the user interface, to the presentation, in particular. If your metaphor is one where the business logic and the presentation layer are reasonably separate then this may be something that you can deal with. If the change is due to a significant alteration to the underlying database then this can also be managed but is likely to have knock on effects throughout the system because so much of it may need to relate to the database structure..

2. Changes to basic business model and functionality.

These sorts of changes can occur at almost any stage of the project. Sometimes they are the result of the team not understanding the client's requirements or business processes at the time and are thus a correction of what was originally thought. These changes need to be related to the current state of the project. If the project has developed a requirements document then the changes may require the introduction or substitution of new requirements statements and these should be expanded into stories. Then the changes have to tracked through the development of the project so that, for example, implications for the underlying database, if one is involved, are considered. When we revisit the integration of the stories into an X-machine model with its accompanying user interfaces and input and output requirements.

The test sets will need to be redefined properly at the systems level. The classes associated with these changes must be identified and the unit tests updated to reflect the new requirements, the code needs to be re-programmed and tested in the usual way.

It is not always easy to achieve these changes without extra work. If the stories have not been implemented yet then things are much easier. As always the later a change is identified the more expensive it can be. It seems that XP and other agile methodologies are more able to cope with change than others. If your project is *fixed scope/fixed price* then there will be times when significant change will not be compatible with the fixed end date for project completion.

Whatever the situation it is vital that all the new parts of the system are properly documented, so that we have to maintain the key information about the system:

the stories;
the models;
the system tests;
the system metaphor or architecture;
the classes and methods;
the unit tests;
the code;
the user manual and maintenance manual.

This updating may be accompanied by comments to indicate what has been done.

Finally the version numbers of the various artefacts listed above must be updated. *Version control* is discussed further in Chapter 10.

3. Dealing with change - refining stories.

Now the requirements may change, this is the point of using XP, so how do we deal with this?

Suppose that the client is happy with the initial set of stories and the requirements document. He/she selects some critical stories to implement first. The team begins to think about how to do this, see Chapter 7.

The client then comes back to tell us that there needs to be a change. To decide how to deal with this and what it means for the system testing we need to consider what sort of change it is.

3.1) Changes to the underlying data model.
Suppose that the new requirement is to have more information about the customers, perhaps an indication of their credit worthiness or whether they qualify for some discount. This involves

changing the internal memory of the model, we can do this quite easily and then introduce a new area of the interface to provide the extra functionality.

So we now have:

customer_details : name, address, postcode, phone, fax, email, discount

where discount is either yes or no.

The screen is now changed to allow a discount yes/no button to be chosen or a discount flag to be checked.
This will impact on our test sets by requiring the discount data to be present in the tests. We need to identify all these changes on story cards so that they are properly documented and we are in a position to know what to do.

3.2) Changes to the structure of the interface, perhaps the introduction of a new screen.
This will mean altering the state machine model by introducing a new state, for example. To access this state a new transition together with an accessing function will have to be defined. The activities within this screen and the exiting from it will also lead to new transitions and functions that need to be defined. Each of these will identify a new story which, in turn, will define a further requirement.

3.3. Adding a new function.
Here we are inserting a new transition with its corresponding function into a diagram. Here a similar strategy applies, for every test sequence that gets to the state where this new function originates we develop a new test sequence that triggers the new function. We then complete each of these new test sequences by creating paths through the machine following on from this function. Again, this will lead us to new test sequences.

3.4. Changing the functionality of a function.
The basic strategy will not be affected here unless there are issues with the preconditions for the function. If the precondition for the operation of the new function is different from that of the replaced function then the test sets may have to be changed in a more subtle way. In other words we may need to test for the non-operation of the function by choosing previous data values and memory values so that the function does not operate. This can only be dealt with on a case by case basis.

4. Changing the model.

The changes are captured using revised story cards and now we need to integrate them into our model so that we can see the effect that they have on the system and how they impact the test sets.

The X-machine model is built from the user stories in order to provide a basis for functional system testing. The changes will involve a number of different transformations of the model which can be considered separately.

4.1. Changing a process.
Suppose that we have the following model:

and the process enter(order) is changed in some way, perhaps the information being input is dif-

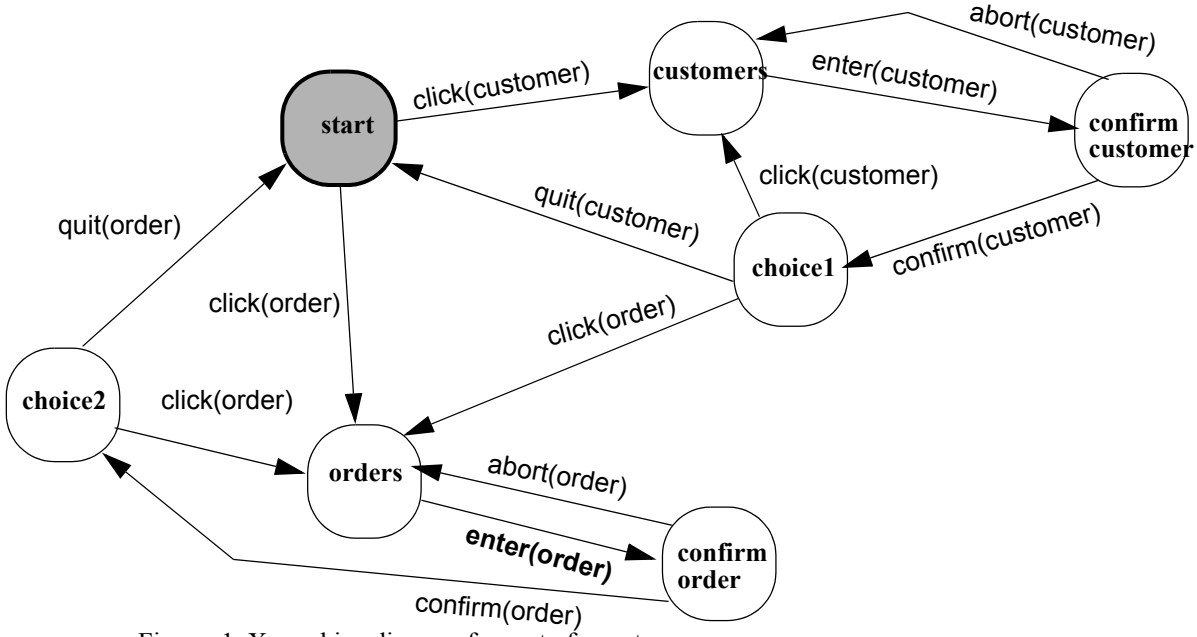


Figure 1. X-machine diagram for part of a system.

ferent, then this needs to be reflected in two ways, the definition of the function is different and so the interface that provides the user with the capability to input the information will need to be changed. The database will also probably have to change to accommodate the new data being input. Suppose that we need to collect more information about the order, for example the customer's tax number (T-no.)

We should make it clear that this process has changed in the model by amending the diagram at the top level by changing the label to enter(order)', as well as altering the lower level diagram where the process is expanded into more atomic processes.

