XP: Have you got the discipline?

Tim Mackinnon, Agile Coach
Iterex Ltd.

This article appeared in TickIt Internation magazine – Issue #2Q04
(http://www.tickit.org/ti2q04.pdf)

Introduction

"I know in advance, having seriously considered it, that constant pair programming is a totally warped idea."[1] – this is one of the many objections to the practices advocated by the agile software development method, eXtreme Programming (XP). While no longer a new phenomena, agile methods are still viewed as being controversial and are the subject of many articles (it was a topic of a previous TickIT article[1]) both for and against them.

Of the agile methods, XP was one of the earliest, and it is the one that draws the most controversy. Descriptions of this method of development began to appear around 1998 on the Wiki-Wiki-Web[3] and a year later in the book, "eXtreme Programming explained". Five years later on – this agile method is still viewed with both enthusiasm and scepticism. At ThoughtWorks, we have successfully used many agile methods, including XP, on different sized projects throughout the UK, India and North America. While we are big advocates, there are others who still want to "highlight the fatal flaws in the XP methodology"[6].

With such a wide variation in opinions, what is it that is causing such depth of emotion? This article describes 5 years of experience using the practices described by XP and gives you some insight into what this method is about so that you can form some of your own opinions. In part this article tries to address the concerns raised by Keith Southwell [2] about different communities not talking to each other or being aware of each others existence. Far from being a development method that lacks in discipline, my feeling is that agile methods like XP complement the objectives of TickIT, "improving the quality of software and its application."

The Software Development Crisis

To set some context about agile methods, it is worth considering the view from outside the software development world. Even today we read about software project disasters. There have been many attempts to describe how to write better software from "waterfall" methodologies to "unified processes", but it still seems that we have to answer many questions like:

Who's happy with…

- Too many projects not being delivered
- Software taking too long to get to market
- Requirements not being met
- Having to "get it right" first time/up front
- High costs to make changes after delivery
- Unhappy customers
- Unhappy developers

Many of these issues will strike a chord with different people. From the pressure of business analysts trying to make sure that they haven't missed a crucial design detail (not to mention the users having to ensure that they have passed on all the relevant information to those analysts) to the poor software developer who now finds that writing software is just getting so complicated and stressful that its simply not fun to write anymore.

Something seems to have gone wrong somewhere, and so it was very startling to read about a completely different approach to these problems in early 1998.

The Agile Umbrella

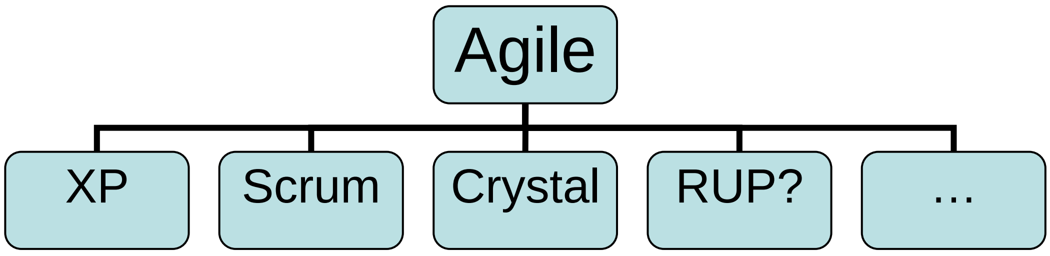
Before describing XP in more detail, it's worth describing what agile methods are, as this is a term that is now very common in the industry. After XP was documented[5], many other groups stepped forward and described methods that also seemed to be solving similar problems. In early 2001, representatives from these different groups met to discuss their similarities and differences and in doing so they found enough commonality to form the Agile Alliance[7]. This commonality has been expressed in the Agile Manifesto:

"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 (in bold) more."

While this manifesto gives a useful context to discuss agile methods, its loose definition means that many methodologies fall (or can be adjusted to) into its definition. As such, the term agile is an umbrella for many methods as shown below:

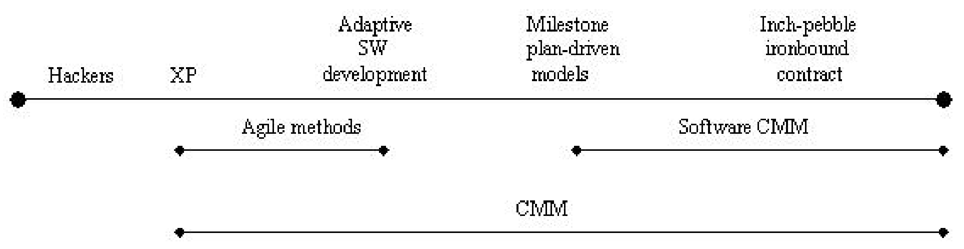


In this diagram, I have deliberately shown XP on the far left, as (which its name implies) it is the most thorough in its definition of what it requires to support its delivery. In fact many of the other methods now describe themselves as "wrappers" over XP so that they can also achieve some of its precise definition.

