An Introduction to Agile Methods

Steve Hayes (Khatovar Technology)
steve@khatovartech.com
http://www.khatovartech.com

Martin Andrews (Object Consulting)
http://www.objectconsulting.com.au

Contents

The Need for New Software Development Processes

"Modern software systems are hard to build, because we've already built the easy ones" - Tom DeMarco

We've been writing software for 30 years, but we haven't made a dent in the software backlog. Instead, our successes have fuelled peoples' imaginations, so that the more software we write the more people want. Consequently, software managers and developers are continually looking for better ways to develop software, and looking on a number of fronts. Compared to 30 years ago we have much cheaper, faster computers, more powerful programming languages, a plethora of supporting tools, better education, and better understanding of the theory of software development. The Internet has changed the way that people communicate, accelerated the transfer of information, and radically altered people's expectations of how software should work. We also have a number of different software processes/methodologies that purport to tell us the "best" way to develop software, and that's the aspect of software development that we're going to focus on for the next two days.

Existing Software development Processes

There have been a lot of software development processes created over the years. In "Rapid Development", Steve McConnell identifies a number of categories of process (most real-life projects employ a blend of these):

- Pure waterfall
- Code-and-fix
- Spiral
- Modified Waterfalls
- Evolutionary Prototyping
- Staged Delivery
- Evolutionary Delivery
- Design-to-Schedule
- Design-to-Tools
- Commercial Off-the-shelf Software

With the exception of code-and-fix, these processes have a few things in common – they assume that software development is analogous to a defined industrial process; they are based on physical engineering processes; they are predictive; and they assume that people can be treated as abstract resources.

"It is typical to adopt the defined (theoretical) modelling approach when the underlying mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the defined approach, the empirical approach is the appropriate choice" –
Process Dynamics, Modelling, and Control, Ogunnaike and Ray, Oxford University Press, 1992

In process control theory, a “defined” process is one that can be designed and run repeatedly with predictable results. In particular, the inputs to the process and the process itself must have very low levels of noise. When a process cannot be defined precisely enough to guarantee predictable results it is known as a “complex” process. Complex processes require an empirical control model. An empirical control model entails frequent inspection and adaptive response.

Software development is not a defined process, at the very least because the main inputs to the process activities are people. Traditional methodologies contain lists of activities and processes that are depicted as reliable and repeatable, however we all know from experience that this isn't the case. Ask two different groups of people to come up with class diagrams from the same requirements and you'll get two quite different results. The problems with treating software development as a defined process are discussed further by Schwaber and Beedle[1].

Physical engineering processes consist of distinct phases. The **design** phase is difficult to predict and requires creativity and imagination. The primary activities in the design phase are intellectual. The design phase is followed by **planning** for the **construction** phase, and then by the construction phase itself. The primary activities in the construction phase are physical, and it is much easier to plan and predict than the design phase. In many physical engineering disciplines construction is much bigger in both cost and time than design and planning. For example, when you build a bridge, the cost of design is about 10% of the total job, with the rest being construction[2].

For the analogy between software development and physical engineering to hold, we need to be able to separate design from construction, be able to produce a predictable schedule, design artefacts that are complete enough for construction to be straightforward, and be able to perform construction with people of lower skills (who are hopefully cheaper to employ). Construction also needs to be sufficiently large in time and effort to make this worthwhile.

The situation in software development is quite different. Rubin[3] reports the following breakdown of software development by activity:

- Analysis 16%
- Design 17%
- Code/Unit Test 34%
- System/Integration Test 18%
- Documentation 8%
- Implementation/Install 7%

In an even more extreme analysis proposed by Jack Reeves[4] - he suggests that code is analogous to an engineering design, and that software is “constructed” by running a compiler. Accordingly, the construction stage in a software development project is essentially free.

Regardless of the exact numbers, it's clear that as a percentage of the overall project effort, construction is much less significant in software development than it is in physical engineering. If construction is analogous to coding, our experience indicates that it's very difficult and expensive to

[1]“Agile Software Development with Scrum”, Ken Schwaber and Mike Beedle, Prentice Hall, 2002, p89ff
[2]“The New Methodology”, Martin Fowler, http://www.martinfowler.com/articles/newMethodology.html
[3]“Worldwide Benchmark Project Report”, Howard Rubin, Rubin Systems Inc., 1995
[4] “What is software design?”, Jack Reeves, http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm

produce design artefacts detailed enough for construction to be straightforward. If construction is analogous to compilation, then we can effectively ignore the construction effort. In either case, processes based on analogies with physical engineering should be treated with suspicion.

The Pitfalls of traditional methodologies

Traditional methodologies also try to be predictive - to create a schedule at the beginning of a project and to conform to this schedule for the life of the project. Yet a common complaint is “the problem with this project is that the users keep changing their minds”. In the physical world people accept that requirements need to be fixed because it's intuitively obvious to them that, because of the expensive construction phase, it's very expensive to make changes after a certain point. However, software is much less tangible. Not only is it hard to be precise about what's needed, it's also hard to see why it should be difficult to change later. Customers **expect** software to be soft. Traditional methodologies establish procedures that discourage requirements changes, they resist change. This helps them maintain a predictable schedule, but it does nothing to ensure that the final results meets the customers real, changing needs. While predictability may be very desirable, it can only be achieved by seriously compromising other qualities – particularly the fit of the final result to the real (emerging) requirements.

The desire for predictability also leads to treating individuals as resources. A typical project plan will indicate that the project requires “three programmers”. Once again, the assumption is that the project can be managed using a defined process, and that the outcome won't depend on the individuals that are assigned. But in our hearts we know that this isn't true – that the outcome depends heavily on the individuals involved. Alistair Cockburn[5] in particular has presented forceful arguments as to why people need to be considered the most influential factor in project success or failure.

Given these flaws in traditional software development methodologies, it’s worthwhile asking why they ever worked, and if they worked in the past, why abandon them now? Has something changed? Fundamentally, the business world has changed. Thomas Friedman[6] argues that during the 1980s a number of fundamental changes occurred in the world, and that the result was that there’s now the “Fast World”, the world of globalisation, and the “Slow World”, who have walled themselves off from globalisation. Software development happens in the Fast World. The changes that Friedman identifies – the simultaneous democratisation of technology, finance, and information, coupled with the removal of political barriers maintained during the Cold War – have produced not just quantitative but qualitative changes in the business world. But fundamentally, things happen faster. Competitors appear, mature, and succeed or fail, in shorter and shorter cycles. Existing businesses change direction just as quickly in response. Product cycles are shorter, and consumers are fickle. It’s no surprise that methodologies that were adequate in the 70s, when product and business cycles were much longer, are being stretched to the point of failure in the 80s and 90s.

[5]“Characterizing People as Non-linear, First-Order Components in Software Development”, Alistair Cockburn, Humans and Technology Technical Report, October 1999

[6] “The Lexus and the Olive Tree”, Thomas Friedman, Harper Collins Publishers, 2000

The Emergence of Agile Methods

During the 90s a number of different people realised that things had somehow changed. These people became interested in developing software methodologies that were a better fit with the new business environment – more nimble, easier to manoeuvre through a turbulent business environment. Although the details of these methodologies differ, they all share certain underlying principles, to the extent that these methodologies are often now grouped under the title "agile methodologies".

Given the opportunity to reject the "software engineering" metaphor and start all over again, what would we consider as we were developing a new methodology?

- Software development is predominantly a design activity
- Characteristics of individuals are the first-order influence on project activities
- Modern software development can't rely on individuals – it requires effective teams
- Customers are unlikely to know what they want in detail before they start
- Customers need to be able to change requirements without unreasonable penalties
- The process needs to be flexible
- Responsibility should be aligned with authority

The Agile Alliance

The methodologies falling under the "agile" umbrella all address these issues in slightly different ways, however the agile methodologies do have common underlying values and principles. The Agile Alliance expressed the values in the Agile Manifesto[7]:

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - **Individuals and interactions** over processes and tools
> - **Working software** over comprehensive documentation
> - **Customer collaboration** over contract negotiation
> - **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more."

They also expressed the principles behind the manifesto[8]:

- 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 time-scale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support

[7]http://www.agilealliance.org
[8]http://www.agilealliance.org/principles.html

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 maximising the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organising teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly."

Existing Agile Methods

Jim Highsmith has done a great job of examining and comparing the major agile methodologies[9]. The following synopses are taken from the introduction to his book.

Scrum

Scrum, named for the scrum in Rugby, was initially developed by Ken Schwaber and Jeff Sutherland, with later collaborations with Mike Beedle. Scrum provides a project management framework that focuses development into 30-day Sprint cycles in which a specified set of Backlog features are delivered. The core practice in Scrum is the use of daily 15-minute team meetings for coordination and integration. Scrum has been in use for nearly ten years and has been used to successfully deliver a wide range of products.

Dynamic Systems Development Method (DSDM)

The Dynamic Systems Development Method was developed in the U.K. in the mid-1990s. It is an outgrowth of, and extension to, rapid application development (RAD) practices. DSDM boasts the best-supported training and documentation of any ASDE[10], at least in Europe. DSDM's nine principles include active user involvement, frequent delivery, team decision making, integrated testing throughout the project life cycle, and reversible changes in development.

Crystal Methods

Alistair Cockburn is the author of the "Crystal" family of people-centred methods. Alistair is a "methodology archaeologist" who has interviewed dozens of project teams worldwide trying to separate what actually works from what people say should work. Alistair, and Crystal, focuses on the people aspects of development – collaboration, good citizenship, and cooperation. Alistair uses project size, criticality, and objectives to craft appropriately configured practices for each member of the Crystal family of methodologies.

[9] "Agile Software Development Ecosystems", Jim Highsmith, Pearson Education, 2002
[10] Highsmith uses the term "Agile Software Development Ecosystem", or ASDE, instead of "agile methodology"

Feature Driven Development

Jeff De Luca and Peter Coad collaborated on Feature-Driven Development. FDD consists of a minimalist, five-step process that focuses on developing an overall "shape" object model, building a features list, and then planning-by-feature followed by iterative design-by-feature and build-by-feature steps. FDD's processes are brief (each is described on a single page), and two key roles in FDD are chief architect and chief programmer.

Lean Development

The most strategic-oriented ASDE is also the least known: Bob Charette's Lean Development, which is derived from the principles of lean production, the restructuring of the Japanese automobile industry that occurred in the 1980's. In LD, Bob extends traditional methodology's view of change as a risk of loss to be controlled with restrictive management practices to a view of change producing "opportunities" to be pursued during "risk entrepreneurship". LD has been used successfully on a number of large telecommunications projects in Europe.

Extreme Programming (XP)

Extreme Programming, or XP to most aficionados, was developed by Kent Beck, Ward Cunningham, and Ron Jeffries. XP preaches the values of community, simplicity, feedback and courage. Important aspects of XP are its contribution to altering the view of the cost of change and its emphasis on technical excellence through refactoring and test-first development. XP provides a *system* of dynamic practices, whose integrity as a holistic unit has been proven. XP has clearly garnered the most interest of any of the Agile approaches.

Adaptive Software Development

Adaptive Software Development, my own [Jim Highsmith] contribution to the Agile movement, provides a philosophical background for Agile methods, showing how software development organisations can respond to the turbulence of the current business climate by harnessing rather than avoiding change. ASD contains both practices – iterative development, feature-based planning, customer focus group reviews – and an "Agile" management philosophy called Leadership-Collaboration management.

As Jim Highsmith indicates, most people who are interested in agile software development are interested in Extreme Programming (XP). This isn't to say that XP is the **best** of the agile methodologies, just the best known. Most XP projects also incorporate practices and principles from other agile methodologies, but Extreme Programming makes a very useful starting point for studying agile methodologies. We'll focus on Extreme Programming for the rest of this discussion.

