Chapter 2

EXTREME PROGRAMMING outlined

Summary: The fundamental principles and the 5 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.

2.1 Some guiding principles

Before we get into the details of the main approach taken in the book, an evolution of Extreme Programming (XP), which incorporates many of the aspects described in the agile methods discussed in Chapter 1, we will consider some of the issues from a broader perspective.

Software development is a human activity and we must ensure that the human dimension is at the centre of our thoughts when we discuss ways to make software more effectively.

There is a social dimension in which groups of people, developers, customers, managers collaborate together to achieve a common aim – the development of a software solution to a business problem. But, it is not just the achievement of this that is important. We all develop and learn as individuals and groups and this has to be at the forefront of things as well.

There are a number of social principles that apply here. No project is without its challenges and any group of people comes with its dynamic relationships. If we are aware of these from the start then we will be able to both benefit from their positive aspects and manage any possible negative ones.

We should consider how people think and feel about things – their work, their environment, their relationships, their hope and fears. Everyone experiences many different emotions and have many different needs. Recognising that people are individuals and respecting that is a key pillar upon which XP is built.

Thus people want to succeed in their work, to enjoy what they do, to feel a sense of achievement, to learn and improve their knowledge and technical skills, to be able to relax with colleagues and to manage their responsibilities without the stigma of failure.

From a technical point of view there are many issues to be considered.

Fundamentally, we are engaged in the development of something of value – a product which may have some economic benefit – it pays our wages and contributes to the company's profits; it benefits our customers by providing them with enhanced capacity to achieve their business objectives – which may not be business based in the narrow interpretation of the term but enables them to do their work better (many of our customers are charities or public sector organisations which are not necessarily profit driven).

Thus we need to think about value and its costs in time and money in a holistic way and these will be at the core of our work.

A key mechanism for keeping these three things together is to proceed in very small steps – this way one can see how well you are doing and if we can combine this with a view of where we are going – accepting that this may not be totally clear at all times – then we can make progress.

Adding value and demonstrating this should be the objective throughout, we can see how this is achieved if we maintain a constant relationship with each other – developers, customers and managers – all the time. To do that we need to *communicate* – to talk to each other, to show each other what we have done, to discuss where we are going, to reflect on what has been achieved in an honest way, to help each other if things go wrong, to keep relationships in a positive state. Sometimes we should think about developing and maintaining these relationships outside of the work environment through social and leisure activities.

A dominant theme is the *flow* of activities. We will focus on building frequent small releases of code that can demonstrate some value to the project; we will reflect on what has happened frequently and regularly; we will respond to setbacks in a positive way that enhances our understanding of the development process and of our professional and personal capabilities; we will promote discussion and sharing of perspectives and knowledge and we will try to improve ourselves, our team and our organisation bit by bit.

These fundamental principles lead to a set of values that will form the basis of XP, good communication; simplicity; feedback; courage and respect.

2.2 The five 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 five basic values of XP are:

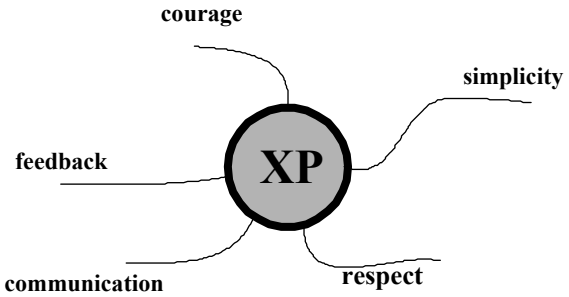

Figure 2.1 The values of XP

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.

Stakeholders include:

customers – managers, financial directors, marketing departments etc.
users – administrative staff, general public etc.
developers – programmers, managers, financial directors, marketing etc.
media and other organisations who may take an interest.

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.

Naturally the phrase "business value" may have a number of interpretations – it can affect organisations that are not commercial companies – charities, non-profit organisations etc. The point about the term is that there is some benefit to the organisation – it makes them more efficient, produce better quality outcomes, improves the experiences of their workers, customers or some other stakeholder. It does not always have a clear financial benefit and it is important to take a holistic view of things.

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. 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 client's quality expectations. We must avoid this -

previous approaches such as Rapid Applications Development (RAD) sometimes 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 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

---

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.)

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 ones 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 develops. 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 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.

Respect

The most important issue about any joint, team-based activity are the relationships between the participants. Many of the problems that arise in software development projects are 'people issues' rather than technical ones. People issues are generally about relationships - between developers in the team and between team members and clients, users, managers and others in the immediate working environment. Of course, relationships with those close to you and your family can have an impact on a project as well.

In order to deal with these you have to treat each individual with respect - allow them to express their point of view, discuss things with them calmly and to actively delegate responsibility for doing things in a reasonable way and to trust them to get on with it. The two complementary facets of responsibility and trust are very important. XP is supposed to be a human way of doing things that is built on both these pillars. If you respect someone you will also trust them and then give them responsibility.