A Word On Discipline

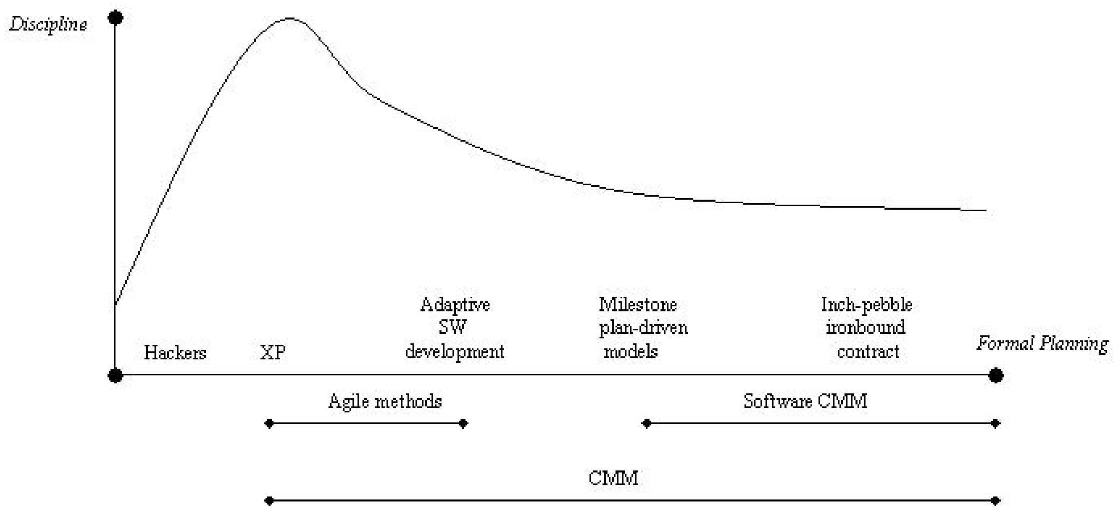
With the Agile Manifesto, there is a tendency to look at its guiding principles and come to the conclusion that agile methods are very loose in their rigour. This often leads to the comparison that agile methods are simply Hacking by another name. In his article, "Get Ready for Agile Methods with Care"[9], Barry Boehm (Director of the University of Southern California, Center For Software Engineering) shows this comparison in a continuum of planning discipline (below).

The Planning Spectrum



At the far left (i.e. no planning discipline) is Hacking, and next to it are the Agile methods starting with XP. At first glance, this figure is quite alarming as anything close to hacking would not be viewed as a quality process in any organisation. Later in the article, however, the author goes on to further describe that agile "places more value on the planning process than the resulting documentation, so these methods often appear less plan oriented than they really are." He then goes on to discuss the comparison to hacking, "although hard core hackers can use agile principles to claim that the work they do is agile, in general these methods have criteria, such as eXtreme Programming's 12 practices, that help determine whether an organisation is using an agile method or not". In reality, when visiting software development teams we find that the practices that agile methods (in particular XP, or those that wrap XP) dictate, result in extremely high discipline (below).

The Development Discipline Spectrum



Even those teams that choose more formal planning methods seem to struggle to achieve, at the grass roots developer level, the degree of repeatable discipline that agile teams consider normal.

Introducing XP

So what is it that differentiates XP from other methods? Kent Beck defined eXtreme Programming as "a lightweight methodology for small to medium sized teams developing software in the face of vague or rapidly changing requirements."[5] With regards to this definition, Beck's original material constrained itself to team size, however with some slight modifications we have found that larger teams can also be effective while still preserving the ethos of the method. The second part of the definition constrains itself to problems with rapidly changing requirements, which given today's business environment is typically the normal state of affairs.

One of the many complaints about XP is simply its name; it conjures up visions of uncontrollable programmers madly racing down mountains and taking risks. In part, the choice of controversial name was done deliberately to make a "startling sentence"[8] that would make people consider what was different about the approach. However, the name is also grounded on the idea that if something is good, you should start "turning the dials to 11!" and do more of it. There are many examples of this but here are some common ones[5]:

- **If code reviews are good**, *review code all the time*
  (referred to as Pair Programming)
- **If testing is good**, *everybody will test all the time*
  (referred to as Test Driven Development)
- **If design is good**, *make it part of everyone's daily business*
  (referred to as Refactoring)
- **If simplicity is good**, *always leave the system with the simplest design that supports it's current functionality.*
  (referred to as Simple Design or The simplest thing that could possibly work)

Of course, it's easy to take this analogy "to the extreme" and cite other examples such as eating chocolate, which taken to the extreme will of course make you sick. The trick however, is to choose the right balance of elements. This is somewhat similar to eating a balanced meal – if you eat too much of one food group you will lack in certain nutritional elements and risk becoming sick.

To help choose the correct elements, XP has some core values, which although rather general, do help set a context:

- Communication
- Simplicity
- Feedback
- Courage

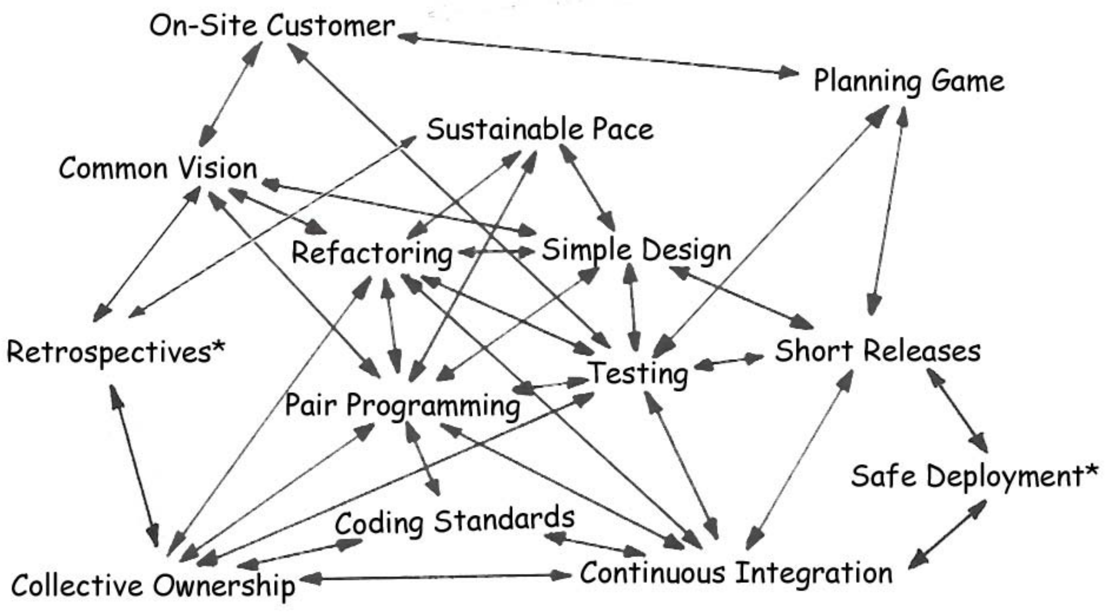
It is through valuing communication that teams will ensure that they talk amongst each other (especially with the customer) and it is through simplicity that teams will write code that is easy to read and maintain. By valuing feedback, teams will test things to discover if there are mistakes and through courage they are empowered to fix the real problems instead of just papering over the cracks. While this list is fairly self-explanatory, it doesn't really say anything unusual (for example most teams would say that they value communication), and as such it is more interesting to look at the concrete practices that XP advocates for balancing these values.

The XP Practices

The full list of practices is shown below, and it's interesting to note that theses practices are simple, common sense techniques, most of which have been used in the computer industry for years:

1. Planning Game
2. Small Releases
3. Refactoring
4. Pair Programming
5. Collective Ownership
6. Testing
   - Customer Tests
   - Developer Unit Tests
7. Simple Design
8. Continuous Integration
9. Sustainable Pace (or "40 Hour Week")
10. Whole Team (or "Onsite Customer"
11. Coding Standards
12. Common Vision (or "Metaphor")

From this list you can easily pick out well-known practices such as testing, something the software industry has always advocated, and similarly coding standards which have long been recognised as a good discipline for large teams. Later in this article I will discuss details and insights for each practice, however it is worth mentioning that a few of the practices are more controversial than others, especially when taken to the extreme "dials to 11" level. A common example is "Pair Programming" which often provokes a reaction similar to the quote at the beginning of this article. What is important to understand however, is how these simple practices work together to form an interlinked platform that is collectively more powerful than the individual items. The following is a slightly updated version of the reinforcing practices diagram in [5], which also shows 2 additional practices marked with an asterisk (explained later).



This platform relies on each practice to balance with its neighbour to form a stable foundation that supports agile software development. However, maintaining this balance is not trivial – if you imagine that the platform is supported by a series of poles (the practices) then in order to keep the platform level each pole has to be kept steady and at the same height. This is quite tricky and at times you find that the platform tilts in one direction and needs additional support from the rest of the team to prop up a weaker practice. However, if everyone runs over to give support to that practice then something else will now be weaker and so the platform then tilts in another direction. After a while, the team learns how to apply the right amount of support to each practice and can maintain a platform that is reasonably flat.

This diagram is also useful when considering the environment you are operating in. For example, if you find that you have trouble having an onsite customer, then (and although it's not ideal) you can substitute a proxy customer in the form of a business analyst. However you now need to consider which other practices are associated with an onsite customer because they will be affected and will need extra support to compensate for the loss. In this example, planning, testing and common vision will all be impaired and will need additional effort to maintain a consistent balance. This is only a rule of thumb, and should be approached with some caution however with a good coach or experienced team members it is definitely an option.

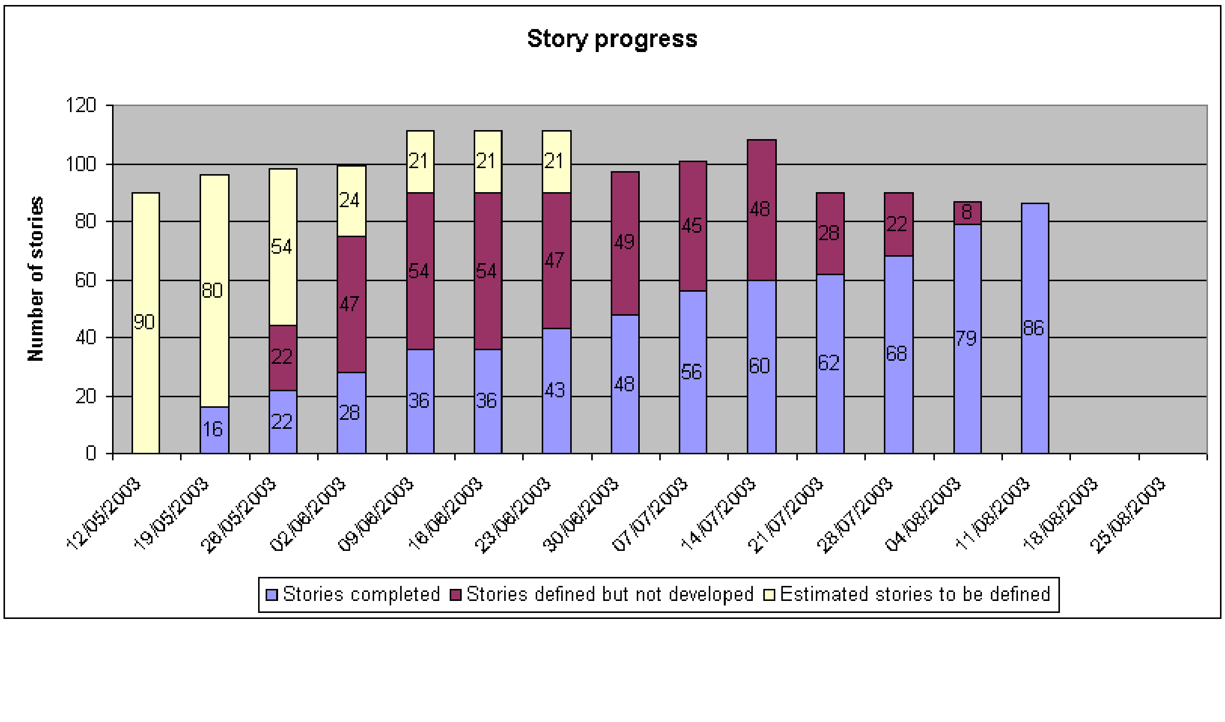
To get a better understanding of how XP works under the hood, I will describe a path through some of the reinforcing practice and explaining some of the details of the relationships that we have discovered over the years. It is also worth remembering that there is a complex relationship between all the practices and we are constantly discovering situations which when later analysed revealed a weakness that we hadn't considered before.