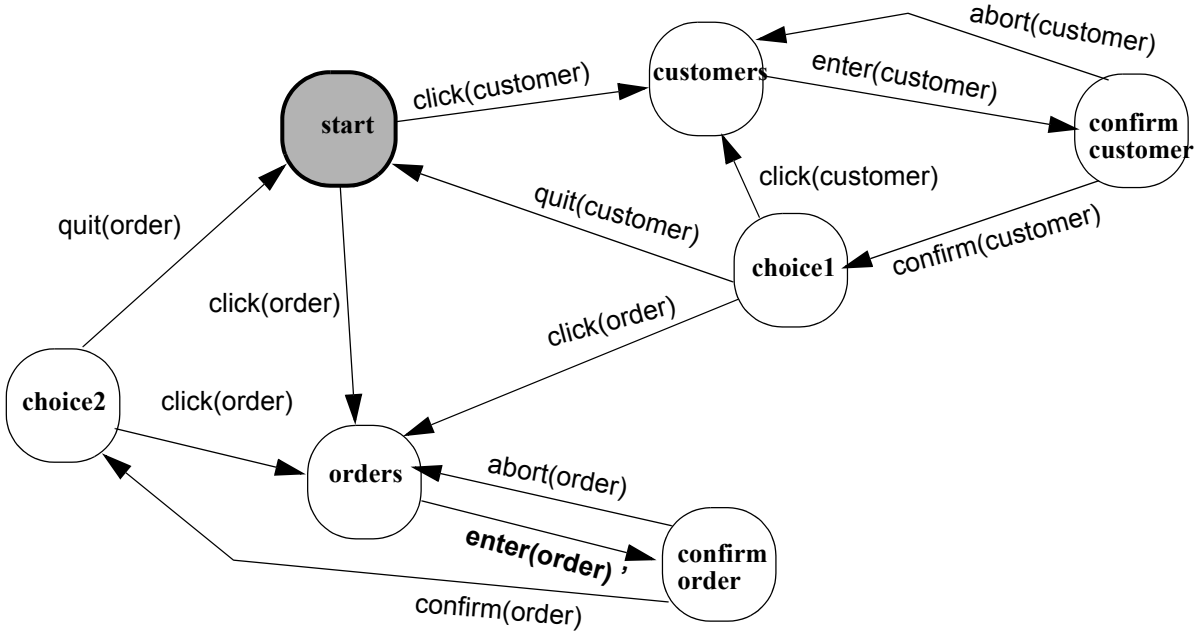
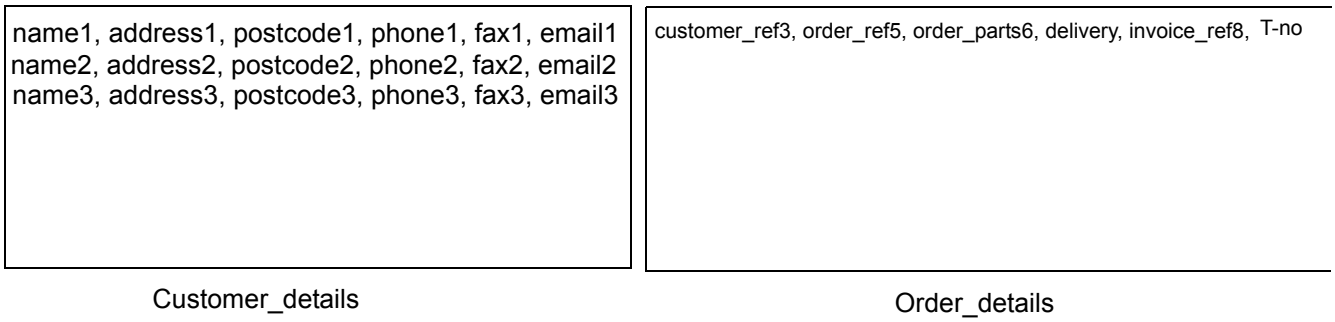


Figure 2. Amended X-machine diagram with operation enter(order) ' replacing enter(order)

The order details diagram also needs to be changed, as below.



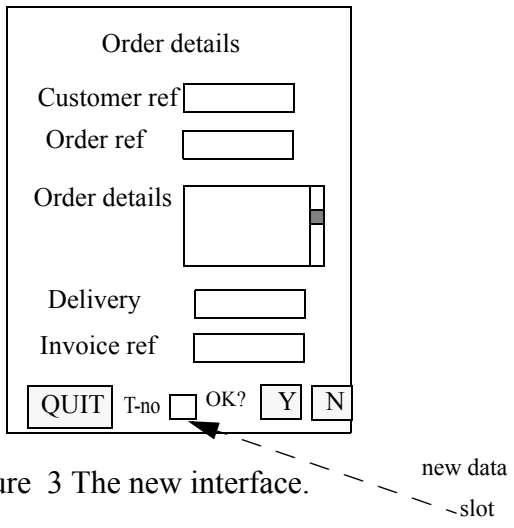


Figure 3 The new interface.

4.2 Removing states.

Here we consider the issues related to removing a state, perhaps the client does not want a particular feature any longer.

If we remove a state then we must remove all the processes that lead to that state and which all those that leave that state. This may interfere with the flow of business processes and it is vital that we check this thoroughly before committing ourselves to the new model of the requirements.

All those processes (or transitions) can now be removed from our architecture and the user interfaces associated with them also. It is vital that we then revisit the system tests to see what the implications of this are. Since much of the diagram is unchanged all the tests that involve sequences that visit this state can be removed also. Take care over this.

It may be necessary to introduce a new process or two to link states before and after the removed state in order to make the whole system work. We look at this next.

4.3 Adding states.
When introducing a new state it will also require the introduction of new processes and their transitions. In fact there is no point in introducing a new state unless there is a process that needs to be dealt with separately. This might be because we decide that a particular case in the business process needs to be dealt with in a separate way and this might mean designing a new user screen specially for this event. This is often better than trying to cover all the possibilities in one screen. The client will have a view on this. So we might then break a process down into several processes with their own states.

The original function which is represented as f or g is split into two separate functions f and g.

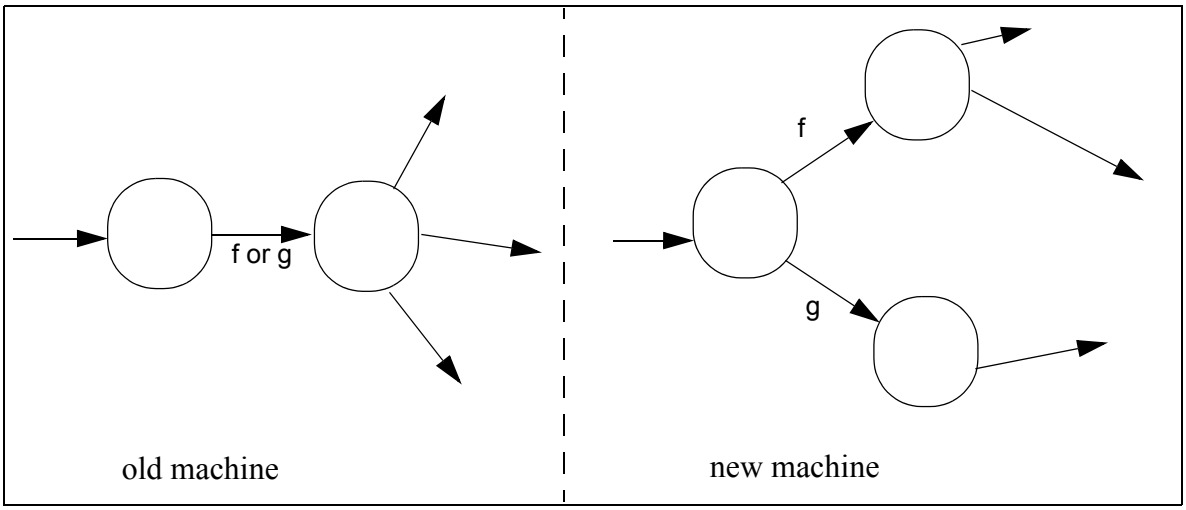


Figure 4 Splitting a function from a state

As before we must update the story cards with the new functions and generate new tests to cover the changing model structure.

The interfaces will also have to be changed and this is important to do at the same time, lest we forget.

The methods implementing the functions will now need different tests and this is another thing that must be attended to.

4.4. Adding processes.
The final type of change is where we introduce a new process which will operate between two existing states. Here we need to consider, carefully, the way that this process will be triggered form the state, we could easily get into a non-deterministic situation if there is any mistake here. The input to the new process together with the expected memory condition at that moment must not overlap with the input and memory conditions for any other process that might already be there.

Thus if we have a process of the form:

```
process: input, memory
```

which is valid for a set of pairs of values from the sets `input` and `memory` and another process of the form:

```
process': input', memory'
```

which is valid for a set of pairs of values from the different sets `input`' and `memory`' then we need to make sure that there are no values which satisfy the condition that they belong to both at the same time.

In mathematics, if the domain of `process` is the set I x M and the domain of `process`' is the set I' x M' then we need that the following holds:

$$(I \times M) \cap (I' \times M') = \varnothing$$

There are several ways in which this can be assured.

1) Choosing I and I' to be completely different with no values in common.
What this means is that if, for example I is the set of inputs corresponding to all strings of characters of length less than 30, say (perhaps these are names of customers) and I' is the set of all strings of numbers of length less than 10 (maybe these are customer reference numbers) then we can be sure that the two processes do not interfere with each other, in other words when we try to trigger either of the functions then only one can work.

2) Choosing M and M' to be completely different with no shared values.

3) If I and I' share values or M and M' share values then we must make sure that there are no pairs which are the same.

For example if I and I' were both all strings of characters of length less than 30 and M and M' were boolean (True of False) then `process` can only work when M is T and process' can only work when M' is F. At any moment when the computation reaches the start state of the two processes either the memory value in M is T in which case `process` works or the value of M is F and thus `process`' operates.