More About Extreme Programming

Extreme Programming is an agile, adaptive software develop methodology with a well-defined set of values and core practices.

- **Agile** : XP is consistent with the values and principles of the Agile Alliance
- **Adaptive** : XP's documented practices are only a starting point – XP teams adapt the process to improve their results
- **Values**: Since XP is adaptive, the details vary from one XP project to another. But the underlying values stay the same. If the values change, then the process is no longer XP
- **Core Practices** : XP specifies a set of mutually-supporting practices that encourage collaboration and reduce the cost of change

XP does not have complex rules, and it does not try to specify exactly how to respond to every possible situation the team will encounter, instead, XP tries to be "barely sufficient[11]". Dee W. Hock, founder of the Visa organisation, made these points[12]:

- Simple, clear purpose and principles give rise to complex, intelligent behaviour
- Complex rules and regulations give rise to simple, stupid behaviour

XP specifies the rules – the values and the practices – and lets the team figure out the detailed behaviour.

Extreme Programming Values and Principles

The XP values are:

- Communication
- Simplicity
- Feedback
- Courage

An XP project relies on these four values. If your organisation or team doesn't truly share these values, then an XP project will fail. Of course, most of those values are motherhood-and-apple pie – it would be hard to find an organisation that said that it didn't believe in them. XP tries to remove some of the vagueness from these values by describing principles that embody the values.

- Open, honest communication
- Quality work
- Rapid feedback at all levels
- Assume Simplicity

[11]Jim Highsmith and Alistair Cockburn both suggest that a methodology should be "barely sufficient" - one that will inject the right amount of rigour for the situation without overburdening the team with unnecessary bureaucracy.
[12]"Birth of the Chaordic Age", Dee W. Hock, Berrett-Koehler, 1999

- Embrace change
- Play to win
- Concrete experiments
- Small initial investment
- Incremental change
- Accepted responsibility
- Honest measurement
- Travel light
- Teach learning
- Local adaptation

Open, honest communication

Most software development problems can be traced back to communication failures – someone didn't talk to the right person, or they talked to the right person at the wrong time, or they didn't talk to anyone all. XP encourages communication by using practices that can't be done without communicating openly and honestly. Very few projects fail because good news was suppressed – it's how the bad news is treated that makes the difference. This means that programmers need to be able to explain the consequences of decisions, to express their fears, and to admit when they've simply screwed up. Of course, we use practices that reduce the chances of things going wrong, but no matter what we do, on every project something **will** go wrong.

Quality work

Let's stop for a moment and consider a simple model for controlling software development projects. This model says that although there are lots of details and variations, fundamentally there are only four things that a manager can manipulate to control a software development – cost, time, scope and quality. When a project is running successfully, most project managers will leave the control variables just as they are – it's what happens when a project starts to fail by some measure that's interesting.

A project manager might choose to correct a failing project by allowing for increased costs. The extra money might be spent on new hardware or, more often, extra staff time. They might choose to allow the project to run for more time with fewer resources (since using the same resources for longer would increase costs as well). Another way to make the project successful might be to leave the time and costs the same, but get the customer to agree to cutting some features from the requirements. And finally, the project manager might choose to leave the costs, time and scope the same, but reduce the quality of the resulting application. In this context it doesn't really matter what definition of quality the project is using, only that reducing quality reduces the required development effort.

Unfortunately, these control variables aren't independent, and the relationships between them aren't linear. For example, we've already mentioned that extending time while leaving resources the same usually also increases costs. Increasing costs late in a project may have very little effect, since making the new resources productive may require considerable effort. This situation is summarised in Brook's Law – "Adding manpower to a late software project makes it later"[13].

[13] "The Mythical Man Month", Fred Brooks, Addison-Wesley, 1975

When projects are operating in simple, undemanding environments, the project manager can set all four control variables (cost, time, scope and quality) and the project will still succeed. However, most commercial projects operate in constrained environments, where there have to be trade-offs between these variables. In general, the project manager can fix three of these control variables, but the fourth one needs to be free to find its own level. When project managers try to fix all four variables in a constrained environment, almost invariably the programmers seize on quality as the floating control variable, because it's the least visible to outside observers. Furthermore, once quality starts to decrease, everything else starts to slip, and the slippage quickly accelerates because of the non-linear relationships between the control variables. XP projects fix quality (Kent Beck says there are only two possible values for quality in an XP project - “excellent” and “insanely excellent”[14]) and use scope as the floating control variable. Plus, people just enjoy their jobs more when they can do quality work!

Rapid feedback at all levels

In a world of constant change, the only way we can steer a project to a successful conclusion is by getting constant feedback. Feedback from the customer at the scale of weeks, months and days, and feedback from the system and our peers at the scale of days, hours and minutes. Constant feedback helps ensure that the system is stable and of high-quality. Feedback from the customer helps ensure that we're building the right system. XP projects get feedback from the customer through frequent releases of a working, production quality system. Feedback from peers is achieved through pair-programming (more on this later), and feedback from the system is achieved through exhaustive automated testing, and continuous integration of the entire system.

Assume simplicity

Historically, software developers were encouraged to predict the future. They were encouraged to write “general purpose” frameworks to handle both current and future requirements. They were encouraged to do this because the cost of making changes later, to running software, was excessive. The result, though, was complex software that was difficult to understand and to maintain, and which frequently contained features that were never used. XP takes a different position – do everything you can to keep the cost of change low, and then focus your resources on doing the simplest thing that you can, right now. Some people like to interpret “simplest thing” as the easiest thing to do, for example to duplicate code through cut-and-paste, but this isn’t what XP means. The simplest thing is the thing that looks simple to the people who come later – the thing that communicates all the relevant ideas clearly, and nothing else. Developing a simple thing is often a lot of hard work.

Embrace change

XP projects welcome change, because changes tell them that people are learning - the programmers are learning how to implement the system better, and the customers are learning how to change what the system does to get better business value.

[14]“Extreme Programming Explained”, Kent Beck, Addison Wesley, 2000, p38

Play to win

Many projects produce artefacts that seem to add no value to the customer or to the developers. These artefacts seem only to have value if the project fails, when they'll be useful in determining how to assign blame for the failure. The **courage** value helps you to believe that the project will succeed, and do everything you can to make it succeed, without worrying about covering your ass. It tells you not to try to preserve the schedule by lowering quality, not to shy away from high-risk, high-reward situations. Courage alone would be just an excuse to hack, but coupled with the values of communication, simplicity and feedback, it opens a world of new possibilities.

Concrete experiments

Frequently, software design discussions devolve into a debate over some theoretical characteristic of the software : “We can't do that, it will take up too much memory”, “No it won't, it will be fine”. XP says that no matter which way you go, there's a chance that you're wrong – so try it and see. Write some simple code, almost always throwaway, that highlights the issue being discussed and lets you see which approach will be more useful in practice. Once you accept concrete experiments as a normal way to make decisions, discussions get much shorter, and your foundations become a lot more stable because they’re based on factual results, not theoretical arguments.

Small initial investment

Keeping a project lean at the beginning keeps it focussed on the customer's immediate requirements and helps maintain simplicity.

Incremental change

The system is evolved slowly, changing one thing at a time, and leaving the system stable and complete after each change. This maintains the confidence of the programmers, and lets the customer both experiment with and adapt to the changes. Rapid, revolutionary change has the opposite effect – it makes the programmers nervous, eliminates opportunities to get feedback from the customers, and forces the customers to absorb many changes at once.

Honest measurement

In our urge to create predictive schedules we've often fallen into the trap of over-specifying our measurements, for example saying a project will take 43 days, when we believe it will really take anywhere between one and two months. Programmers may understand that 43 days is only an estimate, but customers hear a precise number and expect a precise match. XP projects report measurements including uncertainties, and learn to live with this. XP projects also make sure that these measurements are visible to everyone on the team – the developers, the customer, and every other stakeholder. Without this approach to measurement we can't have open and honest communication, particularly between the programmers and the customer.

Accepted Responsibility

No one likes to be told what to do all the time, especially if it looks like the task is impossible. Instead, XP teams let the team members choose what they want to do - to accept the responsibility for completing a task rather than having it assigned to them. This doesn't necessarily mean that everyone always does exactly what they want – the team needs to make sure that all the tasks are assigned, no matter how unpopular some tasks may be, but the responsibility is distributed across the team, not centralised in an individual such as a team leader or project manager. This also isn't an exercise in fixing blame, but we have noticed that when things do go wrong people are much more willing to accept responsibility for tasks they chose and estimated themselves than for tasks assigned to them by someone else.

Travel light

XP teams, willing to embrace change, need to be able to change direction quickly and easily, adapting to changes feedback from the system and from the customer. This means they only produce the things that they need, and nothing else. XP teams often have less written documentation than teams using traditional approaches, because more of the information they need is embedded in working software or communicated verbally with less effort than writing. What documentation they do have is trusted, and the documentation which can't be trusted doesn't exist – quite a contrast to traditional software development.

Teach learning

Adaptive methodologies give teams a great deal of freedom to customise the process and its practices, but this means that the team members need to understand why things are done the way they are, what alternatives are available, and the pros and cons associated with each alternative. They also need to be attentive to what is and isn't working on their current project, and have the time and skills necessary to reflect on the situation. This means it isn't enough to teach the team what to do, you have to teach them how to learn as well.

Local adaptation

No one knows the details of your current project like you do. Don't expect any methodology you get from a book to be ideally suited to you. Do the best you can, understand the choices available to you, then change your process to suit your circumstances. The emphasis here needs to be on understanding the available choices – this means using a practice for a while (even if you don't like it) so you can get experience with it, understand the advantages and disadvantages of the specific practice, and see how it interacts with other practices. No one should just choose a set or subset of practices based on their initial impressions.

Extreme Programming Practices in brief

XP is an adaptive method, so although all XP teams need to consistently embrace the values and principles, the details of the actual day-to-day operation of an XP project vary from team to team. A reasonable question then is "where do I start?". XP provides a set of daily practices that, used together, have been demonstrated to efficiently produce high quality software. These practices are:

- Whole Team
- Planning Game
- Customer Tests
- Small Releases
- Simple Designs
- Pair Programming

- Test-Driven Development
- Design Improvement
- Continuous Integration
- Collective Ownership
- Coding Standard
- Sustainable Pace
- Metaphor

Whole team

Many methodologies rely on a divide-and-conquer kind of strategy. The development process is broken down into distinct steps, different people with different skills are required at each step, and results are communicated from one step to the next through paper documents, CASE tool repositories or the like. On an XP team everyone is involved all the time, and the team members communicate with one another by talking. This is a very effective strategy, but one that requires everyone to be in what Alistair Cockburn calls "radical physical colocation" - they need to sit together. The team needs to include, at a minimum, a business representative (the "customer") and some programmers, but it may also include a coach, who helps keeps the team on track, a manager, who allocates resources and removes external impediments, and any number of other specialists. Everyone is kept fully informed, and everyone works together. The time between a question and an accurate answer is kept as close to zero as possible.

The Planning Game

In traditional software projects the emphasis is on one specific question: will we be done by the due date? In an agile project, this changes to two different questions: how much will get done by the due date?, and what should be done next?. Many methodologies are predictive – they make a prediction of what will happen over the course of the project, and any deviations from this prediction are treated as "errors". XP is adaptive – it continually makes predictions about what will get done, but these predictions are adjusted based on what is actually happening – its the prediction that's considered to be in error, not the actual progress.

Release Planning is where the customer articulates the required features, the programmers estimate the difficulty, and together they lay out a preliminary plan an initial order of development and estimate what can be accomplished with the available people/money/time.

Iteration Planning is where a subset of the required features are broken down into tasks, estimated in more detail, and allocated to programmers. The amount of work that can be accomplished in the next iteration is simply based on the amount of work accomplished in the previous iteration.