The planning Game

For many, planning is the basis of successful development, and XP is no exception – there is a strong emphasis on planning both at the larger release level, and closer to the ground at the smaller iteration level. The key to successful planning is to ensure that it is done in a simple and transparent manner that is clear to both the customer and the developers.

Release Planning

To begin a successful project, users create a series of high level user stories which can be estimated by the developers to get a rough estimate of the size of the project. Because these stories are very coarse grained, the developers estimate in weeks (or partial weeks) and the estimates are averaged out and added up to give a total duration for the project which can be used as a high level tracking device for the project.



It is important to realize that this is not an exact science, and as such it doesn't require huge amounts of effort (hours) – however it gives a fairly accurate forecast for the project that is then refined at the iteration level. In the figure above, the users had roughly 9 functional story areas (e.g. reporting, user administration, numerical entry etc.) The developers collectively estimated these high level stories at 22 weeks or 110 days of work. From previous development spikes they had measured that with 3 pairs they could complete 8 ideal days of work in a weekly iteration. Thus the project was estimated to take 110 / 8 or 14 weeks. In the figure above, this was graphed by story over the course of the project. It was estimated that each functional area would break into about 10 user stories (giving a total of 90), and graduations were plotted on the graph for 14 weeks. Some initial release level stories were broken down into user stories and the team began work. On the first iteration they were extremely successful and completed 16 stories (they were quite easy stories as the users who were new to XP and were still learning how to write stories). You can also see from this graph that the users were a little behind in writing user stories and it was only by the third iteration that they had written a backlog of 22 stories for the development team to draw from (this is not an ideal situation, but it's a situation we often encounter and can deal with). By the eighth iteration (half way) the users had finally completed a total of 97 user stories (with a backlog of 49). In iterations 9 and 10 they added some additional stories (feature creep), however the transparency of tracking made it very easy to see that they would be unable to meet their first product deadline without deferring some of the less important. After some discussions, and as the users were part of the team and realised the tradeoffs and costs involved, they deferred some less important stories to a second release. The first milestone was completed by the 18th of august and the product went live that week.

Iteration Planning

If we now look a little deeper at how planning actually took place for each iteration, we again see a simple and transparent process. Looking at the third iteration, the developers had finished 6 story cards. At the iteration level we are dealing with much finer time grains and so at this level we choose to be a little more accurate with our tracking. The cards in this iteration had been estimated as follows:

3rd Party Setups (#5.1)	1.00	100%
Sort on product (#53.16)	1.00	0%
Group by Country (#93.1)	0.25	100%
Transfer Simple Product (#62.5)	0.25	100%
Transfer Complex Product (#66.2)	2.00	100%
Totalling (#53.20)	0.25	100%
Country Product Setup (#16.2)	0.25	100%
Total	5.00	

Notice the team didn't finish card #53.16 which is why only 6 cards were counted in the release plan described above. However at the detail level, we calculate development velocity by counting up the total estimates for those cards that were completed, thus in this case the velocity is calculated as 4 units. Now using the concept of yesterday's weather, the team can sign up for 4 units of work in iteration 4. The potential stories for the next iteration are laid out on a table and the team collectively places estimates on them based on their previous experience. Note, we differ slightly from the technique presented in [5], and use team estimates which are much simpler than having individual developers track personal velocities.