If these conditions are not valid then we must adapt I' or M' to ensure that they are, otherwise we are in danger of designing a non-deterministic machine which could behave unpredictably.

The new function will be a method in some suitable class. It is possible that the class that implements the method corresponding to the new `process`' is the same class that implements

`process`. In this case we extend the unit tests to include the `process`' method. Otherwise we create completely new unit tests for the new class.

5. Testing for changed requirements.

The system testing will now have to take these changes into account. We will either redo the complete system test set - not a good idea because of the work involved, or test the new part of the machine separately and then runs a smaller number of integrating tests. These could be developed in the following way.

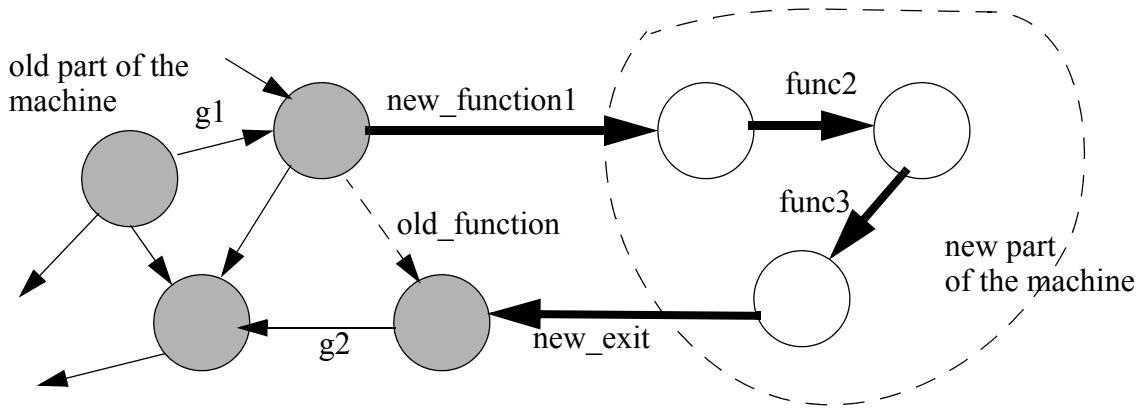


Figure 5. Adding new states.

In every test of the form:

```
...g1 ; old_function ; g2...
```

we create a new test where the function, old_function1 is replaced by a test from the new machine such as:

```
.. ; .g1 ; new_function1 ; func2 ; func3 ; g2 ; ...
```

This will lead to a number of new tests. We do all the old tests as well.

The system tests also need updating. Since we have extended the state diagram with a new transition we need to write tests that will trigger this transition at all possible occasions. This means both legitimate paths through the updated machine but also we need to try to trigger `process`' from every other state to ensure that we have not inadvertently introduced additional, unintended, functionality into the system.

In practice we look at the two models, the original one and the new, revised one and focus on those test sequences that involve the changed parts of the machines. In this way we can preserve many of our original tests and know that we will not have to retest them. However, if there is any doubt as to whether you should retest then you should retest!

6. Refactoring the code.

As the system changes and evolves the version of the code that represents the current state of the system will undergo a number of changes. Some of the changes will be simply as a result of building increments into the existing code, some will be as result of fixing problems and bugs. Refactoring is the activity of rewriting the code without changing its functionality in order to achieve some specific aim, such as to make the code more understandable for maintenance, to make it compliant with some standards or to reorganize it in line with changes to the platform or system architecture.

In this section we will briefly describe some common refactoring techniques, the principle source book for this topic is Fowler [Fowler2000] and this should be consulted for the details.

Suppose that you have developed some classes during the early stages of the project and as you increase the functionality of the system you develop a new class which uses a method that is already in an existing class. The temptation is to *copy and paste* this method into the new class. This now means that the same method appears in at least two places. It might happen, during the course of the system evolution or during maintenance that the method needs changing. The problem now is that you have to remember to change it at all of its locations and you may forget to do this. The maintenance engineering may not even be aware of all the places that this method belongs. The better way is to create a separate class embodying this method and to refer to this class within the body of the classes that previously used it. This process is called the *extract class refactoring*.

Some methods are very large and thus likely to have behaviour that is hard to understand. Rather than spend a lot of time trying to write comments to explain what is going on it might be better to refactor the method by splitting it up into a collection of simpler methods which are organised in a clearer way. Here the *extract method refactoring* can be used. take care with variables, however.

Some classes seem to grow out of control and this might also need dealing with using something like an *extract class refactoring* which tries to group related variables in a sensible way and perhaps introduce components and subclasses to deal with the complexity of the original class.

Data can be reorganized in more natural ways we have already seen the architectures that separate data from business logic and this is another principle that can be applied in the code. Data values can also be replaced by specific objects which provides a neater structure. Awkward arrays can also be turned into objects.

Conditional expressions can be simplified by extracting the conditional into a method and then deal with the then and else parts separately.

Remove confusing flags by using break or continue statements.

Since refactoring should not alter the functionality of the code it is possible to use the test sets to check this. Getting into the habit of continually testing everything as you do it will give greater confidence that you have not broken anything during the refactoring process. This does assume, of course, that your tests are good one.

There are many more things that can be done and Fowler's book is an important source of ideas and inspiration.

An issue ws raised in Chapter 2 concerning the unit tests and what happens to them when the code is refactored. Ideally, there shouild be a unit test associated with each class. For some types of refactoring the overall class structure will not change and so the tests will be still be associated with the right classes. The test sets may need some maintenance, however, especially if there have been changes to variable names etc.

7. Summary.

Coping with changes is something that you will certainly have to do. It makes a big difference if you approach the problem in a systematic and practical way. Panicking is bound to lead to further problems. Stay cool and think through the changes carefully and logically, what they are, how they impact the rest of the system and how they can be managed. The evolution of the test sets is a vital part of the process. XP is totally dependent on the tests for ensuring the quality of the solution, neglecting these will lead to a poor quality system. refactoring the code, making it more understandable and more consistent is an activity that will be on-going as the system evolves and changes. It is important that some effort is made to do this - remember the people coming along behind you who may have to maintain your system. Most university programming exercises do not have this dimension since, in most cases, no-one is going to use the software in earnest.

Exercises.
1. Review the way that you dealt with change. Were you able to revise your test sets to account for the changes in an efficient manner.

2. Did you refactor? What refactoring methods did you use? Could you all agree on what needed to be refactored and why?

Conundrum.
The customer was the IT director of the company which produced and sold biological specimens to research laboratories and pharmaceutical companies. The software was to support the entire company activities, which involved the production process - which had to be fully documented to meet government regulatory procedures - the stock contol process, the ordering process and the invoicing and accounting process.

After working extensively with the customer and delivering a number of incremental versions the customer was satisfied. A final delivery was made and at this point the customer invited several personnel from the company to attend the demonstrations. These potential users of the system pointed out many problems with the business concepts upon which the system was based. It became clear that the customer did not understand his company's business process. It was also clear that we will have to re-engineer the system. The new requirements were significantly different, there were few areas where the detail was the same although the overall architecture would be very similar.

Should we start again from scratch or try to adapt what we had already done, reusing and preserving what we could?

References.

[FOWLER2000], M. Fowler, *"Refactoring - improving the design of existing code",* Addison-Wesley, 2000.

Chapter 10.

Documenting the system.

Summary: The purpose of documentation. Providing maintenance information in the code, coding standards. User manuals, on-line help.

1. What is documentation for and who is going to use it?

One thing that good software engineers and programmers are good at is creating lots of documentation. Poor programmers produce relatively little. However, we should not judge people or organisations by the *amount* of documents generated, *quality* and *relevance* are much more important.

Quality can be defined in many ways but for our purposes it must mean that the document is fit for its purpose. Thus we need to identify *what it is to be used for*, *who is to use it*, *what they are trying to do* and to judge the quality of the document on the basis of how it helps them achieve their objectives in the best possible way.

This brings us to our first difficult problem, people are individuals and whereas a particular document is ideal for one person to use to achieve their task it may be unsuitable for someone else with a different background and experience carrying out the same task under different circumstances.

Some people like to have the documentation available on-line and others prefer a book form. This is something that should be confirmed with the client particularly in regard to the user manual for the system.

We will consider some of the issues relating to the preparation of documentation and its implementation either paper based or electronic.

We will look at different types of document for different uses, documents for programmers and system maintenance, documents for users and documents for managers.