Customer Tests

Customer tests address two common problems in software development – how do the programmers know when they're finished?, and how does the team know that everything that was working last iteration is still working this iteration? For each feature, the XP customer defines one or more automated tests that will show that the feature is working. The programmers know that the feature is complete when the test works, and they can verify that all existing features are still operational by running all the customer tests. This ensures that development never inadvertently goes backwards.

Small Releases

XP teams practice small releases in a number of ways:

They incrementally improve the system, adding small features or parts of a feature, every day (see Continuous Integration).

They release running, tested software that can be deployed to a production environment at the end of every iteration. This means that the customer can actually use the system, providing feedback to the team, and that the customer is regularly receiving observable value for the development to date.

They release to the final end-users as frequently as possible – every iteration for some teams. This is the best way to get high quality feedback on the system.

Simple Design

XP teams consider the simplest thing that could possibly work, and try to implement it, confident that subsequent changes will be supported easily (this is often captured by the acronym DTSTTCPW – "do the simplest thing that could possibly work"). XP teams view design as something done continuously through the course of the development, not something done just at the beginning. Keeping things simple also keeps things easy to understand and removes friction often created by software "flexibility" that isn't required yet.

A complementary principle is to keep things simple by not implementing anything you don't need right now, often known as YAGNI ("you ain't gonna need it").

Experience can be misleading when it comes to keeping things simple. Part of the Agile Universe 2001 conference was an on-site XP development, using developers drawn from the conference attendees. The application was intended to manage the purchase of music for performance in a choral group, and the most important screen was one that showed performers, required music, and orders as a matrix. The customer was an experienced business analyst who stayed with the development for the whole conference.

Unfortunately, their experience told them that to get the structured output they wanted, they needed to have data entry screens first, so that's what they asked for first.

An alternative approach was to simply use text editors to create data files at first, and implement the structured output in the first iteration, adding data entry screens later. This is the simplest thing that will work and let us focus on delivering value to the business.

Pair Programming

All production work in an XP team is done by two programmers sitting side-by-side working on the same machine. Your intuition may tell you that this is unproductive, and this would be true if programmers were constrained by their typing speed. Kent Beck has noted that if programming was actually constrained by typing speed, then the solution to the programming backlog would be to distribute typing tutor software to every developer. However, as we all know intuitively, "programming isn't typing"[15]. In fact, it is much more about thinking and designing small pieces of code, in which case another perspective can be very helpful. Both research[16] and the experience of hundreds of

[15] cited in http://www.xprogramming.com/xpmag/refactoringisntrework.htm
[16] http://www.pairprogramming.com

pairs is that overall, pair-programming is more productive than working alone. Pair programming encourages programmers to focus on the task at hand, and it also helps to disseminate knowledge around the team, particularly if you change the pairs regularly. Some programmers automatically object to pair programming, without even trying it, but most programmers enjoy it once they have tried it.

Test-Driven Development

XP teams don't add any functionality unless they have an automated test case that's broken. This means that they have to write the test for a feature, or an interface, before the code that implements the feature or interface. This helps with two things – first, it forces programmers to focus on the "what", the interface itself, before the "how", the implementation. Writing tests becomes a design exercise. Second, it means that there are tests for every facet of the system, and any pair that makes a change can be confident that there is a test that will tell them if they break something. In this environment, change is low cost and encouraged. These tests are designed, implemented and owned by the programmers, and are distinct from the customer tests that we already mentioned.

Design Improvement

As an XP team incrementally delivers new features they learn about the domain and about the system. They see where functionality can be generalised or leveraged. They find commonality where once they only saw differences, or learn to add differences to previously common behaviour. As they learn they need to be able to evolve the design, using a process called "refactoring". Design improvement is facilitated by simple design, test-driven development, and pair programming, all of which help reduce the cost of change.

Continuous Integration

XP projects are team activities, and XP teams understand that it's important that everyone's code works together. Rather than developing in parallel streams for long periods, XP teams force integration as often as they can. It's not uncommon for pairs to check code into version control and integrate it hourly, and many people do it more often. XP teams don't have code freezes, or go through integration hell – they just integrate all the time.

Collective Code Ownership

Since XP projects are team activities, it's reasonable that anyone can change anything, anywhere. This eliminates bottlenecks ("I can't work on that until Steve's free") and helps improve code quality because each piece of code is looked at by many people. It's feasible to do this on an XP project because pair programming lets people be familiar with a very large proportion of the code, or to be working with someone who is, and because tests catch inadvertent errors.

Coding Standard

All the code produced by an XP team should look like it was produced by the same (very competent) person. An XP team needs this to support effective pairing and collective ownership.

Sustainable Pace

Most project teams work as fast as they can on any given day, starting off very fast and then slowing down, quickly. They slow down because their daily practices cause the accrual of software entropy – stuff that gets in the way of day to day development. This includes duplicated code, long methods, classes that do multiple things, poorly named classes and methods, and dead code, but the sources of increased entropy are limitless. XP teams want to go at the fastest pace that they can sustain indefinitely. This means that each day they spend some time on practices that help them go faster tomorrow, or the day after. This makes software development more sustainable and predictable.

Metaphor

For effective communication, everyone in an XP team needs to be able to talk about the system using the same vocabulary and language. Often they communicate through a metaphor – through the fiction that the software system is the same as some other system that they're already familiar with. Sometimes the metaphor is elegant ("it's like a spreadsheet", or "it's like a library"), sometimes it's a bit more naive ("it's just like the business"). Having a metaphor is useful because it can increase communication compression (allowing team members to communicate a lot of information quickly) and because the metaphor can serve as an architectural vision.

Other Useful Practices

The practices mentioned in the previous section constitute Extreme Programming "out-of-the-box". However, XP is still an evolving discipline, and we've found some other practices that are consistent with the underlying XP principles and add value for most projects. These are:

- Stand-up Meetings
- Retrospectives

Stand-up Meetings

XP projects rely on creating effective teams – groups where everyone is moving in the same direction, towards the same goals. While it's easy to start a team moving in one direction, people start to diverge very quickly, and individuals often encounter details that require subtle changes in direction for the whole team. To keep everyone synchronised, most XP teams have a brief, daily meeting where each developer reports on three things:

- What have they done since the last meeting?
- What do they plan to do before the next meeting?
- Do they have any obstacles?

These meetings are called stand-up meetings because no one takes a seat – keeping everyone on their feet helps them to focus on keeping the meetings short. Any issues that require longer discussion are deferred to follow-up meetings that only the interested parties attend, rather than the whole team.

Retrospectives

It’s extremely unlikely that any software process “out of the box” is an ideal match to your team and your organisation. XP is no exception. Some of the practices may need to be modified, and you may need to add some extra practices. You may even decide that an XP practice doesn’t work for you and you need to discard it. At the end of every iteration an XP team should have a retrospective, discussing what went well, and what needs improvement in the next iteration. This is one manifestation of the values of “teach learning” and “local adaptation” – a team needs to learn about the strengths and weakness of itself and its environment, and adapt its processes to match.

Selected Extreme Programming Practices in Detail

Test-Driven Development

We can't expect to do credit to Test Driven Development (TDD) in the time we have available – TDD is going to be the subject of an entire book by Kent Beck in the near future – but we can provide you with the basics and some examples.

At the heart of test-driven development are two very simple practices[17]:

- Write new code only if you first have a failing automated test
- Remove duplication

Of course, it's easy to say this, and a little harder to actually execute. Before you can do this effectively, you need to have the right technical environment. You need to be working in a language that encourages incremental development, and you need to have a testing tool that lets you write small tests at low cost. Fortunately, most modern languages do support incremental development, and the Extreme Programming community has been the source of testing tools to support these languages, generically called the xUnit frameworks.

The definitive version of xUnit is the Java version, specifically know as JUnit. JUnit was originally written by Kent Beck and Erich Gamma, and has now been downloaded hundreds of thousands of times[18]. We'll show examples of writing tests in JUnit, but you should be aware that you can get equivalent software for many other languages[19].

Customer Tests

There are lots of different approaches to writing tests – this is another one of those topics that's going to occupy an entire book[20]. Traditionally it's the quality assurance department that decides if the application meets the customer's requirements, and they do it at the end of the development process using predominantly manual tests. In an XP project the quality assurance group still has a role, but they get involved in the project earlier, and work more closely with both the customer and the programmers. It's the customer who decides what aspects of the application need to be tested, and who construct many of the test scenarios, but it's frequently the quality assurance professionals and programmers who actually write the tests or the tools to support them. A common approach is for the programmers to write customer tests using the same tools that they use to write unit tests, but at the direction of the customer. This approach is common because it doesn't require any new tools, but it has the disadvantage that the programmers are liable to include the same errors of interpretation in both the tests and the implementation, which reduces the effectiveness of the tests.

A better approach is for the customers to write their own tests, independently of the programmers, but with the support of the quality assurance professionals. To do this the customers need to have access to the appropriate tools, but the complexity of the tools required varies enormously from one project to another. Some projects use a spreadsheet to capture inputs and expected outputs, and then generate test code from the spreadsheets. Some projects use XML instead of a spreadsheet. Other projects use a GUI scripting tool to drive the final application. Regardless of the approach that's

[17] "Test Driven Development", Kent Beck, Addison Wesley, 2002
[18] Junit is supported at http://www.junit.org
[19] http://www.xprogramming.com/software.htm
[20] "Testing Extreme Programming", Lisa Crispin, Ken S. Rosenthal, Tip House, Addison Wesley, 2002 (November)

used, the customer tests need to be under version control, just like the code.

A script for a GUI test might look like this:

1. Open the window
2. Put “Hayes” in the customer surname field
3. Select the “Outstanding Accounts only” radio button
4. Press “Display Accounts”
5. Verify that the Accounts list includes the following accounts and balances:

2311532	$100.00
9082343	$10,000.00
4208523	$852.42

A developer would be responsible for translating this into some executable form.

Design Improvement

Although an XP team will apply design improvements whenever they gain some new insight into the domain or the application, lots of the design improvement over the course of the project comes from judicious refactoring. Refactoring is a practice that emerged from the Smalltalk community, who were accustomed to making both large and small changes to working applications, with the support of the sophisticated Smalltalk programming environment. Although refactoring was common in the Smalltalk community it wasn’t documented as a rigorous or repeatable process – learning to refactor really consisted of sitting with an experienced Smalltalk developer and watching what they did, and even then they might never do the same thing twice. This approach made refactoring relatively inaccessible, especially to people involved with programming languages other than Smalltalk.

Fortunately, Martin Fowler became involved with an experienced Smalltalk team, and decided to document the practices that he observed. The result was “Refactoring: Improving the Design of Existing Code”, which documents both the process of refactoring and 72 distinct refactoring patterns. Applying any given refactoring pattern to an application leaves the overall functionality of the application unchanged, but improves some aspect of the design. An individual refactoring may produce almost no discernible change, but the relentless application of many refactorings produces remarkable changes in the overall design. Martin also deviated from historical practice and documented his refactorings in Java, which appealed to a much larger audience than Smalltalk. Martin’s work has provided a consistent, base vocabulary for refactoring, and the foundation for increased automation of refactoring in languages like Smalltalk and Java. Other languages are less amenable to automation, but can be expected to be supported more extensively in the future.

One of the keys to successful refactoring is to perform a refactoring as soon as you see it – to refactor mercilessly. Refactoring needs to be considered an integral part of the programming activity, rather than a distinct task that needs to be scheduled or performed separately from normal programming.

Continuous Integration

The fundamental basis of continuous integration is some process that regularly takes the application all the way from raw source code to the final product, and verifies that the functionality of the final product is as expected. The simplest approach is to write a batch file that gets all the source code from version control (including the test source), compiles it, runs the tests, and if the tests succeed copies the resulting executable to some trusted area for use by the customer. However, continuous integration can also get much more sophisticated.