The user can then select from the estimated stories up to a total of 4 as shown below:

As is typically the case, there are more stories than will typically fit in the iteration and so the user has to carefully consider which stories should be worked on first. If is from this act of selecting stories that add up to a measured velocity that this practice is called a game – introducing a new card in the line-up of the iteration can trigger the stories to be re-estimated and result in the new challenge of getting everything to add back up to the fixed velocity again. In the picture above, there are 3 users who are looking at the 2nd and 3rd columns of cards to find something that will fit in the space in the first column and bring the total back up to 4. After some deliberation they agreed on the following:

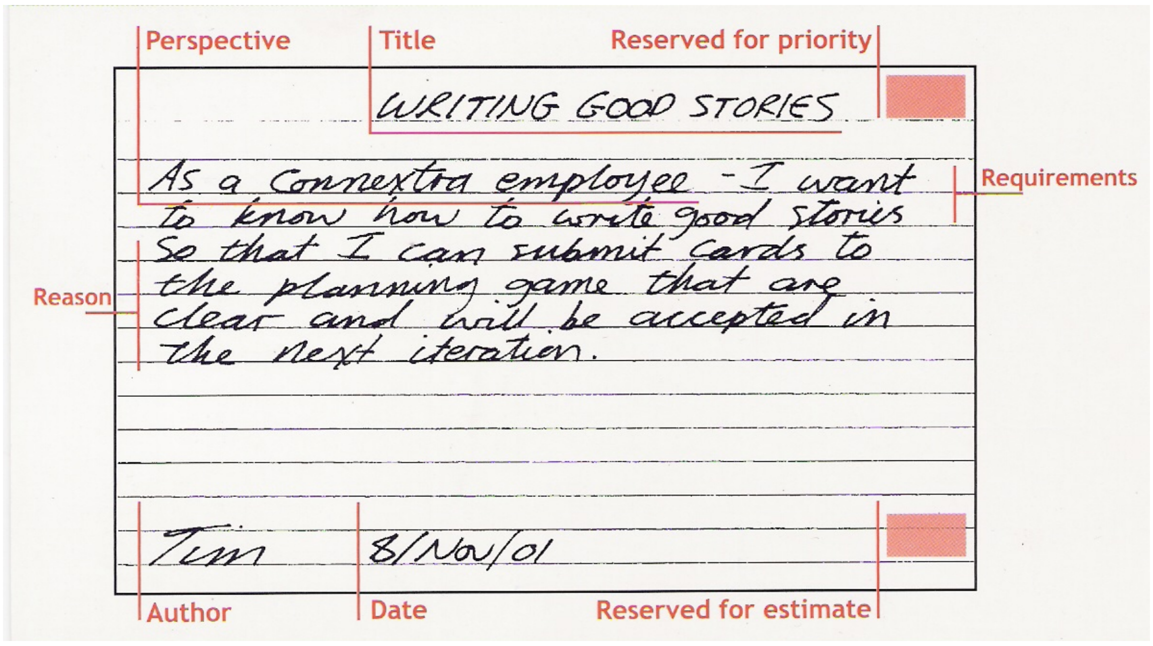
Sort on product (#53.16)	1.00
Product Setup (#62.2)	0.25
Transfer New Product (#66.3)	0.50
Database Qualifiers	0.50
Transfer Multiples (#62.3.5)	0.50
Stock Counts (#62.4)	0.50
Warehouse Isle Setup (#62.1)	0.50
Display Type of Trade (#89.1)	0.25
Total	4.00

As development iterations are typically 1 to 3 weeks in length (in our experience 1 week iterations are highly recommended as they give quick feedback and are much faster to plan due to their small size) any cards that are deferred are available to reconsider in a relatively short space of time. In the case of story #62.3.5, the users agreed to split the story, #62.3, into two smaller pieces so that they could get something that would fit into the iteration sooner.

Finally, once an iteration has been created, the development team will then take any cards larger than half a day and split them into meaningful development tasks to ensure that measurable progress can be made on that story. We have noticed that stories that are roughly larger than 3 days tend to run into difficulty and so would typically suggest that these stories be split into useful chunks that can be more easily tracked. While we don’t strictly enforce this, we found that users noticed this trend as well, and so they started getting better at writing smaller more measurable stories of their own accord. In fact, this tendency actually made the higher level story count release tracking strategy even more effective.

Onsite Customer

As you should have noticed from the description of the planning game, there is a big reliance on having meaningful story cards that can deliver business benefit. Rather than the development team making up what they think the customer needs, there is a strong reliance on the customer working with the team to express stories that can be easily estimated and delivered in an iterative way. Furthermore, because the customer can see how their stories impact development on a daily and weekly basis they can fine tune the way they request stories or specify acceptance tests. Over the years we have found that the best format for story cards is as follows:



WRITING GOOD STORIES

As a Connextra employee - I want to know how to write good stories So that I can submit cards to the planning game that are clear and will be accepted in the next iteration.

Tim 8/Nov/01

While story cards are a useful token for discussion in a planning game, they don’t represent a full description of the requirement when it is implemented. In fact they are really “just in time” requirements. Rather than spending time upfront to fully specify every minute detail of every requirement in the system (some of which may never actually get implemented), users instead write lighter weight place holders in the form of story cards. When a card is actually chosen for an iteration, and work begins on it, then the developers can sit down with the users and fully establish the exact details like fonts to use, or exact validation rules. This works particularly well if the team sits with, or near the customer such that everyone can overhear conversations and step in if simplification can be made or specific details are being missed.

It is also important that customers establish acceptance tests for stories before work begins on them. In this way everyone is clear on what is expected from the delivery of that piece of functionality, making it easier to get signoff in that iteration and thus maintain or improve the development velocity. Again, because there is a real emphasis on their being a whole team this is not an adversarial relationship but one of mutual respect to get the best job done that is possible.

Testing

Following along the lines of supporting relationships, in order for the development team to ensure the successful completion of a story, they need to work with the onsite customer to develop tests that show successful completion. Furthermore, they also need to continue to rerun these tests to ensure that the addition of future functionality doesn't break anything.