2. Coding standards and documents for programmers.

A vital aspect of XP is that the code is understandable and that those reading it can be in a position to change it, up date it or develop it further as easily as possible. In this section we will look at coding standards and how the source code should be presented.

The purpose of coding standards is to ensure that all the programmers in a company produce source code to the same standard in terms of how it is structured and presented. Not all software houses will share the same style and standards but the key point is to get used to working within the constraint of a formalised standards regime. This will be good experience for future careers.

In this chapter we will present and discuss the standards used in the Genesys Solutions company, a software house run by 4th year students at the University of Sheffield, UK. It is an example of a set of standards that works but is not too burdensome.

XP relies on the clarity and understandability of the code and this means that we need to take a lot of care over how we write and document this. Remember one day someone may need to maintain your system, what is now obvious to you now may not be obvious to them - or to you in a few months time!

Maintenance is a vitally important aspect of software engineering. Maintenance can take on many forms from bug fixing (perfective maintenance) to extending or changing the functionality of the system in some way. It is vital, therefore, that the programmers doing the maintenance understand fully what the system does and how it is built. They will not have access to a lot of design information - we have agreed that this is often unreliable and out of date, especially if there have been requirements changes which have not been properly reflected in the designs.

As we mentioned earlier, it is vital, that the code is presented in a readable and understandable form. We have emphasised the need to keep things simple and to organise the code in a maintainable fashion. Some of these issues will be discussed in the following chapter. Here, we concentrate on the basics of code documentation and on coding standards.

The language Java has a major facility that will help here, namely the Javadoc system. If Java is being used for the project then Javadoc should be a mandatory part of the development method. It provides detailed information about the structure and coupling of the program - at the end of the project a Javadoc print out (file) should be made available.

We focus, here, on the issue of coding standards since it is important to address these from the start.

The purpose of coding standards is to establish a common structure and content of object-oriented (or any type of code) that is being developed by a team of programmers. In an eXtreme Programming context it is vital that everyone abides by the same coding conventions since the task of coding is distributed amongst the team. Although pair programming can provide some consistency in coding style it is not enough by itself.

One problem that can occur is that the standards are perceived to be very time consuming and bureaucratic to adhere to. This attitude should be resisted, especially if the grading of the project provides an element related to how well the team adhered to the standards.

We will look at some standards developed for Java within the context of student projects for external clients and comment on why they are the way they are. Standards for other languages can be found, several groups have web sites with proposals for standards and these should be consulted where necessary.

3. Coding standards for Java.[1]

GE NE S Y S S OL U T I ONS Genesys Coding Standard for Java

This Coding Standard has been adapted from the web page entitled *Code Conventions for the Java™*

1. These standards were developed by the 4th year students in the student software house, Genesys Solutions and are used by the 2nd year students in their real projects.

Programming Language, which can be found at http://java.sun.com/docs/codeconv.html.

3.1.1 README

Any directory containing Java Source Files (`.java`) should also contain a file entitled README. This file should summarize the contents of the directory in which it resides. The summary should be a brief description of what the overall purpose of the file is, and not technical details of individual methods, variables or other implementation dependent factors.

3.1.2 Java Source Files (.java)

Each Java Source File should contain a single public class or interface. If private classes or interfaces are associated with a public class, they may be located in the same file. In this situation, the public class must be the first class in the file.

A Java Source File has the following ordering:

3.1.2.1 Beginning Comments

After `package` and `import` statements, and before the main class definition, there must be a block comment of the following format:

```
/*
* Name: The name of the Class
* Author(s): All who contributed to this Class
* Date: Date the Class was last altered
* Version Number: The version number of this update.
* Using the standard major/minor revision system
* Starting from 1.0
* Description: What the Class does. If it becomes too long
* consider breaking it down into smaller
* components.
* Changes History: List of changes, referenced by version number
* outlining changes from previous version
* i.e.
* 1.1 fixed bug that caused program to
* crash.
* 1.2 added the blah functionality.
*/
```

3.1.2.2 Class and Interface Declarations

The following order should be maintained within Class and Interface declarations:

· Class (`static`) variables:

These should also be sorted into the order `public`, `protected`, package (no access modifier) and finally `private`.

Instance variables:

These should be sorted in the same order as for class variables.

² Constructors.

² Methods:

These should be grouped by functionality and not by the access modifier that they possess. Each method must have a block comment of the following format:

```
/*
* Name: The method name.
* Author(s): The name of all authors who have contributed to
* this method. Only include if there is more than
* one author for this Class.
```

```
* Description: Brief description of what the Method does. If it
* is too long, consider decomposition.
* Parameters: List of input parameters.
* Output: The relevance of the returned value. Only
* include if the return type is not void.
*/
```

Four spaces should be used as the unit of indentation. This avoids excessive horizontal spread across the screen in deep sections of source code.

Comments should not be enclosed in large boxes drawn with asterisks or any other characters. Also, consider that many people believe that frequency of comments sometimes reflects poor quality of code. If you are about to add a comment, take a moment to see if you can rewrite the code to make it clearer.

3.3.1 Block Comments

These should be indented to the same level as the code it is referring to and preceded by a single blank line. To aid setting it apart from the actual code, each new line in a block comment should start with an asterisk as shown in the example below:

```
/*
* This is a block comment.
* Each new line, like this one, starts with an asterisk.
*/
```

3.3.2 Single Line Comments

These should also be indented to the same level as the code it is referring to and be preceded by a single blank line. If the comment can not be written on a single line then the block comment style should be used.

```
if (condition)
{
// This is a single line comment.
```

3.3.3 Trailing Comments

These can be located on the same line as the code they are describing. However, they must be short and should be shifted to the far right. If there are multiple trailing comments in a given method, they should be aligned with one another. The use of this // comment delimiter to comment out chunks of code is preferred over a block comment style because of the ease of uncommenting individual lines at a later date:

```
if (foo > 1)
{
// int i = 0;
// i++;
// foo = i;
return TRUE; // explain why here
}
```

3.3.4 Comment Format

To allow for easy determination of who has altered pieces of code, and to ascertain when the changes were made, the following format should be adopted for all comments:

```
// XYZ – The following will do something new – DD/MM/YY
...
/*
* XYZ – DD/MM/YY
* This needed some extra explanation...
*/
```

Where XYZ are the initials of the programmer who has added the comment and DD/MM/YY is the current date/month/year.

3.4.1 Number Per Line

There should be no more than one declaration per line since this encourages the use of trailing comments to describe the purpose of the variable. It is also recommended to indent the names of variables in a block of declarations to the same level, to enhance the readability of the code:

```
int percentageComplete; // how much the project is complete
int daysRunning; // how many days the project has run
int i, j; // AVOID!
Object currentProject; // the current project
```

3.4.2 Initialisation

Where possible all variables should be initialised upon declaration. The only time that this can not be done is when some computation is required before the initial value of the variable is known.

3.4.3 Placement

Declarations should only appear at the start of a block (or clause) of code (This meaning a group of statements surrounded by `{` and `}`). You should not wait until their first use to declare a variable. However, for one-time 'throw away' variables in a for loop, they may be declared as part of the statement:

```
public void aMethod()
{
int int1 = 0;
...
if (condition)
{
int int2 = 0;
...
}
for (int i = 0; i < int1; i++)
{
...
}
}
```

If variable `foo` is still in scope, new variables should not be named using this same name, which would hide the declaration of `foo` at the higher level.

3.4.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be adhered to:

- No space should be left between a method name and its opening parenthesis (which starts its parameter list,
- The open brace { should be located on the next line down at the same level of indent as the method or class name,
- The closing brace } should start a new line on its own and be indented to the same level as the open brace }. This ensures that paired braces are at the same indent level and are easy to spot,
- Methods should be separated by a single blank line.