The Java world is particularly well served with tools to support continuous integration. One of the most popular is the Ant project from Jakarta, which is a Java based replacement for the 'make' family of tools. Ant scripts are based on projects, targets and tasks, where tasks are either predefined units of functionality or locally created extension tasks. Although Ant itself is written in Java, it can be used to control the build of applications written in any language.

Another tool is CruiseControl. CruiseControl extends Ant by providing a set of tasks to automate the checkout/build/test cycle, providing a web page for viewing the status of the current build as well as the results of previous builds, and by automatically emailing the team when a build fails. Builds based solely on Ant typically run at regular intervals, or immediately on the completion of the previous build. Adding CruiseControl lets you run a build immediately after a change to the source code repository, but only when there's been a change.

The Planning Game

Fundamentally, the planning game is about ensuring that business decisions are made by business people, and technical decisions are made by technical people. While this might sound like an obvious principle, on many projects it isn't true. Take this scenario for example: a business person asks a programmer how long it will take to implement half a dozen new features. The programmer estimates that it will take 6 weeks, but the business person responds that it has to be done in 4 weeks. The programmer says "well, I'll try". What's gone wrong here? Well, first the business person has made a technical decision, when they decided how long it would take to implement a set of features. But what's also about to happen is that the programmer will make at least one, and probably many, business decisions. Knowing that they probably won't get all the features implemented in four weeks, they'll decide which one to start with and which ones to leave to last. This sort or prioritisation should be done by a business person. A healthier approach would be for the programmer to say, "I can't get all of these done in four weeks. Tell me which ones are the most important to you and I'll do those. If there's time left over at the end you can tell me which one to do next". The planning game formalises this approach by introducing simple planning rules.

In an XP project all the features required of the application are recorded as stories. A story is a short description of a feature – enough for the developers to estimate the work required, but certainly not enough information to complete the implementation. Stories are similar to use cases, but with a lot less detail than most use cases, which makes them much faster to record, and less of an investment. Stories belong to the customer – anyone on the team, including the developers, can and should suggest stories, but only the customer can decide if a story is worth adding to the application.

Once the customers have come up with a list of stories for the application, the developers need to assign estimates to each story. Stories don't need to be the same size, so there might be large variations in the estimates. When the developers get to a story that they can't estimate, then they go back to the customers for more information, and they usually break the original story down into smaller pieces that can be estimated. It's not important to estimate the actual time that each story will take to implement – the important thing is to judge the relative size of the stories eg. "writing report ABC will take twice as long as writing report XYZ". At the end of the estimation exercise the actual units of the estimates are discarded, but many people find that estimating in some sort of real units (eg. person-weeks) is a good way to start.

Once all the stories have been estimated, the team decides on an iteration size. Although the XP literature suggests this can be anything from one to three weeks (and Scrum uses 30 calendar days), most XP projects are trending towards one week iterations.

Given the iteration size, the developers guess how many story units they can complete in an iteration. Before development work begins this is very much a guess, which will quickly be refined by feedback from the first few iterations.

The next stage of planning is called Release Planning. The objective is to create chunks of functionality that make sense from a business point of view, and which could be released to some external audience. Often each release has a specific purpose as well – eg. Demonstrate progress to the sponsor; meet a contract milestone; demonstrate functionality at a trade show. The customers generally pick the release dates, and the developers and customers collaborate to decide what stories should be included in each release.

It's important to understand that releases don't need to be built just from the existing stories. Discussing each release may suggest new stories that should be added to the plan, and it frequently suggests that existing stories need to be broken into smaller pieces, with each piece allocated to a different release.

At this point you may be thinking that the release plan looks very much like a traditional project plan. One the surface this is true, in that the plan consists of lists of features and dates. However, there are two important differences. The first difference is the way the plan was constructed – as a collaboration between the customers and the developers. Traditional planning normally takes requirements from the customers and then has the developers construct a plan – there's very little iteration or collaboration. The other difference is the subsequent attitude to the plan. In traditional projects the plan is treated as the truth, cast in stone, and deviations from the plan are treated as errors that need to be corrected. An XP release plan is a flexible guide to what will be implemented – the XP customer steers the project by adjusting priorities and altering scope based on the feedback from each development iteration.

Once the team has a release plan they begin on the first iteration. The customer chooses stories whose estimates total the team's estimated velocity for development. The developers, as a group, break the stories down into engineering tasks, and estimate the engineering tasks. Some engineering tasks may be common to a number of stories in the iteration, many will be unique to a single story. Breaking the stories down into engineering tasks may require more information from the customer, and may affect the estimate assigned to the story.

Once the stories are broken down into engineering tasks, developers take responsibility for engineering tasks. This is quite different to traditional projects, where the project manager or team leader assigns tasks to individual developers. In an XP project responsibility is accepted, not assigned. Although individuals take responsibility for tasks, it's understood that part of the responsibility is finding partners to help with the pair-development of the task.

Throughout the iteration, often via stand-up meetings, the developers and the customer keep track of how much has actually been done, and adjust resources to try to maximise productivity, using XP technical practices as a guide. At the end of the iteration, the team records how many story points were actually completed, and individual developers record how many engineering task points they completed. This tells the team how many story points to commit to for the next iteration (and how many points will probably be completed in the release), and individuals how many engineering task points to commit to in the next iteration. Using the points actually delivered in the last iteration as the commitment level for the next iteration is sometimes referred to as "yesterday's weather"[21], because of the observation that 70% of the time today's weather is the same as yesterday's weather.

Now the team goes into a new planning game, with a new commitment level based on actual experience rather than a guess. The customer picks stories, the developers break them down into engineering tasks, sign up for tasks, and start development. This cycle repeats until the end of the release, and then the team moves on to the next release.

Extreme Programming Step-by-Step Example

The Steps	**Comments from an example project**
Iteration 0	
Build continuous integration environment	
Select and install development tools	
Customers list all the stories they expect to include in the application	
Developers put estimates against each story	
When a story is too large or vague to be estimated, the customers break it down into smaller stories	
Developers re-estimate new stories	We have 250 points of stories
Team decides on an iteration size	1 week
Developers guess initial velocity	10 points per iteration
Team decides how many iterations to have in each release	4 iterations per release
Customers group stories into releases. Each release should have an overall purpose and produce a reasonable, working application. Stories may need to be broken down across different releases	Release 1 is for sponsors, and shows navigation between screens. Release 2 fills in data entry Release 3 fills in reporting Release 4 automates notification of various conditions Subsequent releases are left undefined
Team discusses and constructs metaphor/architecture	This can occur anywhere, but needs to be completed before the first iteration. The team here thinks the metaphor is a bit weak and will need to be revisited
Release 1	
Iteration 1	
Planning Game	
Developers use velocity guess from Iteration 0	10 points per iteration
Customers pick stories to be included in	

[21] "Planning Extreme Programming", Kent Beck and Martin Fowler, Addison-Wesley, 2001

iteration	
Some stories need to be split apart	
Developers estimate new stories	
Developers break stories down into engineering tasks	Engineering tasks include Layout screens Link screens Provide dummy data
Individuals sign up for tasks	GUI experts take responsibility for most tasks, and ask for other team members to be available to pair. One person takes responsibility for dummy data.
Development	
Individuals negotiate pairs and start development	Pairs are mostly staying fixed for each day as the team explores the technology and metaphor
Incrementally, pairs do design, write tests, refactor existing code to make changes easier, and make changes	
Automated, continuous integration is ongoing	
Daily stand up meetings	
Task owners report progress since last stand-up, intentions for day, and any barriers to development	In Thursday's stand-up meeting, the team notes that two 2 point stories are not going to be completed by the end of the iteration
Team assigns resources to remove barriers	In a separate meeting, the customer breaks each story into two 1 point stories, one of which can be completed and the other of which will be ignored
More development, as above	
Iteration 2	
Retrospective	Customer demonstrates completed stories from Iteration 1. This ensures that everyone on the team is familiar with the functionality
Everyone in the team, including the customer, comments on what went well and what could be improved	The customer notes that some developers reported completion of stories without confirming the functionality with the customer, or asking for acceptance tests
The team agrees on things to change for the next iteration, including changes to the process	The team agrees to focus more on their interaction with the customer, and to revisit this in the next retrospective
Planning Game	
Developers use completed story points from Iteration 1 as the velocity for Iteration 2 ("yesterday's weather")	8 points
Customers pick stories to be included in iteration	The customer decides not to implement the two 1 point stories not implemented in Iteration 1, as these were the less important parts of the Iteration 1 stories. The customer picks 6 points of stories, and wants to include one more 3 point story.

Some stories need to be split apart and developers estimate new stories	There isn't enough room in the iteration, so the 3 point story is split into two 2 point stories – there's no reason the points need to add up the original total. The first 2 point story is included in this iteration.
Developers break stories down into engineering tasks	
Individuals sign up for tasks	People who overcommitted last iteration cut their commitment based on their personal, engineering version of yesterday's weather.
Development	
Individuals negotiate pairs and start development	Pairs are now swapping multiple times a day, as the team has learned to break work down into smaller engineering tasks, and people are more familiar with the project metaphor
Incrementally, pairs do design, write tests, refactor existing code to make changes easier, and make changes	
Automated, continuous integration is ongoing	
Daily stand up meetings	
Task owners report progress since last stand-up, intentions for day, and any barriers to development	In Tuesday's meeting, the team notes the compile time for the application is growing, and one developer volunteers to have a look at this. Another developer volunteers to pair with them. In Wednesday's meeting, one developer notes that they're probably going to finish their commitments before the end of the iteration. The rest of the team is on track.
Team assigns resources to remove barriers	The customer picks a small story to add to the iteration
More development, as above	
Iteration 3	
Iteration 4	The customer takes the results of this iteration and demonstrates them to a much broader group of stakeholders, including project sponsors. They provide high-level feedback to the customer.

When can I use Extreme Programming, and what do I need?

When a business is understood very well, and the requirements for a system to support the business are also well understood, then a predictive software development methodology will work well. Unfortunately, most of the systems we build today don't fit into this category. Today's business environment is incredibly dynamic. Even if requirements are known at the start of a project, they're certain to have changed by the end. To make matters worse, as Tom DeMarco has observed, the systems we build today are complicated, because we've built all the simple ones already.

Team Size

Extreme Programming "out-of-the-box" is ideally suited to groups of less than 15 people. Psychologists define a group of up to 15 as our "sympathy group"[22]. This is the number of people that we can focus on in detail without having anyone suffer from a lack of attention. Once a group gets bigger than this you need to change some of the XP practices to maintain good communication, and no matter what you do, things will always be less efficient than they were in the smaller team.

Communication

XP teams also need to be able to communicate openly and honestly. Team members need to respect one another, and be able to check their egos at the door. Kent Beck says "If members of a team don't care about each other and what they are doing, XP is doomed"[23]. The best results come when the team works in an open, honest environment However, we've seen XP work in situations where the team was open and honest, but normal (dishonest) corporate culture operated outside the team. In this situation the team needs to be buffered from their external environment, and the manager, the customer and the coach get a few more challenges. If it's simply impossible for you to create an environment where openness, honesty and respect are valued, then XP certainly isn't for you. Just in case you think openness and honesty are the norm, here's a story from Dale Emery[24] that may sound familiar:

> "A small division of large company had worked with a partner, a small company, for several years. They decided to form a new company by splitting off the division and merging it with the partner.
>
> A colleague and I were invited to observe a three-day planning session, at which 20 people from the not-yet-formed company would plan their inaugural project. Somehow (without planning) they had gotten the idea that the project would take 9-12 months.
>
> At lunch the first day, my colleague and I ate at separate tables, so we could meet more people. As I was sitting down at the table, four people started talking to me at once. "But you don't understand! There's no way we're going to be able to do this in a year! We have all kinds of issues that nobody is talking about!"
>
> As I caught my breath, my colleague came running over, and said, "You won't believe what's going on at my table. People are going nuts, and bringing up all kinds

[22] "The Tipping Point", Malcom Gladwell, Abacus, 2001
[23] "Extreme Programming Explained", ibid
[24] Posting to extremeProgramming@yahoogroups.com, Dale Emery, reprinted by permission.

of issues!"