2.3 The twelve basic practices of XP

2.3.1 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 can help to supply some tests, they understand the business processes in their organisation – or should do – and these, if they are to be replicated or involved in the system being built will provide a wealth of potential test material. 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 and debugging as activities are Cinderella subjects 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 centred 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.

2.3.2 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 will be using the keyboard and the other will be looking at the screen with them, both discussing what they are doing.

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.

2.3.3 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's 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. This project dates from before we adopted XP.

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.

We can use many ways of trying to bridge the gap between what the client thinks they want and what the development team thinks the client wants. It may not be based solely on writing things down, videos of potential users working; recordings of interviews with stakeholders, studies of similar systems used by other organisations including competitors are also valuable sources of information.

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.

2.3.4 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.

Stories, which are about small pieces of working software, should be developed over a short time – a week is a good target – and need to be customer focussed – the customer must understand them and their place within the overall system.

From time to time – maybe around monthly – a review of the business objectives should be carried out with the customers and users.

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 involve people with professional skills in testing – the development team should develop an understanding of systematic system testing. 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. There is a tendency to be too optimistic – there may be problems ahead with the use of the technology chosen, the supply of data needed to build the system as well as revisions to what the client wants. All of these things confuse the estimation process. Only by recording carefully what you think will be the cost of implementation at each stage and seeing how it changes over time will you develop the experience to make more reliable judgements in the future.

Once a collection of meaningful stories have been agreed and the cost estimated 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.

Of course, things won't always go smoothly. It is important to build in some slack to use if problems arise that could impact on the progress of the project.

2.3.5 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 that 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 processes 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.

2.3.6 Small, frequent releases

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.

2.3.7 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.

2.3.8 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 could be done every few minutes if that is possible. Some of the work can be automated – particularly running tests. Working code is committed to a CVS or similar repository.

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 that 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.

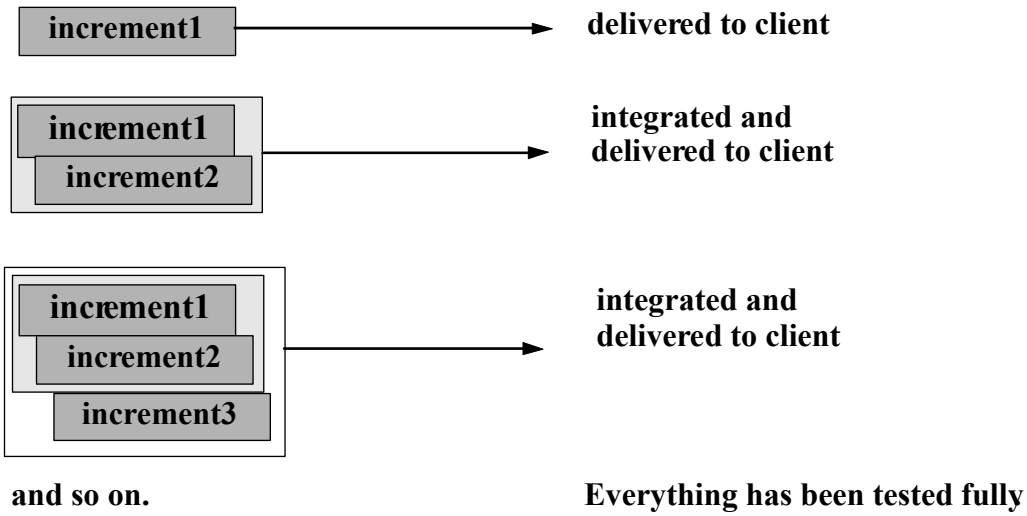

Figure 2.2 Incremental delivery

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.

2.3.9 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.

2.3.10 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.

2.3.11 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 browser* of which there are a number supporting different programming languages.

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.

2.3.12 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.

2.4 Review

We have briefly described the original values and practices of XP. They seem to make a lot of sense but do they work in real projects? The first thing to say is that this style of software development may not be suitable for all types of application and industry. Very large projects involving hundreds of developers will be extremely difficult to bring to a successful conclusion whatever method of working is employed. It may be that some of the elements of XP will be useful in these situations - test first is an obvious one - but only time will tell. Certainly, current methods are problematical as a perusal of the trade literature and its articles about recent failed projects will testify. Another important point is that the XP approach is evolving all the time as we learn from experience. There is probably less emphasis on these 12 practices as specific activities that have to be engaged with in a dogmatic way but an emphasis on the principles underlying them and their adaptation to the specific context of an agile development. We have adapted and strengthened some of these ideas as we shall see shortly.

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 requirement 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.

2.5 The evidence for XP

There has been a lot of research into whether agile developments actually deliver what is claimed. In particular some of the practices discussed above have been considered to see if they work. In many cases the detailed experiments involve students carrying out tasks in a laboratory setting. Unfortunately these experiments often lack credibility in the sense that it is not possible to generalise their finding to real industrial settings – in other words the results lack external validity. Secondly, by taking the individual practices and considering them in isolation from the other practices may also fail to provide evidence for the benefits of XP and other 'complete' agile methodologies.