3.5.1 Simple Statements

Each line should contain at most one statement:

```
argc++; // OK
argc++; argv++ // AVOID!
```

3.5.2 return Statements

A return statement that includes a value should only use parentheses if this aids the clarity of the statement:

```
return;
return anObject.aMethod();
return (size? size : defaultSize); // adds clarity!
```

3.5.3 if, if-else, if else-if else Statements

The following format should be adopted for these statements:

```
if (condition)
{
statements;
}
if (condition)
{
statements;
}
else
{
statements;
}
if (condition)
{
statements;
}
else if (condition)
{
statements;
}
else
{
statements;
}
```

Note that in the situation where `statements` is in fact a single `statement`, the following is permitted:

```
if (condition)
statement;
```

3.5.4 for Statements

The `for` statement should be formatted like this:

```
for (init; condition; update)
{
statements;
}
```

3.5.5 while and do-while Statements

The `while` statement should have the following form:

```
while (condition)
{
```

```
statements;
}
```

Similarly, the `do-while` statement should look like this:

```
do
{
statements;
} while (condition);
```

3.5.6 switch Statements

A `switch` statement should have the form shown below. Notice that each time a case falls through (i.e. there is no break command) there should be a single line comment to warn of this. This helps prevent simple errors upon later re-visiting the code. Every `switch` statement must have a `default` case.

```
switch (condition)
{
case ABC:
statements;
// falls through!
case DEF:
statements;
break;
case XYZ:
statements;
break;
default:
statements;
}
```

3.5.7 try-catch Statements

A `try-catch` statement is shown below. Notice that it is not essential to provide a `finally` clause.

```
try
{
statements;
}
catch (ExceptionCase e)
{
statements;
}
```

3.6.1 Blank Spaces

Blank spaces should be used in the following circumstances:

- A blank space should appear after commas in argument lists.
- A binary operator should be separated from its operands with a blank space. A unary operator should not be separated from its operand.
- A blank space should appear after the semi-colons in the for loops expressions.
- Casts should be followed by a blank space.

3.2 Blank Lines

Blank lines should be used in the following circumstances:

- Between methods.
- Between the local variables in a method and its first statement.
- Before a block or single line comment.

[2] Between logical sections within a method that will increase readability.

The following table outlines the conventions that should be used when naming an identifier. These are essential for readability and quickly determining what the function of an identifier is.

Identifier type	Conventions	Examples

Class	Should be nouns in mixed case with the first letter of each internal word being a capital. Do not use all capitals for acronyms.	`class Person;`
		`class PageCreator;`
		`class HtmlReader;`

Interfaces	Follow the conventions for *Class*	`interface Storing;` `interface PersonDelegate;`

Method	Should be verbs in mixed case with the first letter being lowercase, and the first letter of each internal word being a capital.	`run();`
		`getBackground();`
		`findPerson();`

Variable	Should be mixed case with the first letter being lowercase, and the first letter of each internal word being a capital.
	Names should be designed to indicate its intended use to a casual observer.
	One-character variable names are allowed for one-time use throw away variables.
	For integers use `i` to `n`;
	For characters use `c` to `e`.
		`int i;`
		`char c;`
		`String personName;`

Constant	Should be all uppercase and words separated by an underscore.	`static final int MIN_WIDTH = 4;` `static final int MAX_WIDTH = 99;`

Table 1. Naming conventions for identifiers.

3.8.1 Referring to Class Variables and Methods

Avoid using objects to access a class (`static`) variable or method. Instead, use a class name:

```
classMethod(); // OK
AClass.classMethod(); // OK
anObject.classMethod(); // AVOID
```

3. 8.2 Constants

Numerical constants should not be coded directly except for –1, 0 and 1, which can appear in, for example, `for` loops as counter values.

3.8.3 Variable Assignments

Avoid assigning multiple variables to the same value on a single line or using embedded assignments:

```
foo1 = foo2 = 2; // AVOID!
/*
* The following should be:
* a = b + c;
* d = a + r;
```

```
*/
d = (a = b + c) + r;
```

Do not use the assignment operator where it can be easily misinterpreted as the equality operator:

```
if (c++ = d++)
{
...
}
```

3.8.4 Parentheses

Ensure that the use of parentheses is very liberal. Always prefer to include parentheses as opposed to allowing possible operator precedence problems. This is still the case even if you think the operator precedence appears clear to you – it may not be so clear to another person!

```
if ((a == b) && (c == d)) // We prefer this...
if (a == b && c == d) // ...to this
```

3.8.5 Returning Values

Think twice about returning values dependent on certain criteria.

```
if (booleanExpression)
return false;
else
return true;
// The above is equivalent to the following!!!
return !booleanExpression;
// Here is another example!
if (condition)
return x;
else
return y;
// Again, the above is equivalent to the following!!!
return (condition ? x : y);
```

3.8.6 ?: Operator

If there is a binary operator in the condition before the `?` in the ternary operator `?:`, use parentheses:

```
return ((x >= 0) ? x : -x);
```

4. Maintenance documentation.

Your system will be the subject of maintenance, assuming that it gets used at all. Someone will have to deal with the support of the system and possibly the further development of it. In your professional career maintenance will often play a large and important role and it is usually regarded as an unpopular activity. We should aim to make it as easy and as painless as possible. Much maintenance carried out in industrial and commercial contexts is seriously hindered by a lack of documentation that prevents the engineer from fully understanding the system and what it is supposed to do. In the past the popular belief was that large amounts of design documents would be the resource that was the most effective basis on which to carry out different types of maintenance. In reality this is rarely the case as we discussed in an earlier chapter. The design documents may not fully reflect the source code, these designs may not have been updated as the requirements changed or as implementation problems drove the design away from the the-

oretical position adopted at the beginning. We have to provide some basic information relevant to the maintenance team that is reliable, understandable and complete, as far as is possible.

It is assumed that the requirements documents and user stories will be available. These are numbered and organised in a systematic way. The system metaphor and overall software architecture should also be present and should match the actual system. This is easier to achieve than trying to relate everything to a large design which may not have been updated during the development of the system because of the need to solve unforeseen problems in implementation, the changes to the requirements etc.

The code should be consistent with the coding standards and so the comments are useful and complete. They should refer to the other parts of the document so that we can trace how different parts of the code relate to the user stories.

The test sets that were used to demonstrate compliance with the requirements should also be available so that they could be rerun for retesting or parts of them used for regression testing (testing that checks that the overall system works properly when parts of the system have been changed).

Testing documents should be available from the project. We discussed how these should be designed, using tables and spread sheets to describe the tests and the test results. All this information should be preserved and included in the system documentation for future maintenance.

5. User manuals.

These are vital parts of the system, at least as important as the code in the sense that a poor manual will compromise the success of the system, people can't or won't use it properly, it fails to assist users in carrying out their tasks and so on.

What makes a good user manual and how can we write one?

We need to go back to think about the purpose of the system. This has already been encapsulated in the user stories, the user characteristics of the system and in some of the functional and non-functional requirements documents.

The document should start with a brief review of the purpose of the system and then provide a structured basis for carrying out all the tasks commonly expected. This should be written in simple, jargon-free language with plenty of screen shots and other simple diagrams to explain the processes described. A good *index* is vital as well as a *glossary* of the terms used.

Look at a few examples of user manuals for systems that you have used and ask yourselves how good they were for you. Generally, user manuals are written by technical people, often programmers in the project team, they are often written at the end of the project and they are often written poorly. In an XP context it is likely that some of the manual cannot be written until the end but quite a lot can be done beforehand, especially if the system is being delivered incrementally. In this case the manual will have an incremental structure.

Some may take the view that the system is so intuitive to use that no manual is necessary. This may be the case with some web-based systems, perhaps an e-commerce development or an in-

formation system based around a web browser. Do not make any assumptions about this. If you think that your system in intuitive and it is obvious how to use it then you should prove this. Choose some typical users and ask them to use it and observe them. You will probably be surprised at the difficulties some people have even with the *simplest* system. Many of the unpopular and unusable software systems of the past (and present) have been built under the assumptions that the use of them is obvious to all.

The requirement stories had estimates of the change likelihood and this will give you an indication of when parts of the manual can be written. Some authors, notably [Weiss1991], suggest that the manual should be written first, before the code, so that it provides clear information to the programmers and could be used as a basis for testing. We have, essentially adopted this position here with the use of X-machines as the basis for the specification of the test sets. The user manual could be a simplified version of the paths through the X-machine written in everyday language. As the requirements change and mature this will be reflected in the X-machine structure and thus the structure of the user manual.

The user manual, like everything else in the project, needs to be reviewed and tested with users or representatives of the type of people likely to be users. Creating a simple questionnaire for users to fill in as they use the manual to operate the finished system will provide helpful feedback, this can be used to show your client that you have tested the system thoroughly ready for acceptance.

One question that needs to be answered by the client is what type of user manual is needed. Should it be a paper booklet or an on-line system. There are advantages and disadvantages for both. One advantage with a small paper manual is that it can be easily flicked through and read prior to using the system. The on-line manual is easier to use when searching for information.

Some examples of a user manual produced by a student teams using XP is in Appendix C.