As each person brought up a new issue, my colleague and I encouraged them to bring up the issues during the meeting. People were very reluctant to do that, and kept saying, "We could never tell the managers this stuff!"

So we talked to the managers, telling them that people seem to have some issues that they are afraid to speak about. We worked out a way to get the issues out into the open.

That night, we arranged an extra session. We gave each person a few index cards, and asked each to write down any important issues they have been reluctant to raise. (We asked that anyone who didn't have any issues should write, "I have no issues" a few times, so that nobody could tell who wrote an issue and who didn't.) Then we read the issues to the group.

There were some very sticky issues, but once each issue was out in the open, the group was perfectly willing to acknowledge it, and even to address most of them.

So the second day of planning seemed more real. People acknowledged the issues, and factored that into their plans and estimates. It became clear that the project was going to take more than 12 months. The most optimistic estimate said 18 months.

The managers said, "Yes, but what are we going to tell the CEO?"

My colleague and I (along with the developers) encouraged the managers to be honest with the CEO, to give estimates they believed in. They said, "But we could never tell him what we really believe!"

On the afternoon of the last day, the CEO arrived. The managers laid out the plan, bucked up their courage, and said that the most optimistic estimate was 18 months.

The CEO said, "Yes, but what am I going to tell the investors?"

[So the answer to your question is:] People make crazy demands because they are afraid of someone above them who is making crazy demands. I'm not sure who the investors are afraid of.

Postscript: The CEO told the investors they'd ship in nine months. The investors handed over the money, and chartered the new company. Within two months, the developers (and a few managers) abandoned ship and got jobs elsewhere. Also within two months, the top four execs of the new company cashed in their stock options and retired."

Customer Availability

You need to have a customer as part of the team. XP teams don't expect the customer to spend a lot of time writing copious requirements documents, but they do expect the customer to be available to answer questions all of the time. XP projects not only give the customer the opportunity to steer the project, they **require** the customer to steer. Every modern software development project works in a constrained environment, which means that somewhere or other trade-offs need to be made. XP says that the trade-offs need to be made in the scope of the project, and it's essential that the customer make these trade-offs, not the developers. Many teams report that with XP the programmers are no longer the bottleneck, but the customers feel a bit more pressure.

Of course, not every team is able to get a customer actually co-located with the team, and in some situations it isn't appropriate. The important thing is that there be someone who is able to speak from the customer perspective, that they always be available to make trade-off decisions, and that they are the person who is responsible for the success or failure of the project. Although customers will definitely have more work to do than they did in a traditional project, they won't necessarily be required 40 hours a week every week. Some customers relocate to be with the team, but bring their old work with them, at a lower intensity (this is ideal). Some customers stay in their current location, but are always available by phone/fax/e-mail/video conference (in this case it's important for the customer to have spent some time with the team before they go back to their original location, so that they still feel like part of the XP team to everyone).

When it's not appropriate to have an actual customer with the XP team, some people talk about having a "customer proxy". This is someone who will answer questions and make trade-offs as if they were the customer, but won't make them quite as quickly or accurately as an actual customer (usually because they need to consult with an actual customer).

Technical Environment

You need a technical environment that supports and encourages small changes. Most modern development tools fit into this category. If your technical environment mandates a 2 hour compile and link cycle before you can run a test, then XP isn't for you. Actually, it's surprising that you can get anything done with that sort of environment, so you should probably throw it out and get something more flexible anyway.

Physical Environment

Since XP teams require effective communication to thrive and survive, it helps to have a physical environment that supports, and sometimes enforces, close communication. Alistair Cockburn has written a lot about patterns of communication in agile software teams[25], but a good starting point is to sit everyone close together, in an open environment that the team can rearrange themselves. Don't keep customer and the developers on separate floors, and don't spread the developers across multiple cities if you can avoid it.

Pair programming works best when the pair can sit side by side, either sharing a mouse, keyboard and screen or having one each connected to the same computer. If the pair need to change seats to swap roles then you should change the physical environment. Some teams have noticed a benefit from switching from large CRT monitors to flat screen monitors, because the flat screens can easily be moved around, and fit in more locations than the CRTs.

XP teams also do lots of informal design, so it's good to have lots of spaces for them to draw pictures and share them. Whiteboards are good for this, and also provide places for the team to record information that needs to be shared for a short period of time, then discarded (an iteration plan is a good example of this).

[25] "Agile Software Development", Alistair Cockburn, Addison Wesley, 2002

Coach

Finally, you need to have someone who believes in the process, has enough technical knowledge to be respected by the programmers, and enough people skills to be able to manage the frictions that arise when people are placed in a truly collaborative environment for the first time. Make this person your coach. We’ve observed that while someone with this range of skills makes an excellent coach, you can also distribute the coach role. Our experience is that excellent coaches provide at least 8 different services to an XP team:

Keeping the team on process

Every team is prone to drift from its agreed process from time to time. When a team is new to XP there’s an enormous temptation to drift back to old habits, and although all the team members have some responsibility for keeping each other on track, it’s also very useful to have someone sitting a little outside the pressure of daily delivery, capable of telling people to slow down and get back on the agreed process, so they can go faster later.

Providing design support

Most coaches are experienced software developers, and frequently they have lots of experience in object oriented development and good design skills. Team members often draw on this skill and experience and use the coach as a de facto architect and design reviewer.

Providing process technical advice

Most programmers who move to XP find that they need to acquire new skills – social skills such as pair programming, but also technical skills such as writing effective tests, learning to detect code smells, and applying effective refactorings. New XP teams need to have someone who can provide expertise in these areas so that the programmers don’t need to learn everything from scratch.

Diagnosing process problems

Although XP is a very good starting place for a team moving to agile development, most teams will find that there are places where it doesn’t quite fit properly, and needs to be modified. However, it can be difficult to judge whether the process is fitting poorly, or whether the team members aren’t picking up the skills they need to use it effectively, and it’s especially hard to tell the difference when the team is just beginning to move to XP. An experienced coach has seen these transitions before, and is better placed than most of the team members to diagnose the malady behind the symptom. An experienced coach has also tried different remedies in the past, and can provide good advice on how to adjust the process or provide skills training and mentoring as appropriate.

Smoothing interpersonal friction

Regardless of how relaxed and cooperative your team members are, they are going to come into conflict at some point. Although every team activity, from sports to programming, requires both technical and social skills, programming training to date has emphasised technical skills over social skills. This means that many programmers lack the social skills to handle conflict effectively. Many programmers, accustomed to a hierarchical environment with a clear team leader, will be more comfortable taking their conflicts to a third party for resolution.

An effective coach will do everything possible to help the team members resolve conflict themselves – the worst thing a coach can do is force a resolution, since this makes the team members more dependent on the coach for future conflict resolution, and undermines the message of shared responsibility. Fortunately, social skills can be acquired just as technical skills can – social skills training would certainly help most programmers be more effective members of a team.

Buffering the team from outside pressures

One of the strengths of extreme programming is that bringing a customer into the team lets the programmers focus on programming, rather than learning the business at a detailed level. Focussing on the programming helps produce a more effective system more quickly. Some teams are buffeted by external forces – requests for information, required attendance at meetings with no clear purpose, changing priorities. A coach can help a team by buffering it from these outside pressures – this sort of buffering is essential when the team are first learning the XP practices. Over time a coach should expose more of this buffeting to the team, and let them decide how to handle it themselves.

Coaching the customer, quality assurance and other non-programming resources

The XP literature makes it clear that the programmers need to have a coach, since the job of the programmer changes significantly when they transition to XP. However, there’s very little discussion about the relationship between the coach and other members of the team. It’s fair to say that everyone’s job changes when they become part of an XP team, and all of them benefit from coaching.

For example, customers may be accustomed to specifying all the requirements at the beginning of the project and then not being involved again until the end of the project. They may expect the programming team to resolve any ambiguities or conflicts within the requirements statement. In an XP project the customer doesn’t have as much work to do up-front, but they have to make a lot more decisions as the project proceeds, often involving more detailed specification than they’re accustomed to, and often requiring them to resolve conflicts between different stakeholders. It’s useful to have a coach to provide guidance on how to accomplish this, and how to balance work and decision making between programmers and customers.

Quality assurance is in almost the opposite situation. Instead of waiting until the end of the project to take delivery of the application and report defects, they now have an opportunity to be involved in the creation of the testing suites, provide recommendations on testing and testability, and provide feedback on every, frequent, release. A good coach helps to engage the quality assurance department in the project as early as possible.

Providing resources

One thing that a team often can’t do for itself is provide additional resources. These resources may be things as simple as new desks or computers, but they might be additional staff, whiteboards so that the team can do more low-cost design work, or extra soft drinks for the fridge. A good coach can act as an advocate on behalf of the team with management to help get these resources, though it's even better if management is involved in the regular team meetings (retrospectives, the planing game and/or stand up meetings). In the Scrum methodology this role is referred to as the Scrum Master. Many people have noted that developers work better when the coach is good at providing toys and food!

When does Extreme Programming fail?

Fundamentally, XP will fail if you can't satisfy the prerequisites we described in the previous section. However, some failure modes are more common than others.

Using the wrong people

Software development using XP is very team-centric. Like any team activity, it requires both technical skills and social skills. Most traditional software development approaches, and our education system, emphasise technical skills over (and almost to the exclusion of) social skills. XP teams don't work very well if the team members have very poor social skills. In particular, XP team members need to be reasonable verbal communicators, and need to be able to share with others – not just toys, but code and designs as well.

Having the wrong customer

As we've already noted, the customer is responsible for the success or failure of the project. It's important that they be able to make the appropriate trade-offs to steer the project to a successful conclusion. Some people have interpreted this as meaning that they must be a senior business person – certainly this would be useful, but having a junior business person who spent a lot of time networking and coordinating with more experienced business people would also work.

Poor communication

This doesn't just mean using people with poor communication skills, as we mentioned in the previous section. It also means trying to do XP in an environment that hampers or discourages communication. A simple example is physical environment – if the team is spread across multiple floors, or even cities, then it's harder for them to communicate effectively, regardless of their personal skills. Also, as we've noted before, some corporate environments discourage open, honest communication.

Being XP'd on

The technical practices that comprise XP weren't selected randomly – some practices have deficiencies that are corrected by other, supporting practices. While it's ok to adopt single practices (more on this in the section on adopting XP), there are practical limits to this. Some teams try to adopt just the practices (or even aspects of practices) that they like, and then say they're doing XP – for example, adopting the principle of travelling light and reducing documentation, but not supporting the system with automated tests can be a disaster[26]. Practices need to be selected and adapted judiciously.

[26] "Are You Getting XP'd On", Kent Beck, 2002, http://groups.yahoo.com/group/extremeprogramming/files/Are%20You%20Getting%20XPd%20On.pdf

What does an Extreme Programming project look like from the outside?

From the outside, an XP project can look very different to a traditional project. First, your XP team won't request a voluminous requirements document – the XP team will want enough details to put a scope on the project, but they'll discourage the customer from trying to predict the precise requirements in detail. Both the programmers and the customers understand that the details will be fleshed out over the course of the project.

Once development work starts, the programmers won't be delivering a substantial design or architecture document. Instead, they'll start by delivering working software – often a thin vertical slice through of the application that demonstrates feasibility, but is also production quality. When there are significant technical risks the programmers may also perform some technical spikes – concrete experiments that confirm they have a reasonable solution to each of the risks.