2.5.1 Evidence for Test First

The evidence for the benefits of a Test First approach is mixed. The first thing to say is that it is actually quite difficult to carry out – nearly all of the approaches to teaching introductory programming that are found in universities tend to downplay testing – if it is considered seriously at all. Thus many programmers find the idea of writing a test set before they start coding unnatural. This makes it difficult to carry out comparative experiments without having to undertake some extensive training on the technique.

Janzen [Janzen2005] found that the design of the software was 'better' – smaller and less complex units whereas [Siniaalto2007] found that the quality of the design was poorer. There are a number of other examples of inconclusive results e.g. [Muller2002].

One comment that is worth making is that the time that tests are written and used is

only one of a number of factors that might impact on the benefits of doing Test First, another is the type of tests created – testing is very dependent on the capabilities of the test sets to detect faults and so influences the ultimate quality of the software under test. Simple measures such as test coverage can provide some insight into the quality and effectiveness of a test set but things are more complicated than that. Test design is discussed in more detail in Chapter 7.

2.5.2 Evidence for pair programming

Williams [Williams2000] presented some evidence for the benefits of pair programming. The approach has been criticised on blogs such as hacknot.info [hacknot2004] and other experiments by other researchers [Nawrocki2001] have produced different conclusions. In practice, pair programming is not for everyone since some people's personality is such that they seem to be unable to cope with the intense relationships with their partners that are needed. Our experience has been that, in the right context, that is, a real project with a real customer and combined with the other XP practices, pair programming works for most people – many of whom become very enthusiastic about it.

2.5.3 Evidence for XP

There is again a problem in identifying conclusive and convincing evidence about the benefits of XP. Carrying out comparative experiments in an industrial setting is always going to be a problem and much of the evidence is based on simple experiments involving students. Macias [Macias 2003] found a small benefit in a comparative study involving small industrial projects. Abdullah [Abdullah2003} found that the more XP practices used the better the quality of the software, again in small industrial projects. Abdullah also found [Abdullah2006] that there was evidence that teams using XP experienced a higher level of well-being than teams using a design-led approach involved in the same projects.

There are many critics of XP and the book [Stevens2003] M. Stephens and D. Rosenberg *Extreme Programming Refactored: The Case Against XP,* Apress, 2006.takes a number of aspects of XP and criticises them. Some of the criticism is based more on the exaggerated claims of some who were not adopting the full set of practices; other issues need a more detailed consideration. Part of the rationale of this book is to consider these issues in the light of our experiences over 7 years with hundreds of XP projects their programmers and their clients.

2.6 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 agile project, based on the XP principles, 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 where it is possible to discuss what you are doing without disturbing other users.

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.

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}

output:
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.

Table 1:

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}

output:

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.

For the other practices, discuss how you might be able to adopt them – what are the difficulties, how might they benefit you?

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

[Abdullah2003] Sharifah Syed-Abdullah, Mike Holcombe, Marian Gheorghe, Practice Makes Perfect, *Extreme Programming and Agile Processes in Software Engineering*, *Lecture Notes in Computer Science*, Springer LNCS Volume 2675, 2003
[Abdullah2006] Syed-Abdullah, S., Holcombe, M., & Gheorge, M. (2006). The Impact of an Agile Methodology on the Well Being of Development Teams. *Empirical Software Engineering, 11*(1), 143-167, 2006
[Beck1999] K. Beck, *"Extreme Programming Explained"*, Addison-Wesley, 1999.
[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. - .
[Janzen2005] Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. *Computer*, vol. 38, pp. 43-50. (2005)
[Jeffries2001] R. Jeffries, "*XP Installed"* is available on the website (www.xprogramming.com)
[Macias2003] Macías, F., Holcombe, M., & Gheorghe, M. (2003). A Formal Experiment Comparing Extreme Programming with Traditional Software Construction. In the *Proceedings of the Fourth Mexican International Conference on Computer Science* (ENC 2003). Tlaxcala, México, 73-80. Sep. 8-12, 2003
[Nawrocki2001] J Nawrocki, A Wojciechowski - European Software Control and Metrics (Escom), 2001 [http://www2.umassd.edu/SWPI/xp/pairprogramming/Nawrocki.pdf] [Stevens2003] M. Stephens and D. Rosenberg *Extreme Programming Refactored: The Case Against XP,* Apress, 2006.
[Williams2000] Williams, L. Kessler, R.R. Cunningham, W. Jeffries, R. Strengthening the case for pair programming, *IEEE Software*, 17, pp 19-25, 2000

Web sites

http://www.xp.programming.com
http://www.extremeprogramming.org/
http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
http://www.hacknot.info/hacknot/action/showEntry?eid=50