6. Version control.

One important practical issue which arises in the course of developing any software system is that there will be a number of versions of different documents created. These documents will include requirements documents, source code, test sets and user manuals at the very least. All will be available in different versions since they will have been developed and revised over a period of time. Many of these will refer to aspects of the system that changes as the development proceeds. Members of the development team will have to refer to these documents and will need some way of ensuring that at any given stage they are consulting the correct version. This may not be the most up-to-date version. We have therefore a *version control problem*. We need to keep track of what version each document has reached and which version we need to consult or change. In an XP development, where all team members are interchanging their roles and sharing the responsibilities for the entire project, it is vital that we avoid the situation where an individual is working on a version of a document that the others do not know about.

It is natural that some members will need to work away from the laboratory, perhaps on their own machine, and it is vital that they regularly update their colleagues with what they have done. At any rate, there should always be two people involved in any part of the development whether it is on the university machines or an individual's.

This problem is made worse by the fact that for some systems there may be a number of different configurations of the software which need to be produced, for instance to contain different special features for particular clients, or to run with different hardware or operating systems. Thus, as well as versions created at different times in the history of a system, there may also be different versions in parallel for different configurations of the software, and these need to be managed properly. This involves the task of *configuration management*. Furthermore, as a system develops there are likely to be different versions of each configuration of it that are released to the clients at different times: these different versions are often referred to as different releases of the system. Thus, in principle there are three aspects to be managed: the different versions of components, the different configurations and the different releases.

6.1. The project archive.
We need to set up a proper archive for the project in a systematic way and develop some conventions and protocols for its use. There are tools available that can help with this. One, CVS, is widely and freely available. It is worth investigating and asking for it or something similar to be installed on your machines.

The simplest approach is a shared directory on a network where the team have privileged access to this directory.

In this archive we will be putting documents of various types: requirements documents, story cards, test plans, source code, ancillary material, manuals, as well as management information such as minutes of meetings, plans and other material. One way to structure this is:

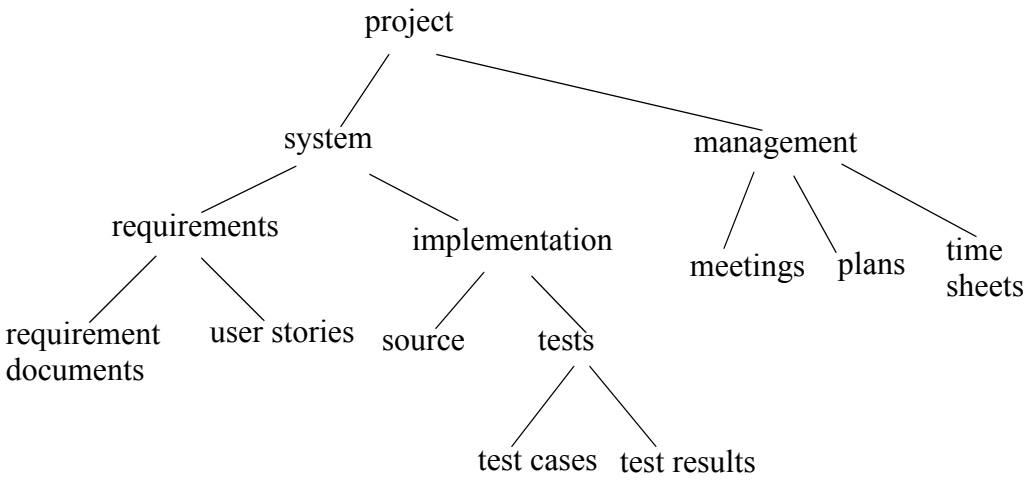


Figure 1. Project archive structure.

Within each basic component the archive will be organised in terms of versions and date of creation. It is sometimes useful to have a spreadsheet that describes what state each document is, for example whether it has been reviewed and confirmed as an acceptable product or whether it is still under development. The authors of the documents and the time of its creation and review are also useful to help everyone know what is going on.

Although XP stresses minimal bureaucracy we cannot use this as an excuse for an unprofes-

sional approach, remember there are people, your clients, who are depending on the outcome of your work and being sensible about how you organise things is vital.

6.2 Naming conventions.
A scheme for naming documents and components of the system should use sensible and descriptive names for the document, explaining, if possible, its nature as well as the relationships that will exist between it and other documents. There will be different versions of each document and these might also involve different releases and configurations of the whole system.

In terms of the version control, the usual basis is that the different releases will usually be numbered in a linear sequence involving a two-level scheme with major and minor releases. Each XP release should be given a new initial number. Then each release number will change when there are major changes to the functionality of the system such as new stories implemented and then integrated into a working system. Internal, minor release numbers will be used during the development of new stories to describe the various versions that are being developed prior to integration.

In traditional software development it is common that for each release there will be an alpha version (built for internal testing only), a beta version (built for release to a small number of selected clients for them to test), and then the final version which is released generally to all clients, so that the minor release numbers also need to identify whether this is an alpha, beta or full release of the system. In XP we hope to avoid the subsequent issue of patched or corrected versions by delivering in well tested increments and getting feedback from the client.

The scheme for naming documents (and other components of the system) is then based on this numbering scheme for releases, so that versions of these are given new numbers whenever they need to change in order to match the new release of the system. In practice, though, the naming scheme for documents will also need at least a third level of numbering, since it is likely that the updating of a document to match the development of a new release may happen in several stages, as new versions are inspected and corrected before finally being accepted.

7. Summary.

We have discussed the different types of documented associated with a software system, who reads it and what it's for. We have discussed the two main forms of documentation, paper-based and on-line. The project archive is considered as a key resource and a mechanism for preventing the project from descending into chaos. The issues of version control and configuration issues are also covered.

Exercises.
1. Read through the user manual in Appendix C. Is it clear, could you use the system following it? How could it be improved?
2. Each team pair should review another pair's code to see if it meets the coding standards. They should report back their findings to the whole group. If necessary the code should be refactored.

3. The group should review all of the software being developed and identify what stage it is at, the connection between the stories and the classes, the status of the code in terms of whether it passes all of the unit tests, what has been integrated and delivered, where any new requirements may be needed. All this should be documented carefully.

Conundrum.

When should the user manual be written, at the end when all is completed or much earlier?

Reference.

[Weiss1991],Weiss, Edmond H.. - "*How to write usable user documentation*", - 2nd ed. - The Oryx Press, 1991.

Chapter 11.

Reflecting on the process

Summary: What has been learned, what will be useful for projects in the future. How has XP been used, how can it be improved and adapted to different circumstances.

1. Skills and lessons learnt.

The satisfaction gained from delivering a high quality product to a grateful client is hard to beat. It makes much of the struggle and the hard times worth while. It's one of the attractive aspects of being a programmer and software engineer, solving people's problems using sophisticated and powerful technology.

In the course of doing this, however, you will find out a lot about software development, about programming, about writing good documentation and of quality assurance. You will also learn about working and communicating with a business client - an invaluable experience - and how a team operates. There will have been problems, problems with working with people, of trusting them and of being trusted. All of these experiences will be important in your development as a professional in the field.

You will also improve your programming skills, your knowledge of different languages and systems greatly. It is much better to do a real project with a team like this than any number of small programming exercises in traditional programming courses.

At the beginning a skills audit was suggested as a way of identifying what your abilities were before starting on this programme. It is a good idea to revisit this and to see how you may have developed. bear in mind, however, that as your skills become more sophisticated so do your expectations. Where once you felt that you were an average programmer, say, you may still feel that is the case, but what you can do now will be vastly improved compared with what you could do then. Try to estimate the improvement in terms of what aspects of the language you can now deal with easily and which were once hard or unknown.

Look at the way you manage things and each other. How you plan your time. It might not have been plain sailing but there are many things that you would do differently in a future project. Note these down.

2. The XP experience.

Here is an opportunity to reflect on the XP process. You will have found that it is quite hard to stick to all the 12 practices. The 4 general principles of XP; Communication, Simplicity, Feedback and Courage should have directed our efforts. How did it turn out? What was good and what was less useful about XP.

Did you find that writing tests first was hard? Was pair programming something that you enjoyed or hated? Was the relationship with the client good, did you communicate well, obtain feedback that was timely and helpful? Are you satisfied with the documentation that was pro-

duced. Did you get the balance right between recording useful and necessary information rather than producing material for the sake of it.

How did you get on with your team mates? Did everyone put in the same amount of effort? How could team working skills be improved?

How would you adapt XP to the sort of projects that you might be involved with in the future?