As the project proceeds, the team will work at a consistent pace. At first it may look like they're going slower than you expected, but then they'll start to look like they're getting faster and faster.

There'll be a lot of chatter. Programmers will be working in pairs, talking about what they're doing, investigating different options. Pairs will be leaning over asking one another questions, and asking the customer a lot of questions. The customer may feel like they're under a bit of pressure.

The team won't be working overtime. Instead they'll be putting in very full days, working hard as a team, then going home and coming back refreshed. The programmers will understand that overtime lowers code quality and increases the cost of change. They may work overtime one week, but never two in a row. Instead, the team will work together to change the scope to meet the other project constraints.

The customer's confidence will increase with each iteration, as the team consistently delivers new functionality and increased value to the business. The programmers will get more confident because they are reaching closure at the end of each iteration, rather than letting tension simply build until the end of the project, and they can see that their commitments are based on historical fact. Managers will become more confident, because they're not exposed to the loss of any particular team member, since everyone on the team understands what's being done and how it's being done.

Leaders will still be leaders, but decisions that used to be made by authority figures will now be distributed across the team. Particularly early in the project, there will be a lot of team meetings as the team encounters architectural issues and looks for common understanding and common solutions.

Rather than the team reporting what the new completion date will be, they'll be telling you how much can be done by the predetermined completion date. The customer will be selecting features so that the business value of the system is maximised on the date.

Planning an XP project

The first step in planning an XP project, as with any other project, is to determine the scope. In XP this means listing all the stories. A story is a narrative description of some piece of functionality that will be of value to the user. Conceptually it is like a use case, but it has a lot less detail than a use case, because the team expects that the details of the story will change as the application is developed, so there's no point into going into the details right now. A story has been characterised as a "promissory note for a future conversation" – this means that the story records that something needs to be done, but everyone on the team understands that the customer will need to talk to the developers to give them the details before the story is implemented.

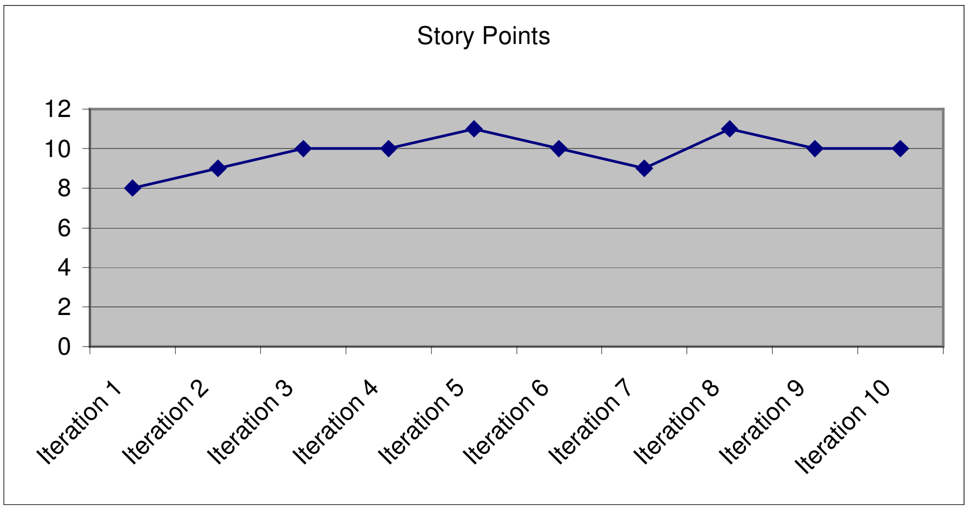
So how much detail do you need in a story? A story needs to have enough details for the developers to put an estimate against it. If a story is too big or too vague to be estimated, then it needs to be broken down into smaller stories, and this process needs to be repeated until the developers are willing to put estimates against all the stories. The actual size of the estimate isn't important at this stage – some stories may take one developer a single day to implement, some stories may take the whole team weeks – the important thing is that there is enough shared understanding between the customer and the developers for the developers to assign estimates.

Stories can be estimated in any units that you like, because in the end all we're interested in is the relative size of the stories. Some people pick some small story and arbitrarily assign it a value of "1", then estimate all the other stories relative to that one. Some people find this very difficult, and perform their initial estimates in person days or person weeks, then just drop the units to give them dimensionless estimates before they move on.

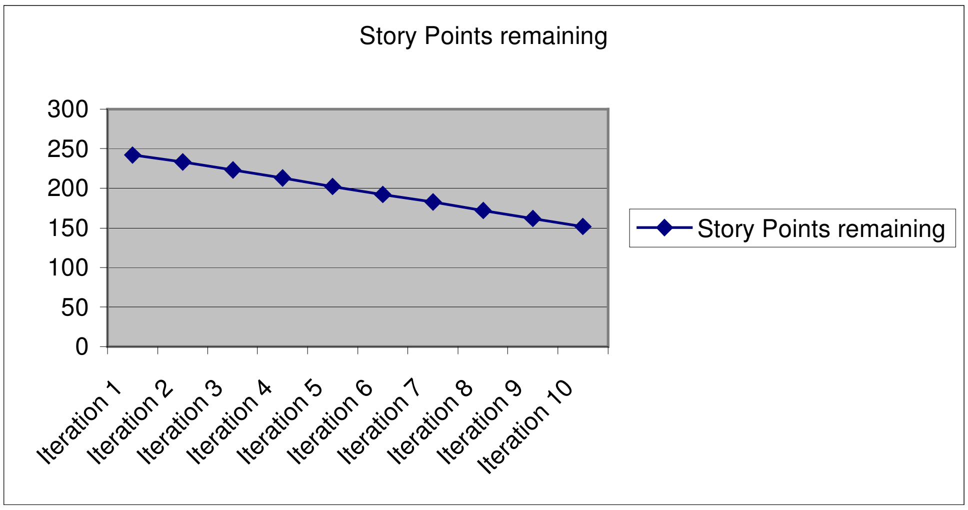
At the end of the initial scoping phase the team has a list of stories, and the total size of the project in story points. What they don't know is how long it will take this particular team, with this particular technology, in this particular environment, to implement the stories. The team can guess based on their previous experience, but this guess needs to be validated as soon as possible. The way that an XP team validates the guess is to decide on an iteration size (commonly one week), choose some stories, and implement them. The number of story points that the team gets through in an iteration is their "velocity", and the number of iterations required to implement all the stories in the list is simply the total number of story points divided by the velocity.

Choosing the correct stories for the first iteration is important. The first iteration should build a vertical slice of the application, exercising at least part of each of the obvious subsystems. For example, a typical business application's first iteration should require user interface components, business logic and database access. In general stories should be selected by the customer based on business value, but there also needs to be a dialogue between the customer and the developers so that technical risks are addressed early. One simple way to communicate risks to the customer is to put banded estimates (eg. 3-15 story points) against stories that have high technical risk. This makes risks obvious to the customer and will often encourage them to choose to address these risks early to solidify the estimates.

Once you've started working through the iterations, how should you track progress? To remain sympathetic to the XP principles, and to avoid distracting the developers with administrative tasks, you want to keep project tracking as simple as possible. The simplest thing to track, and something that every XP project should track, is velocity from iteration to iteration.

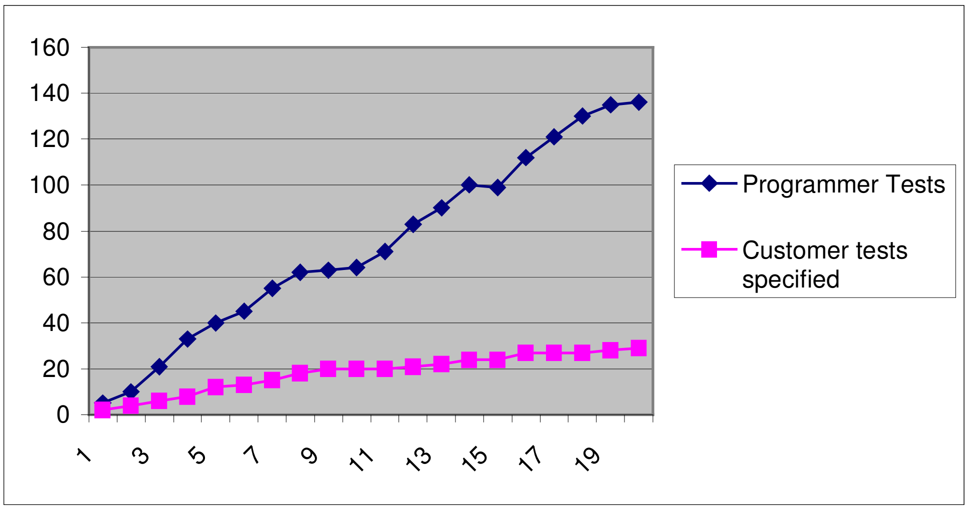


A stable XP project should demonstrate consistent or improving velocity. A blip on the velocity graph should be investigated, and a gradual deterioration in velocity points to some problem (often lack of refactoring, or some other code quality issue).

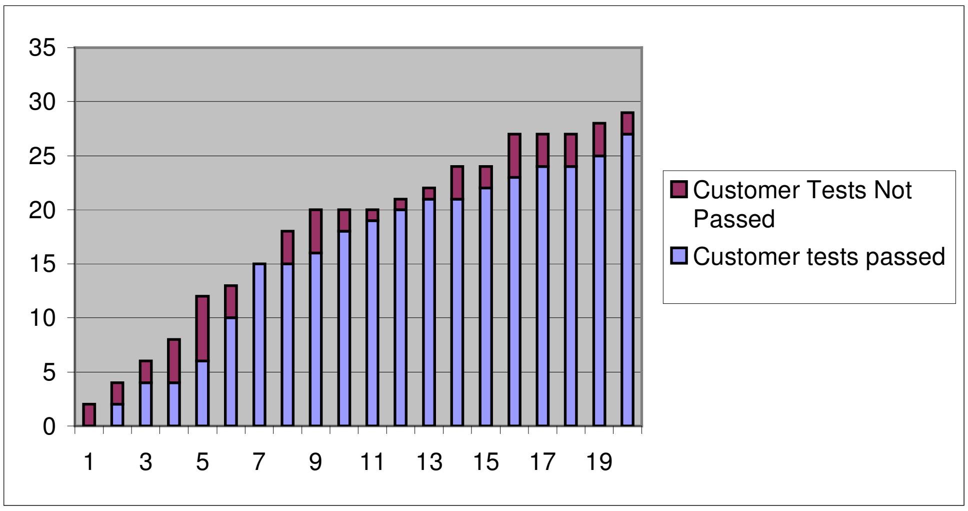


A closely related value to track is the number of story points in remaining stories. This should trend downwards over the life of the project, affected by stories completed by the developers, but also by stories added or removed by the customer. This information can be used to derive an estimated completion date, or to estimate the number of story points that will be completed by a fixed date.

Both these measures are really size metrics, in that they give us an idea of how much of the project is done and how fast it was done. Fluctuations can tell us that we need to examine the project more closely, but they don't tell us much about the internal state of the development. The internal health of the development can often be related to the level of testing, and there are two simple testing metrics we can track.



The first is simply the number of tests (both programmer and customer tests). We can track this from iteration to iteration, or at a more fine grained level.



This graph shows us how many tests are being written by programmers, and how many tests are being specified. We can assume that all programmer tests are being run successfully many times a day as part of continuous integration, but different projects have different approaches to customer tests. On this project, let's assume that customer tests are run as part of each integration, but aren't necessarily expected to all run successfully. In that case, we might like to know how many customer tests there are, and how many are being run successfully.