Testing is a very big deal on an XP team, and a lot of time and effort is put into establishing tests before any functionality is written, this is referred to as Test Driven Design and it has several important benefits:

- It establishes the completion criteria for a piece of work (once the tests pass, no further work is required unless additional tests need to be written first)
- It ensures that tests are actually written (afterwards is often too late, as no-one is interested)
- It is a design technique that ensures that testable software is written that provides functionality from a "client" driven perspective

There are typically two types of tests, User acceptance tests, and developer Unit Tests.

User Acceptance Tests

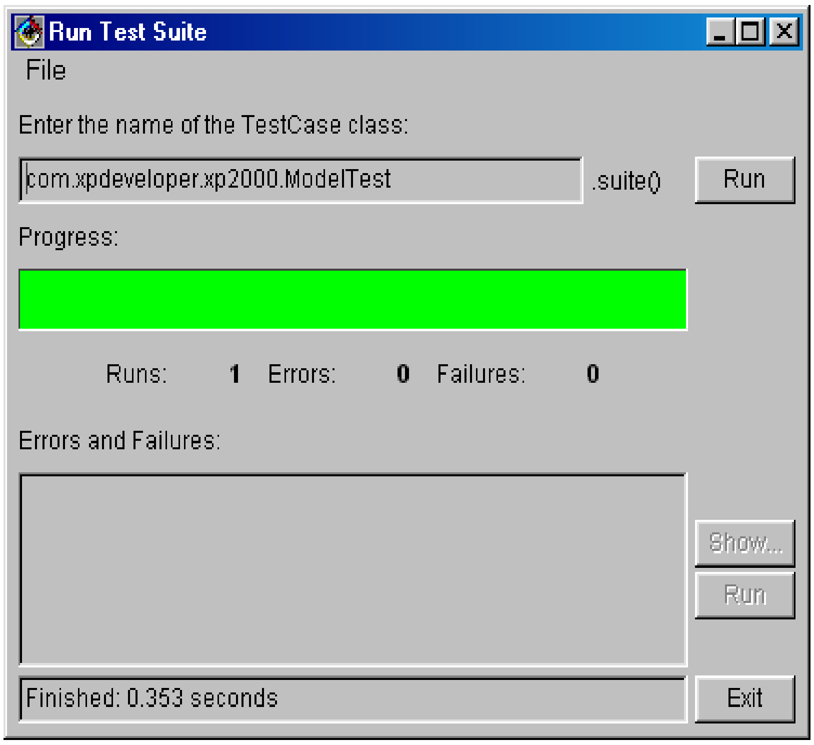
Acceptance tests are normally defined in customer terms, often in a format similar to the following:

Action	**Result**
Enter a valid product in the order entry screen with quantity of x	When you finish entering a quantity the total field will show "product price" * x

The team will attempt to automate these tests such that they can be run repeatably to confirm that functionality has not been broken. There are many tools from FIT to Marathon that can help with this, and many teams will also have a dedicated QA tester to work with the customer to ensure that this happens. We have also had some success with tools that allow the user's expression of the problem to be used to generate code stubs that the developers then codify to give repeatable tests. In the worst case scenario, teams can resort to simply having manual test scripts which over time they will work out how to automate.

Developer Unit Tests

Along with acceptance tests, when working on a story that has been clarified by a customer, developers will begin coding by writing a test case that will help them express what the code must do to satisfy its task requirements. This technique is referred to as test-driven design and is an extremely important discipline on XP teams. For many developers this experience is quite different from what they are used to, however if you pair program with someone knowledgeable with this technique you quickly appreciate its powerfulness. Discussing what form a test should take is definitely a concrete way of having a design discussion. Having got a test in place, it is very noticeable that most pairs will then run the test to make sure that it actually fails and is not accidentally solved by some pre-existing code. Finally, partners will then take it in turns to work out the best code that actually solves the design problem and allows them to check in a completed, tested piece of functionality. The defacto test framework that supports this type of activity is refereed to as X-Unit, which has versions for Java, C## and most other popular programming languages:



Enter the name of the TestCase class:
com.xpdeveloper.xp2000.ModelTest .suite()

Progress:

Runs: 1 Errors: 0 Failures: 0

Errors and Failures:

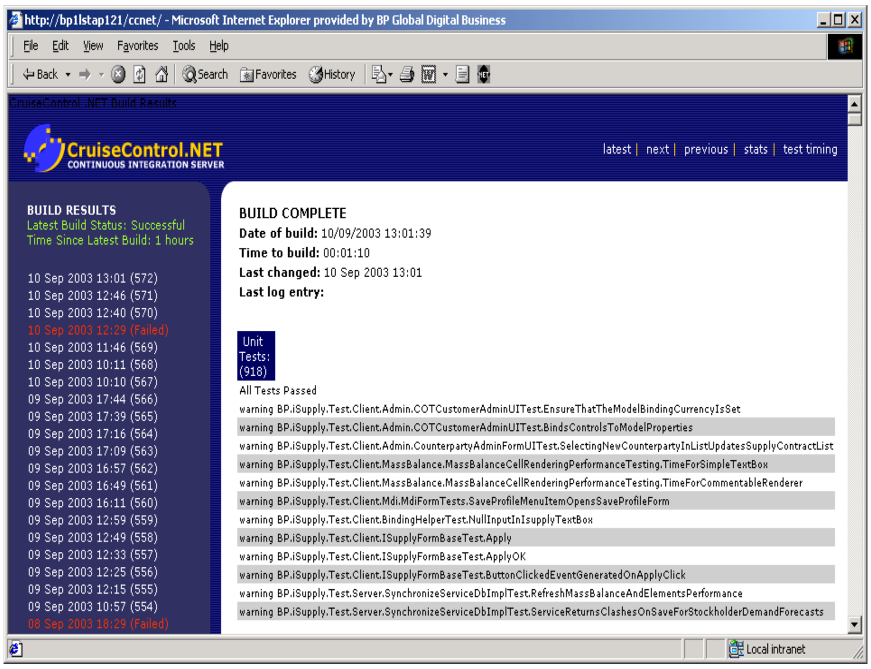
Finished: 0.353 seconds

This technique has become so popular in the development community that most modern development environments now include an integrated X-Unit test runner that makes continuously running tests very quick and easy.