Does XP work or is it another case of theory being out of step with reality? Can only highly motivated and skilled programmers make XP a successful approach to software engineering?

These questions will be answered in the fullness of time. Your experiences may help to guide the process of finding better ways of building software in a fast changing world.

3. Summary.

Reflecting on your experiences with the XP approach, even though you might not have been able to follow all the practices completely will enable you to understand an alternative approach to software engineering to the traditional design-led approach. Dealing with a real client with a real business need should have also transformed your understanding of the business of software creation.

The references at the end are just a sample of the rapidly growing number of books on eXtreme Programming, they are often fairly accessible and provide a useful insight into many aspects of XP, both theoretical issues and experiences of XP in industry.

Exercises.
1. Write down your experiences of the project and of XP. What have you learnt? How enjoyable was it? Did XP deliver? How can XP be improved?
2. Look at some of the other books on Extreme Programming and see how they suggest an XP project should work and compare these accounts with your own experiences.
.

4. Conundrums - discussion.

Chapter 1. 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?
Answer. The IT director is right but probably for the wrong reasons. His priority is to protect his department, he wants to be involved in the software development, he hasn't the resources for a big new development but he could do something that would work up to a point. He doesn't want external consultants taking over and saddening him. He doesn't yet know how to connect the web front end to the transaction system but is confident that they can fix something.

In this case the right reasoning is based around business pragmatics. What the bank actually did was to get the internet site and front end working quickly and to commission it. At the time there was no connection with the legacy transaction system so a large number of typists were employed who printed off the internet transactions and typed the details into the mainframe system. This was initially expensive but effective. This bank grabbed 70% market share of this new business. This then generated the income to allow them to build an integrated system involving a newly engineered back office produced by their consultants.

Meanwhile the rival banks were building integrated solutions before going live, they never ever captured market share from the first bank.

Our bank demonstrated *agility* in both its business planning and in IT deployment. Using people for the data entry instead of a software alternative also demonstrated a *low risk* approach since the alternative was, at the time, an untried technology. This illustrates that business considerations as well as technical ones are vital, neither should dominate the other. In an ideal world the business leaders should have a deep and realistic knowledge of IT and the IT specialists should have an informed and pragmatic understanding of business realities and the fundamental need to be able to make money, to maximise market share and to deliver quality products and services. Agile methodologies need to be compatible with these objectives.

Chapter 2. 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?

Answer. Your client may have a very good reason for wanting something quick, there may be more business benefit in doing so. Our scenario relates to the business person's need to be able to demonstrate a piece of working software to the business backers who will decide on putting further money into the project. Showing something working, even if it did not have all the functionality required or was a little unreliable, to these backers was much more important than doing a good software engineering job. If the extra funding becomes availa-

ble then the proper engineering of a reliable and maintainable system would probably become a priority.
This also illustrates that the link between the business context and the software development process is fundamental. Traditional software engineering text books discuss approaches which are dominated by technical issues and the pursuit of quality without looking at how the real business pressures can force the way things have to be done. Although XP provides a number of practices that can guide us towards building high quality and relevant software solutions they shouldn't prevent us to respond to real business needs in favour of some abstract notion of how things should be done properly.

Chapter 3. 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.

Answer. First we need to review the objectives of the project. Sure, we all want to deliver a great system for the client. But we also want to learn more abour programming, particularly in this new language, we wanrt to learn how to work in an XP team and we want to learn how to manage a real project and work with our client. The team needs to discuss all of htese things in a rational and calm manner.

Let's look at a possible way forward. We need to look at the project plan. It will contain a number of on-going tasks, liaising with the client, writing code, working on stories and so on. We need to include amongst these the need to learn the new language. Schedule some sessions where the expert gives a tutorial to the others. Of course, this may seem like wasted time because no productive code is being generated but the benefits will come later. The expert should identify, with the others small pieces of code that they can produce in pairs. Meanwhile the expert looks at some other issues such as story analysis and the definition of both functional tests and unit tests. While this is being done the others are getting up to speed a little. After two or three weeks, if everyone works hard and with a positive attitude we might get to a position whereby any pair can program together in a reasonably effective way and they will get better all the time.

Chapter 4. Your team is in trouble. The client has not been in touch with her feedback on the proposed system. She doesn't have much experience of IT and only has a rather vague idea of what she wants. There are no similar systems known to you that you can show her. You need to start getting some requirements identified and some initial stories prepared.

Do you:
a) wait until she has thought further about the system she wants?
or
b) build a simple prototype using your imagination and background research in order to show her something that might stimulate her ideas?

Answer. Choose option b). The delay may be caused by the client not knowing how to proceed and lacking in confidence, it may be their first experience as a client and their knowl-

edge of IT is minimal. Some good suggestions from you could be a lifesaver for them. Even if they don't like your ideas they may stimulate them into suggesting things that are suitable. It also gives you an opportunity to be creative instead of just sitting around waiting.

Chapter 5. The company wanted an *intranet* that provided support for many of their business activities and also their personnel management. The site would contain information about the various company activities, a diary system and templates for administrative tasks such as the submission of illness and absence forms. The users would be able to log on remotely to carry out tasks as well as from within the company offices. The customer was able to maintain a very close relationship with the development team and had a clear idea of what the company needed. There were 3 teams using XP working on this project, each competing with all the others. Initially all the teams thought that the project would take 10 weeks. It didn't quite work out like that. When the first team delivered their first increment they discovered something important that had not emerged from the planning game. The company had a service agreement with a third party network solutions company which provided the computer system and the internet connection for the customer. This led to a serious problem for some of the teams and resulted in some failing to meet the 10 week deadline despite the careful planning.

What might have been the problem?

Answer. The network service provider had an important policy on the type of technology that they supported. Not all the teams were using a compatible approach. The customer did not know about the technicalities of this side of the project. The early delivery of a functioning piece of software brought the issues to light when the team tried to install it. The customer was then asked to negotiate with the network supplier to introduce the required support. This was eventually achieve but some teams still, unfortunately, used an unsupported technology and so, though their solutions were good, they did not work for this customer.

It is important to investigate all aspects of the customer's situation, including the services used and the constraints that may apply. Getting an early release installed at the customers site can help to identify problems like this.

Chapter 6. Two leading supermarket chains introduced their first internet ordering system at around the same time. Their e-commerce sites, although superficially looking similar, fared rather differently. One saw a much greater growth in business than the other. Yet the technology used, the warehousing and delivery systems were very comparable. Customers just didn't like using one of the sites.

What could have been the differences between the two user interfaces that made this happen? (It was nothing to do with the look and feel of the web pages or the way that the orders were managed or the price of the goods.)

Answer. One company relied on graphics arts specialist to design the web pages, the other used a combination of graphic artists and computer scientists. In the unsuccessful web site there were lots of attractive graphic images featuring a popular TV chef in the home pages. The other site had attractive graphics too, but these had been optimised in terms of their memory size without any obvious loss of image quality. This meant that the pages down loaded much faster, especially over slow modems. The other site took much longer and many prospective customers gave up waiting and switched to the other site. (Note that in the

UK local telephone charges can be significant.) Thorough testing under all likely conditions should have identified this problem - it shows that some non-functional requirements can be critical.

Chapter 7. A local retailer specializing in luxury goods commissioned an e-commerce system. This was completed and installed satisfactorily. The shop gave the job of printing out the internet orders and processing them through the orders system to one of the sales assistant to do at the end of their shift. This worked well at the beginning as the numbers of orders gradually grew.

After a few months of steady growth the sales figures of orders placed through the Web suddenly collapsed to nothing. Initially it was thought to be a software fault but no problems were found when we investigated.

What could have gone wrong?

Answer. Initially, it was thought that the volume of orders was too great for the database system chosen for the application. This hypothesis was soon rejected. Other thoughts focused on the architecture of the system and on the connection between the interface, business and database layers. Again, no problems were uncovered here. We then looked more carefully at how the system was operating in the business. The reason for the problem found to be because the increase in the number of orders was not accompanied by a corresponding increase in the people dealing with the orders. The assistant became increasingly frustrated at the volume of work which was having to be done after the end of their shift. The desperate solution taken was to delete all the internet orders instead of dealing with them.

This indicates that the system must be designed to include the human dimension as well as the computer. A management strategy should have been designed alongside the introduction of the computer system so that it could adapt to the needs of the business as these changed.