These are just examples of metrics that might be tracked during an XP project. It's important to select the metrics that are important to you, especially ones that reinforce particular lessons of XP or diagnose problems, and to make the metrics as visible as possible. Once a metric isn't interesting any more (which may be because it isn't changing), then stop tracking it. For example, we've worked on projects where we tracked the number of programmer tests on a daily basis through the first few iterations to encourage developers to write tests, and to reinforce how important programmer tests were to an XP project. After the first few iterations we stopped tracking this formally (though developers could still see how many tests were executed in each of their own test runs).

Introducing Extreme Programming

Total Immersion

The ideal way to introduce XP is to take a small group, say 2 to 4 people, introduce them to all the XP practices (possibly through an immersion course) and then have them execute some project or part of a project using all the XP practices together. This very quickly gives them a feel for the way the XP practices interact in a mutually supportive web. With this approach it's very important to keep the team small – during the learning phase people will misinterpret or misunderstand some practices and conventions, and as a result the team will need to change direction quickly to recover. It's much easier to correct behaviours in a small group of people than in a large group of people, and the cost of recovery is much lower. Once the small group has had a chance to make the practices habits, build some XP infrastructure (eg. a continuous build process), find solutions to the most common problems presented by their environment, and get experience working together, then more people can be added to the team and shown the ropes by the existing team members.

One Practice At A Time?

Some teams aren't able to adopt all the practices at once. There can be a lot of different reasons for this – lack of management support for all practices and resistance to some practices within the team are common reasons. It's quite feasible to introduce one XP practice at a time, focussing on the practices that address the specific problems of your environment first. However, there's a synergy to be had from all the practices that's missing from this approach.

A common question is, all other things being equal, what order would you introduce the practices in? A year ago we might have said that either test-driven development was probably the first practice to introduce, followed by refactoring, but our experience since then is that pair-programming makes introducing both test-driven development and refactoring much smoother by reducing swings in the direction of either too much or too little of each, and that learning new practices in pairs is more beneficial than learning solo, because there's always someone to discuss an approach with, to lead when the other person doesn't have a clue, and to flag a stop when the other person is pursuing some unfruitful avenue. So now we recommend introducing pair programming before test-driven development or refactoring, even though it's technically not essential.

We've produced a guide to constraints we perceive in introducing practices, however we need to emphasise that this is based on our experiences, rather than set in stone. We also split out Programmers Tests as a particular aspect of Test-Driven Development. Although Test-Driven Development requires Programmers Tests, you can introduce Programmer Tests without introducing Test-Driven Development.

Practice	Dependencies
Coding Standards, Continuous Integration, Customer Tests, Pair Programming, Programmer Tests, The Planning Game, Sustainable Pace, Whole Team	Can be introduced independently
Collective Ownership	Benefits from Programmer Tests
Design Improvement	Benefits from Pair Programming and both Programmer and Customers Tests
Simple Design	Benefits from Design Improvement, Metaphor
Small Releases	Benefits from Continuous Integration
Test Driven Development	Benefits from Pair Programming, Programmer Tests, Whole Team and Metaphor

Given these constraints, find the place in your development process that's causing the most pain, and select the practice or practices that will help relieve the pain.

Another question that comes up is, if I'm doing only some of the XP practices, am I doing XP? Our first answer is usually, "why do you care?". There's no prize for doing XP – the prize is for finding a set of practices that work well for your team and organisation, and the form it takes is much higher productivity and customer satisfaction. The hard-line answer is that you're doing XP if you were using all twelve XP practices at some point, even if you subsequently changed them. If you've never done all twelve practices together, then you're not doing XP.

Using an Experienced Coach

Although it's quite possible to familiarise yourself with XP and introduce the method on your own, most teams will benefit from having an experienced coach. An experienced coach has seen many different flavours of XP (or other agile methods), can help set up your infrastructure, provide technical support, and help advise on customising the project.

At the moment only a few Australian companies can provide agile methods coaching and mentoring. Two of these are Object Consulting (http://www.oopl.com.au) and Khatovar Technology (http://www.khatovartech.com).

Existing Roles Within an XP Project

So far our discussion of XP has only identified a limited number of roles – customer, developer and coach. Traditional projects and teams typically have a wider range of roles. In an XP team most of these roles are subsumed under "developer", with developers simply signing up for tasks based on preference and ability, but some existing roles are worth discussing in more detail.

Architects

Many traditional project teams will include a technical architect, or will use the services of a technical architect provided by some other department of the organisation. The architect is responsible for describing the application at a high level, and providing direction to the programmers in terms of subsystems, core classes, or some other high level constructs. To provide this information, the architect needs to have access to the complete, detailed application requirements.

In an XP team the emphasis is on shared knowledge and responsibility, rather than taking direction from a single (particularly if external) source. Also, the complete application requirements aren't available in detail at the start of an XP project. This means that the architect role needs to change. XP architects need to work with the team at the beginning of the project to outline a tentative technical architecture (or metaphor), and help the team members obtain the knowledge and expertise they need to extend and modify the architecture (metaphor) over the course of the project. Ideally this means working as one of the developers and sharing experience and skill through pairing.

Business Analysts

Traditionally, business analysts combine knowledge of the business domain with knowledge of the technical environment and constraints, without themselves being either customers or developers. Programming teams frequently take direction on business issues from the business analysts.
There are two ways to adapt this model to an XP environment. The simplest way, though probably not the most effective, is to use the business analysts as the XP customer. The reason this approach may be sub-optimal is that, although the business analysts have a good understanding of the business, they may lack the detailed, leading-edge knowledge that would let them steer the project to a truly superior solution.

The other model is to use the business analysts as coaches for the customers, encouraging them to add requirements that might not be considered by a customer who wasn't familiar with the technology, and facilitating communication between the customers and the developers. In this model business analysts may be able to handle more than one project simultaneously.

Quality Assurance

In a traditional team, quality assurance receive a supposedly complete, defect-free application, then find all the defects that were missed by the developers. Frequently the quality assurance schedule has been compressed to accommodate development overruns, and both quality assurance and development are under enormous pressure to rectify (or ignore) defects found by QA.

In an XP project, quality assurance staff become members of the development team, along with the customer, the developers, and any other participants that are required. Quality assurance works with the customers and developers throughout the project, suggesting features that would improve the testability of the application, and working with both the customers and developers to specify and execute tests iteratively. This spreads the QA effort across the life of the development, so that almost nothing remains to be done when development is completed.

Fitting XP to Common Commercial Models

Time and Materials Contracts

Extreme Programming is perfectly suited to a time and materials contract. The vendor should do enough design work to let them tender a quote, and offer the customer the opportunity to steer the project by adjusting requirements at the end of each iteration. If the customer stays uninvolved, the project will benefit from the appointment of a proxy customer – frequently the project manager – whom the developers treat as the customer for all intents and purposes. This lets the project manager leverage all the advantages of an agile methodology, even without the direct involvement of a customer.

In this situation we suggest that the development team construct a release plan and provide releases to the real customer as frequently as possible. Once a customer has seen the quality of these releases, and has also seen evidence that their feedback is quickly and effectively incorporated into working software, then they may become more involved in the project and the project manager can relinquish some of the customer role.

Fixed Price Contracts

Clearly there's less synergy between XP and fixed price contract arrangements, since the fixed price contract arrangement is usually fixed scope as well. An XP team tendering for a fixed price contract should take the same approach before tendering that they do now – in particular, they should do enough design work that they are confident in their quote. However, they can add value to the quote by offering the customer the alternative of adjusting functionality at interim points, and move to a more time and materials basis. Kent Beck and Dave Cleal have discussed one model of this approach[27].

Should the customer choose not to take advantage of this opportunity, then substitute your own proxy customer. One of the characteristics of the XP customer is that they should be responsible for the success or failure of the project, so in a fixed price environment, the project manager is again a logical choice for the proxy customer. So long as everything is going as hoped, the proxy customer/project manager doesn't need to make any difficult decisions, but if resources start to get stretched, then they're the appropriate person to decide how to adjust scope, and how to explain the situation to the real customer. Regardless, the development team gets all the advantages of XP, and the project manager gets regular releases they can demonstrate to the real customer.

Common misconceptions about Extreme Programming

The XP practices are new, revolutionary and untried

All the XP practices have an established pedigree. Lots of software teams have used individual XP practices to help them solve their development problems. For example, pair programming is often

[27] "Optional Scope Contracts", Kent Beck and Dave Cleal, http://www.xprogramming.com/ftp/Optional+scope+contracts.pdf

considered one of the most radical XP practices, but Larry Constantine documented the use of "Dynamic Duos" at Whitesmiths in the early 1990s[28] and Jim Coplien included the pattern "Developing in Pairs" in his organisational pattern language[29]. What's new about XP is the combination of the practices in a mutually supporting web, and emphasising that practices should be done all the time.

All the XP practices have already been tried – there's nothing new in XP

The first part of this is definitely true – none of the XP practices are new. However, a number of the XP practices have flaws when they're used in isolation. XP advocates a set of practices that work well as a group, overcoming issues that might have been experienced using individual practices in the past. XP also emphasises using all the practices all the time – a significant departure from previous use of these practices.

XP ignores gathering requirements, and you can't document them with XP

Requirements are central to an XP project, but an XP project doesn't expect the requirements to be as detailed as they are for a traditional project.

In a traditional project requirements need to be detailed so that the project plan can be accurate, but in real life, no one has an accurate project plan anyway. We all understand that, but XP makes it explicit.

In XP there's no design step, so there won't be any design

Traditional methodologies usually divide development into three major phases – analysis, design and coding. The implicit assumption is that all design can be successfully completed before any coding is attempted, and that coding can be completed without any resort to further design. XP makes quite a different assumption – that design and coding need to be intimately interlaced. This means that in an XP project there isn't a distinct "design step", so many people feel that there isn't going to be any design at all.

XP projects encourage everyone to be designing all the time. The design process should use whatever tools are available and comfortable to the developers – white boards, CRC cards, paper, and code. There are really only two constraints imposed on XP developers: 1) if there's any contention between the designers, it should be resolved by writing code, not arguing over other design artefacts; and 2) don't spend time designing for features that aren't required yet (among XP'ers this is known as YAGNI – you ain't gonna need it).

When some developers hear that they shouldn't design for features that aren't required yet, they start to write large objects with many methods, because this is all they need "right now", and because it's "simple". This approach is based on misunderstandings of what XP practitioners mean by "simple", and the role of discrete objects in a object-oriented design.

XP leans heavily on the idea of "simple", but in the sense that a simple design is one that is very easy to understand and to use. Good object-oriented designs accomplish this, at least in part, by having distinct concepts represented by distinct objects – objects that do one thing and do it well. Separating large objects into compositions of smaller objects isn't designing for the future, it's designing for the present.

[28] "The Benefits of Visibility", Larry Constantine, Computer Language Magazine, Volume 9, #2, 1992
[29] "A Generative Development-Process Pattern Language", James O. Coplien, Pattern Languages of Program Design 1, Addison Wesley Longman, 1995

Design contention should be resolved in code because this is ultimately the only place that counts – if you can't code the design, then it's not much use. It's also the best place to resolve contention because it's the only place that forces the designers to confront all the details that need to be addressed to turn the design into something that can be run.

Designing for features that aren't required yet on an XP project is just a time sink. Traditional projects do this because their cost of change is very high, which makes it important to make the implementation as complete as possible the first time. XP projects work hard to keep the cost of change low, so that there's no real difference between implementing something now and implementing it later, when you know exactly what you need.

XP says there shouldn't be any design

As mentioned before, XP encourages continuous design, not no design. However, a big difference between XP and traditional methodologies is their approach to contention between the design and the code.