An additional benefit of writing unit tests is that they serve as a form of living documentation for the code that they are testing. Applications such as TestDox will create documentation based on the names of tests that have been written which has the side effect that developers have a useful reason to give tests meaningful names that will give them documentation for free. In this way, rather than creating documentation that invariably gets out of date, the documentation is actually executed and so is kept up to date as part of the test suites that run it.

Continuous Integration

Having discussed the strong relationship between customers and testing, continuous integration is another important practice that leverages the value of testing and ensures that customers and developers alike have the feedback that the software is correctly building to create something that can be frequently used. Rather than waiting until the end of an iteration to discover whether everything still works, XP teams integrate and build hourly (or even more frequently). Normally this practice requires that a team uses a version control system and some software (for example CruiseControl) that checks out the current code baseline, recompiles it, runs all of the tests and then finally creates an executable that can be installed by the customer on their machine or in a test environment.


http://bp1lstap121/ccnet/ - Microsoft Internet Explorer provided by BP Global Digital Business
File Edit View Favorites Tools Help
Back Search Favorites History
CruiseControl .NET Build Results
CruiseControl.NET
CONTINUOUS INTEGRATION SERVER
latest | next | previous | stats | test timing
Local intranet

BUILD RESULTS
Latest Build Status: Successful
Time Since Latest Build: 1 hours

10 Sep 2003 13:01 (572)
10 Sep 2003 12:46 (571)
10 Sep 2003 12:40 (570)
10 Sep 2003 12:29 (Failed)
10 Sep 2003 11:46 (569)
10 Sep 2003 10:11 (568)
10 Sep 2003 10:10 (567)
09 Sep 2003 17:44 (566)
09 Sep 2003 17:39 (565)
09 Sep 2003 17:16 (564)
09 Sep 2003 17:09 (563)
09 Sep 2003 16:57 (562)
09 Sep 2003 16:49 (561)
09 Sep 2003 16:11 (560)
09 Sep 2003 12:59 (559)
09 Sep 2003 12:49 (558)
09 Sep 2003 12:33 (557)
09 Sep 2003 12:25 (556)
09 Sep 2003 12:15 (555)
09 Sep 2003 10:57 (554)
08 Sep 2003 18:29 (Failed)

BUILD COMPLETE
Date of build: 10/09/2003 13:01:39
Time to build: 00:01:10
Last changed: 10 Sep 2003 13:01
Last log entry:

Unit Tests: (918)
All Tests Passed

```
warning BP.iSupply.Test.Client.Admin.COTCustomerAdminUITest.EnsureThatTheModelBindingCurrencyIsSet
warning BP.iSupply.Test.Client.Admin.COTCustomerAdminUITest.BindsControlsToModelProperties
warning BP.iSupply.Test.Client.Admin.CounterpartyAdminFormUITest.SelectingNewCounterpartyInListUpdatesSupplyContractList
warning BP.iSupply.Test.Client.MassBalance.MassBalanceCellRenderingPerformanceTesting.TimeForSimpleTextBox
warning BP.iSupply.Test.Client.MassBalance.MassBalanceCellRenderingPerformanceTesting.TimeForCommentableRenderer
warning BP.iSupply.Test.Client.Mdi.MdiFormTests.SaveProfileMenuItemOpensSaveProfileForm
warning BP.iSupply.Test.Client.BindingHelperTest.NullInputInIsupplyTextBox
warning BP.iSupply.Test.Client.ISupplyFormBaseTest.Apply
warning BP.iSupply.Test.Client.ISupplyFormBaseTest.ApplyOK
warning BP.iSupply.Test.Client.ISupplyFormBaseTest.ButtonClickedEventGeneratedOnApplyClick
warning BP.iSupply.Test.Server.SynchronizeServiceDbImplTest.RefreshMassBalanceAndElementsPerformance
warning BP.iSupply.Test.Server.SynchronizeServiceDbImplTest.ServiceReturnsClashesOnSaveForStockholderDemandForecasts
```

By doing this the team ensures that if there is an error in the build (either through a compiler error or failing tests) it is within recent memory of the team to discover what has been changed. Correcting the problem is then usually quite simple and it has the extra side affect of encouraging developers to make sure they are running all of their tests on their local machines to ensure they don’t cause such breakages which slow down the rest of the team.

We often find that teams additionally leverage the continuous build cycle to add additional consistency or data migration checks to make sure that coding standards are being met or that data integrity has not been compromised.

Pair Programming

When using techniques like continuous integration, from time to time something will break and the team will be notified that something has gone wrong. Working out that cause of the problem can sometimes be challenging and it often helps if you can sit with someone who knows something about the code that you have conflicted with to work out an acceptable solution. This is often most developers’ experience with pair programming and no-one seems to dispute how useful this is. However, on an extreme team, this situation is taken a step further – and developers are encouraged to pair constantly throughout the day.