Chapter 8. About 20 years ago the UK Government purchased a system to deal with Air Traffic Control over London. It was based on a number of similar systems that had recently been installed in the USA. Unlike the American systems there were serious problems with the London system. Planes flying over London would suddenly disappear from the screen. Equally alarmingly, planes would suddenly appear as if from nowhere. Extensive testing was carried out, especially on the component of the system that was fed the radar information and dealt with the display of the positions of the planes on the screen. No defects were found, everything was exactly as the requirements demanded.

Why did the system work in the USA but not in London?

Answer. The system used latitude and longitude positions to track the planes. These were treated as real positive numbers. Now the problem is that London lies on the Greenwich Meridian, that is Longitude 0° and so planes would be moving across this line. Planes moving from West to East would have their Longitude value reduce from a positive number through 0 to negative values. The system had not be designed to deal with negative numbers so they were ignored. Similarly planes coming from the East had negative Longitude and were also invisible. Of course, there is no situation like this in America and so the problem was not considered during the development of the system.

The problem ultimately showed up during unit testing after the initial system test failed. The moral is to think the unthinkable, question all the assumptions and to do this make sure that they are all documented so. Then, in problems like this someone may realise that what might have been true under certain conditions may not be true under others.

Chapter 9. The customer was the IT director of the company which produced and sold biological specimens to research laboratories and pharmaceutical companies. The software was to support the entire company activities, which involved the production process - which had to be fully documented to meet government regulatory procedures - the stock control process, the ordering process and the invoicing and accounting processes.

After working extensively with the customer and delivering a number of incremental versions the customer was satisfied. A final delivery was made and at this point the customer invited several personnel from the company to attend the demonstrations. These potential users of the system pointed out many problems with the business concepts upon which the system was based. It became clear that the customer did not understand his company's business process. It was also clear that we will have to re-engineer the system. The new requirements were significantly different, there were few areas where the detail was the same although the overall architecture would be very similar.

Should we start again from scratch or try to adapt what we had already done, reusing and preserving what we could?

Answer. Genesys started out trying to adapt the system to the new requirement. Initially this seemed to involve redesigning some of the screens, altering the database and re-testing the system. This took a number of months but progress was held up by problems with system quality. Whenever we thought that we had tested and debugged on section we discovered more problems with it afterwards. After a while the decision was taken to start again. Because of the previous work on the system the new build was very rapid and a complete working system was delivered within a relatively short time to a quality that the customer and his colleagues were happy with.

Should we have started on a complete rebuild immediately after we found out that the requirements were wrong? It is difficult to answer this. Trying to adapt the original system did help us to capture and understand the new requirements and this enabled us to build the final system quickly. Perhaps this was the best strategy.

Chapter 10. When should the user manual be written, at the end when all is completed or much earlier?

Answer. Traditionally it's done at the end. It's often done by a junior member of the development team and is rarely tested with potential users.

Why not start thinking about it when we are creating the stories? A simple page or two giving the outline *script* for a user can be developed in tandem. As the stories get changed then the draft manual will also be changed. The development of the system metaphor's interface will create the framework for the user manual and the final screen shots can be inserted into the document when they are ready.

A final word.

We hear a lot about failed software projects, failure to deliver, failure to deliver something usable, failure to deliver something of value. Extreme programming and other agile development approaches are an attempt to try to improve matters. Whether they do provide a better way will be determined over the next few years. What is clear, however, are two basic points:

i) software development methods, including XP, will continue to evolve to try to meet the challenges of tomorrow;

and

ii) we cannot afford to ignore the complete environment of any software solution, the clients and customers, the users, the business process, the marketplace all will impact on the success of whatever we build.

The stories that were discussed in the *conundrums* are all real examples of situations that you might face in the future, adopting XP and ignoring any of the stakeholders in ii) may still result in failure.

There is still much research to be done on XP and the agile approach. What is certain is that many of the principles articulated by these ideas will stay with us. Your experiences trying to apply them in a real project in a professional way will provide you with a firm foundation for a future career in software development. It is not easy applying all these ideas, it needs discipline and perseverance, as you have probably found. Doing it well will, I am sure, leave you with a great sense of achievement.

Being agile might be hard but it's worth it!

References.

[Auer2001], K. Auer & R. Miller, "*Extreme Programming Applied*", Addison Wesley, 2001.

[Astels2001], D. Astels, "*Practical Guide to EXtreme Programming*", Prentice Hall, 2002.

[Jeffries2000], R. E. Jeffries, et al, "*Extreme Programming Installed*", Addison Wesley, 2000.

[Marchesi et al 2002], M. Marchesi, G. Succi, D. Wells, L. Williams, "*Extreme programming perspectives*", Addison-Wesley, 2002.

CONTENTS

Bibliography.

S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles. *Extensible Stylesheet Language (XSL) Version 1.0*. http://www.w3.org/TR/xsl/.

A. J. Albrecht, "*Measuring application development productivity*", SHARE/GUIDE/IBM Application development Symposium, 1979.

S. Ambler, "*Agile modelling*", John Wiley, 2002.

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.

D. Astels, "*Practical Guide to EXtreme Programming*", Prentice Hall, 2002.

K. Auer & R. Miller, "*Extreme Programming Applied*", Addison Wesley, 2001.

R. Banker, R. Kauffman et al, "*An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment*", J. Management Inf. Systems, 8, 127-150, 1992

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

B. Boehm, "*Software engineering economics*", Prentice-Hall, 1981.

B. Boehm et al, "*Cost models for future life cycle processes: COCOMO 2* ", Balzer Science, 1995.

T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. http://www.w3.org/TR/REC-xml

S. Brown, R. Burdick, J. Falkner, B. Galbraith, et al, "*Professional JSP*", 2nd Edition", 2001.

P. Checkland and J. Scholes, "*Soft systems methodology in action*", Wiley, 1990.

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

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

W. Cunningham & K. Beck, "A diagram for Object-oriented Programs", *Proceedings OOPSLA-86.*

S. Eilenberg, "*Automata, machines and languages*", Volume A, Academic Press, 1974.

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

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

M. Holcombe & F. Ipate, "*Correct systems: building a business process solution*", Springer, 1998 available on-line at: http://www.dcs.shef.ac.uk/~wmlh/

M. Holcombe & A. Stratton, "VICI - experiences with running student software companies". in *Project'98 - projects in the computing curriculum, (with A. Stratton, S. Fincher, G. Gillespie), Springer, 1998, 103-116.*

M. Holcombe, A. Stratton & P. Croll, " Bringing industrial clients into the student learning process," in *Project'98 - projects in the computing curriculum, (with A. Stratton, S. Fincher, G. Gillespie), Springer, 1998, 47-69.*

W. S. Humphrey, "*A Discipline for Software Engineering*", Addison-Wesley, 1996.

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

F. Ipate & M. Holcombe, "Specification and testing using generalised machines: a presentation and a case study." *Software Testing, Verification and Reliability. 8, 61-81, 1998.*

[ISO 9126]

R. E. Jeffries, et al, "*Extreme Programming Installed*", Addison Wesley, 2000.
C. B. Jones, "*Systematic software development using VDM*", Prentice Hall, 1986.
M. Marchesi, G. Succi, D. Wells, L. Williams, "*Extreme programming perspectives*", Addison-Wesley, 2002.
R. Miller, "When pairs disagree, 1-2-3", in D. Wells & L. Williams (eds), *XP/Agile Universe 2002*, Springer-Verlag, Lecture Notes in Computer Science, vol. 2418, pp. 231 - 236, 2002.
M. Poppendieck & T. Poppendieck, *Lean development: a toolkit for software development managers*, to be published by Addison-Wesley, 2003
R. S. Pressman, *"Software Engineering a practitioner's approach,"* McGraw Hill, 2000.
B. Schneiderman, *Designing the User Interface*, Addison-Wesley, 1998.
K. Schwaber & M. Beedle, "*Agile software development with SCRUM*", Prentice Hall, 2002.
I. Sommerville, *"Software Engineering*", Addison-Wesley, 2000.
J. M. Spivey, "*The Z notation: A reference manual*", (2nd. Edition). Prentice Hall, 1992.
S. St. Laurent & E. Ceramie, *Building XML applications*, McGraw Hill, 1999.
J. Stapleton, "*DSDM: The Dynamic Systems Development Method,*" Addison Wesley, 1997.
Weiss, Edmond H.. - "*How to write usable user documentation*", - 2nd ed. - The Oryx Press, 1991.
J. Yuan, M. Holcombe & M. Gheorghe, "Where do unit tests come from?", *Submitted to XP200*3.