Alistair Cockburn has observed that "With design I can think very fast, but my thinking is full of little holes"[30]. This frequently means that there are problems with a design that are only apparent to the people who try to implement it. Traditional methodologies encourage programmers to stick with the design, which has been approved and signed-off on by many people, warts and all. XP encourages the programmers to change the design immediately to reflect the realities of the implementation environment. In this environment designers are also rewarded for designing in small, incremental steps, each step based on the validation received through the implementation of the previous step.

No architect or Big Design Up Front (BDUF) means the results will be chaotic

Messes usually come about through lack of communication – different people doing different things at different times (or even the same person doing different things at different times) rather than working cohesively. Traditional projects try to address this by creating an architecture document that describes the overall structure of the application before any coding begins. This architecture document is a central mechanism for maintaining consistency among developers who otherwise work alone.

XP teams use a number of different practices towards the same end. First, the practice of using ubiquitous language, or a project metaphor, helps ensure that everyone is working in the same conceptual framework. Stand-up meetings help to maintain the flow of information across the team, and pair-programming helps maintain

XP's informal, often verbal, communication of architectures and design occurs most effectively when there is a high amount of compression – when a few words can be used to convey a lot of information. It's facilitated by programming languages which exhibit high compression, and hindered by languages which are verbose or difficult to read in other ways.

Ubiquitous language, metaphor and design patterns are all mechanisms for increasing compression. So are diagrams and design models. Teams that work with verbose programming languages, and lack a clear metaphor, will often find themselves using more diagrams and design models to compensate. Some teams combine all three approaches by representing their metaphor language as a high level design on a whiteboard, the objects in the design becoming the elements of the ubiquitous language

[30] quoted by Martin Fowler, "Refactoring: Improving the design of existing code", ibid, p67

consistency. All of these can be complemented by diagrams and writings as the team sees fit, but the team drives the architecture, which often emerges incrementally, rather than the architecture driving the team.

XP doesn't address technical risks

Some people feel that since the customer chooses the functionality that will be added in each iteration, there's no opportunity for developers to address technical risks. This can be addressed in a few different ways.

First, remember that XP stories need to be detailed enough for developers to assign estimates to them. If there's enormous technical risk, then it's unlikely that the developers will be able to assign an estimate to one or more stories, and there'll need to be a discussion with the customer. There could be a number of different outcomes – the story could be discarded, the customer may settle for either a single large estimate or banded estimate that reflects the uncertainty, or the developers may agree to immediately do a technical spike to give them enough information to assign an estimate to the story. A technical spike is some concrete development activity that confirms that a task is possible, and gives the developers enough information to provide estimates on the related stories. A spike should be the simplest work that meets these objectives, and at the end of the spike any resulting code should be discarded.

Also, remember that although the customer is responsible for the final choice of stories in an iteration, there's an ongoing dialog with the developers that helps them make these decisions. It's just that one of the things they can't do is change estimates made by developers (and nor can developers work on stories that haven't been agreed with the customer). Most customers are quite willing to work with developers to ensure that technical risks are addressed early in the project, especially since it's the customer that's responsible for the success or failure of the project.

Developers will hate it

Change is difficult, regardless of the type of change. If you move to an agile development approach, it's unlikely that all the developers will be just as content as they were before. Some developers will be much happier, some will be less happy. Some people will want to move on to another team, and some people will want to join the team because they are attracted to agile development approaches. We've seen all these things happen.

Most developers find that using XP helps them build a stable, reliable system in a more predictable way, and that it helps them meet the expectations of the customers. Overall, they have a lot less stress and solve a lot more business problems, and this is extremely satisfying.

Experienced XP coaches work hard to try to make everyone on a team as comfortable as possible, and to adapt the practices to local conditions, including the personalities and preferences of the developers. Frequently everyone on the team can be accommodated, but sometimes they just can't. Particular challenges are:

> **Pair Programming** – almost everyone finds pair programming challenging to begin with. Some people, particularly people who were attracted to programming because of the long periods of solo activity that it offered, never adapt. We've given up on trying to predict who will and won't adapt in advance.

Satisfaction – programmers frequently get satisfaction from being able to point at a particular subsystem and saying “I wrote that”. When a team moves to a collective code ownership model developers need to get more satisfaction from the overall growth of the application rather than from specific subsystems.

Customers will hate it

Some people feel that customers will be less satisfied with an XP project than they are with a traditional project, predominantly because of the lack of formal specifications. Most customers find that although their workload goes up, and they need to be more involved with the team, these effects are more than offset by the additional control that they get over the development of the application.

XP is hacking

Some people interpret the lack of formal requirement and design documents in an XP project as a license for the developers to simply hack out some low quality solution, and consequently they don’t treat XP as a serious software development approach.

Projects that implement all the XP practices are quite different to this. Having a customer involved with the team all the time, working side by side with the developers, increases the developers’ commitment to getting the customer what they need. Regularly delivering working software to the customer improves accountability and makes progress (or lack of it) highly visible. Pair-programming and stand-up meetings improve visibility and accountability within the developer community. Overall, XP teams are more dedicated to producing a high-quality product than traditional teams, where quality is often the control variable.

XP says there shouldn't be any documentation

XP says that teams should “travel light” – that they should only produce the documentation they need, and nothing more. This is certainly not the same as not producing any documentation at all!

> If your developers say that doing XP means doing just what they’re doing now, but without producing any documentation, don’t believe them. Kent Beck has written a paper on this situation – “Are You Being XP’d On?”.

In an XP team, some traditional documents are replaced by executable test cases, and some are replaced by conversations. Requirements are conveyed to the developers first by conversation, and are then “hardened” by recording the requirements in acceptance tests. Subsystem or object-level documentation is mostly replaced by executable unit tests. Lots of information that might have previously been contained in memos is communicated at stand-up meetings and during the planning game.

However, there are clearly limits to this style of communication, and when the team feels it’s appropriate they should produce some documentation. Things that help the developers go faster are at their discretion, everything else is recorded as a story and prioritized by the customer so that the trade-off between documentation and other work is made clear. An XP team doesn’t produce documentation simply because the process says they should.

Pair programming must take twice as long

Pair programming is the XP practice that seems to generate the most heated resistance, but it's also the practice that has the most non-anecdotal evidence behind it. The simplest way to state the research conclusion is that sometimes pair-programming saves a lot of time overall, and even when it doesn't, it produces a much higher level of quality in the same time as solo development[31]. Also, remember that "programming isn't typing"[32].

Developing unit-tests is a waste of time/money

In an XP project, a significant proportion of the time spent writing unit-tests is design time, determining what the interface should look like, how the objects should interact, and what boundary conditions need to be considered. Another proportion of the time can be considered documentation, writing tests that show how the object should be used. Only a small portion of the time can be purely allocated to writing repeatable tests, and this time is more than offset by the reduced cost of change and increased flexibility that results.

XP is easy

We wish that this were true. XP is satisfying, productive, fun, challenging, but most of the time it isn't easy. Learning XP means acquiring new habits – one of the hardest things that people do. Some XP practices are almost the opposite of the practices that developers learned at university and have previously used in industry (eg. test-driven development versus big design up-front), and switching from one to the other requires a lot of focus and dedication. This is one of the reasons that most people suggest using an experienced coach to help a team transition to XP.

Extreme Programming in Australia and New Zealand

A number of Australian companies have already used XP successfully. The ones that have made their experiences public in one way or another include:

- Citect
- Compac Sorting Equipment
- Eagle Datamation International
- Hubbub
- Macquarie Bank
- MetService (New Zealand)
- National Australia Bank
- Taste Technologies Limited
- Torus Games

[31] http://www.pairprogramming.com
[32] cited in http://www.xprogramming.com/xpmag/refactoringisntrework.htm

Agile Method Resources

Bibliography

1. **“Adaptive Software Development: A Collaborative Approach to Managing Complext Systems”**, James Highsmith, Dorset House, 2000
2. **“Agile Software Development”**, Alistair Cockburn, Pearson Education, 2002
3. **“Agile Software Development Ecosystems”**, Jim Highsmith, Pearson Education, 2002
4. **“Agile Software Development with Scrum”**, Ken Schwaber and Mike Beedle, Prentice Hall, 2002
5. **“The Benefits of Visibility”**, Larry Constantine, Computer Language Magazine, Volume 9, #2, 1992
6. **“Birth of the Chaordic Age”**, Dee W. Hock, Berrett-Koehler, 1999
7. **“Characterizing People as Non-linear, First-Order Components in Software Development”**, Alistair Cockburn, Humans and Technology Technical Report, October 1999
8. **“The Cluetrain Manifesto: The End of Business As Usual”**, Rick Levine, Christopher Locke, Doc Searls, David Weinberger, Perseus Publishing, 2000
9. **“Constantine on Peopleware”**, Larry Constantine, Prentice Hall, 1995
10. **“DSDM – Dynamic Systems Development Method: The Method in Practice”**, Jennifer Stapleton, Addison Wesley, 1997
11. **“Extreme Programming Applied: Playing To Win”**, Ken Auer and Roy Miller, Pearson Education, 2002
12. **“Extreme Programming Explained: Embrace Change”**, Kent Beck, Addison Wesley, 2000
13. **“Extreme Programming Explored”**, William C. Wake, Addison Wesley, 2002
14. **“Extreme Programming Installed”**, Ron Jeffries, Ann Anderson, Chet Hendrickson, Addison Wesley, 2001
15. **“Extreme Programming in Practice”**, James Newkirk and Robert C. Martin, Addison Wesley, 2001
16. **“A Generative Development-Process Pattern Language”**, James O. Coplien, Pattern Languages of Program Design 1, Addison Wesley Longman, 1995
17. **“The Lexus and the Olive Tree”**, Thomas Friedman, Harper Collins Publishers, 2000
18. **“Peopleware: Productive Projects and Teams”**, Tom DeMarco and Timothy Lister, Dorset House, 1987
19. **“Planning Extreme Programming”**, Kent Beck and Martin Fowler, Addison Wesley, 2001
20. **“A Practical Guide to Feature Driven Development”**, Stephen Palmer and John Felsig, Prentice Hall, 2002
21. **“Principles of Software Engineering Management”**, Tom Gilb, Addison Wesley, 1988
22. **“Project Retrospectives: A Handbook for Team Reviews”**, Norm Kerth, Dorset House, 2001
23. **“Rapid Development”**, Steve McConnell, Microsoft Press, 1996
24. **“Refactoring: Improving the Design of Existing Code”**, Martin Fowler, Addison Wesley, 1999
25. **“Test Driven Development”**, Kent Beck, Addison Wesley, forthcoming

26. **“Testing Extreme Programming”**, Lisa Crispin, Ken S. Rosenthal, Tip House, Addison Wesley, 2002
27. **“The Tipping Point”**, Malcolm Gladwell, Abacus, 2001
28. **“What is Software Design?”**, Jack Reeves, C++ Journal
29. **“Worldwide Benchmark Project Report”**, Howard Rubin, Rubin Systems Inc, 1995

Web Sites

http://jakarta.apache.org/ant/index.html	Ant build tool
http://www.agilealliance.org	The Agile Alliance
http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm	Jack Reeves’ article on software compilation as construction
http://www.controlchaos.com	Ken Schwaber and SCRUM
http://www.crystalmethodologies.org	Information about both Crystal and Adaptive Software Development
http://www.junit.org	JUnit downloads and support
http://www.khatovartech.com/resources	Khatovar Technology (Australia) list of XP and agile related resources
http://www.martinfowler.com/articles/newMethodology.html	Martin Fowler’s article on why we need a new methodology
http://www.pairprogramming.com	Pair programming research and articles
http://www.poppendieck.com/talks/Sustainable_Agile_Development.pdf	Mary Poppendieck presentation comparing agile manufacturing and software processes
http://www.retrospectives.com/	Norm Kerth’s site on project retrospectives
http://www.testing.com/agile	Testing in agile methods
http://www.xprogramming.com	Ron Jeffries’ XP site