Chapter 4

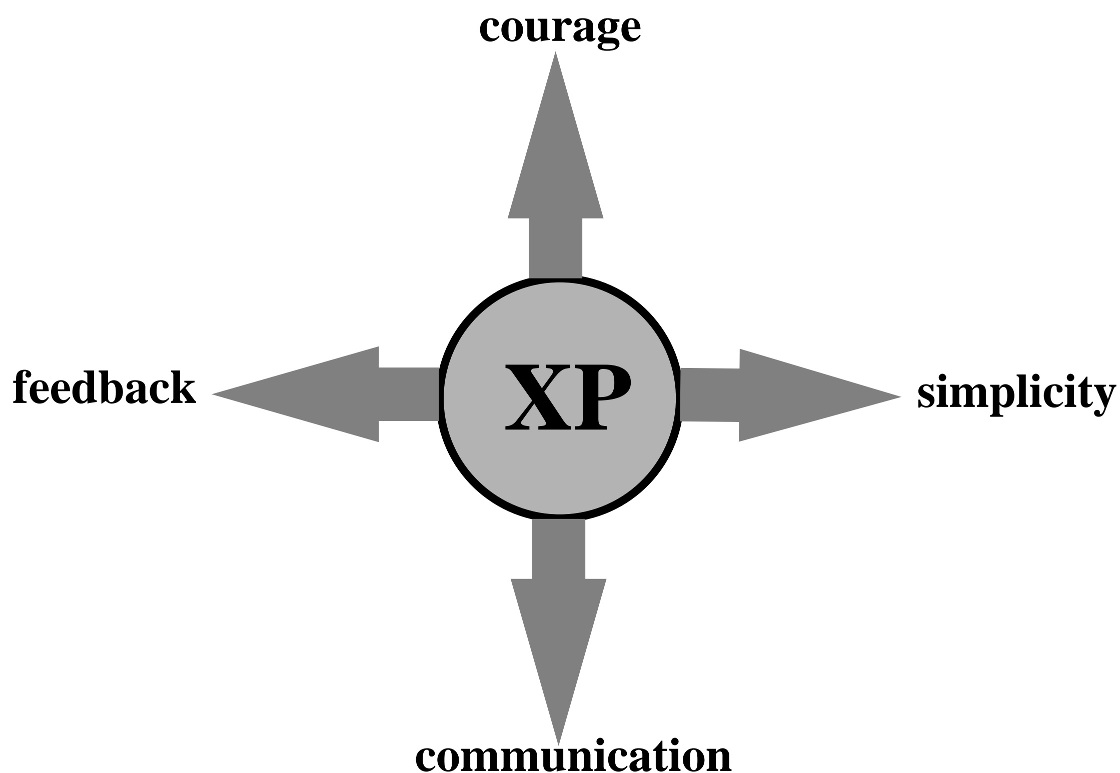
XP outlined

Summary: The 4 values and the 12 activities involved in XP are introduced.

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 the developers and the clients, 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 principle 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 of 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 between the developers and the clients. This is also, obviously 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. (*Aside*. 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!)

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

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.

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.

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.

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.

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.

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 between pairs 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. 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. 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.

It 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.

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 [Gilb**], 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 implicilty, building models of the behaviour of parts of the system. This is an important reource 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.

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 represen-

tative of what you are trying to do.

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

2.6. Small, frequent release.

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

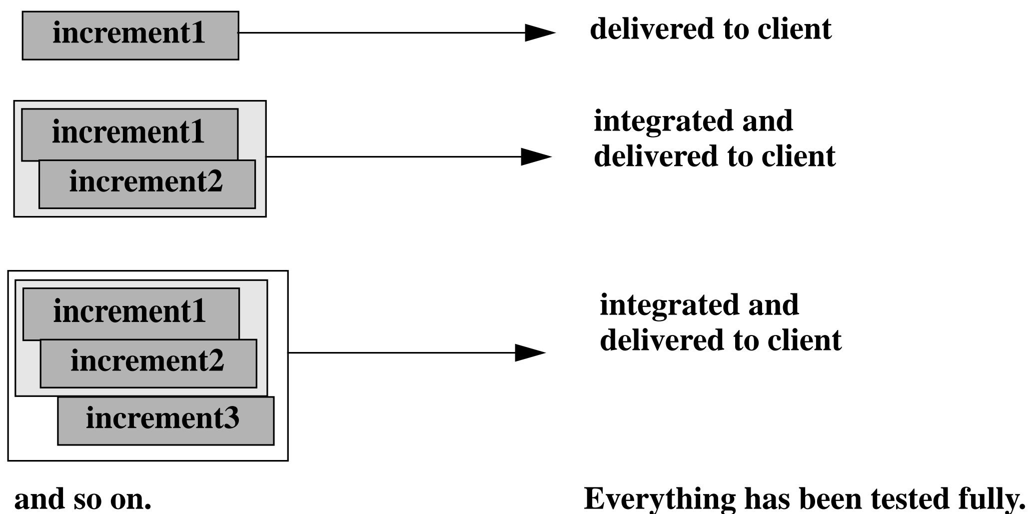
For the programmer this philosophy could be embodied in the practise of using, for example, the minimum number of classes and methods to pass the tests. There are some dangers, here, however, and they will be looked at under 2.11. Simplicity of code does not always correspond to simplicity of function,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



2.9. Coding standards.

These define rules for shared code ownership and for communication between different team'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 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. This is 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
and so on.

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

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.

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

References.

[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. - .