Unfortunately this practice goes very much against the traditional view of programming as a solo, genius locked in a room, type activity and so it upsets many conventional programmers. Alongside this is the experience that many developers have had of watching someone type at a keyboard – which is completely boring and pointless. Paired programming is not at all like this – it's a collaborative way of working where two people help each other achieve a common goal. Just as when driving a car, it's helpful to have a navigator reading the map and keeping track of directions while the driver concentrates on the immediate streets and rules of the road. So to with paired programming, the driver is typing in the code and making sure that everything will compile. The navigator on the other hand is usually holding the story card being worked on and is making sure that the driver is coding up something that satisfies their agreed upon test. At times the driver may get "mouse blindness" and lose track of the direction at which point it's common that the navigator will grab the keyboard and step in with a sense of renewed direction.

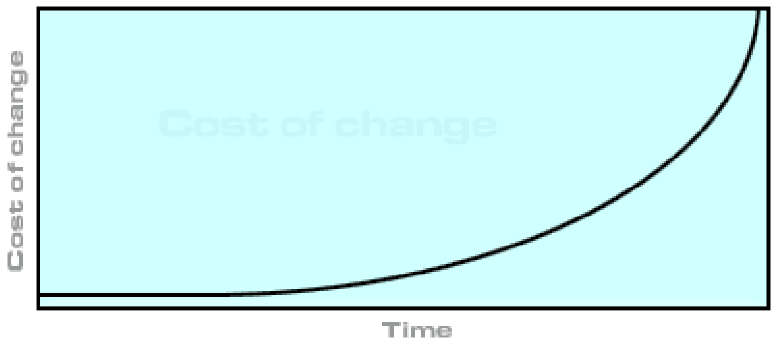
Far from being a wasteful, two people at one keyboard is extremely efficient, not only is it a form of "Constant QA" but its also a way of encouraging a sharp focus as partners prevent each other from getting distracted on side issues that are not core to the task being worked on. This type of activity seems to work best when the pair is practicing test-driven design, as it is this practice that sets boundaries for limiting over-development or task distraction. Furthermore, developers find that they encourage each to work better, as well as creating an enjoyable experience that makes work more fun.

As far as project risk is concerned, pair programming is also an excellent way to ensure that knowledge is not trapped inside a single developer who may cause a project to be stalled if he or she is sick or away on a business trip. It also has additional health benefits as by changing roles from navigator to driver, pairs reduce incidents of RSI and eye strain, however it does require that work areas are adjusted such that two people can comfortable sit side by side at a desk without knocking knees on under table filing cabinets.

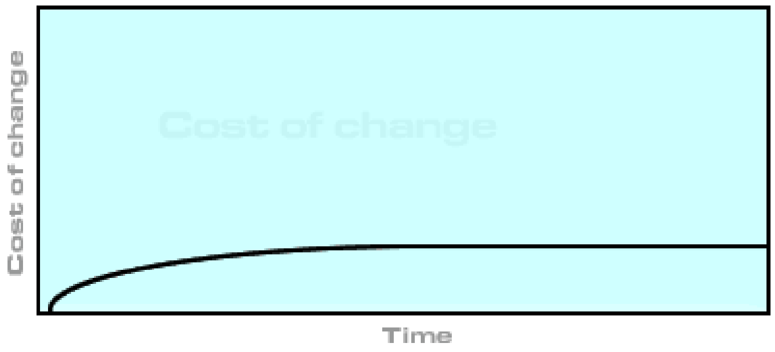
Finally, many people ask if paired programming works only for pairs of equal experience. This is definitely not the case, a senior developer who works with a novice plays the part of mentor and through explaining what it is they do they often find that they improve their own practice or discover pieces of the system that don't work as described. The novice programmer, far from always listening can definitely question tests – in fact their most powerful weapon is to ask "what's the test for that" (which often prevents an over eager expert from jumping ahead without putting in place an important test). Similarly, two junior programmers is also very effective as it gives them a chance to compare notes and work at a slightly slower pace and reinforce the learning they have achieved through working with more advanced partners.

Simple Design

While pair programming might be controversial to some, few would argue for having an overly complicated design. However, achieving simplicity is not an easy task unless you consider that by leveraging the related practices there is a real opportunity to remove unnecessary complexity. Many software developers have learned the old adage that as time progresses the cost of correcting a change increases exponentially:



And in this type of situation, no-one would ever want to be the poor soul that cost the company £10,000 for a change that could have been implemented for £100 early on in the course of the project. However, with modern tools and supporting development practices like full unit testing and ruthless refactoring, it is possible to keep the cost curve to a slow increase.



Furthermore, developers in XP teams have a definite instinct to not over design in advance. The cost of carrying features that are not yet being used adds to the cost curve. With full unit tests it is very easy to add features exactly when they are needed. Similarly if the existing system is already in the simplest state possible, when new features are added it is the minimum amount of code that is in place that may need modification. Previous to working on XP projects I have often corrected a bug in some code and then gone through all of the compiler errors to get the system working again, only to notice most of the way through the changes that I am actually fixing problems in code that is either duplicated or should have been removed ages ago. It's wasteful, and it clearly makes you understand how the cost curve can be artificially inflated by carelessness. Fortunately, supporting practices like pair programming and refactoring make it more likely that you will work with someone and spot some duplication and then take the time to correct those problems exactly when you spot them. This is especially easy when you have the confidence of a version control system that will allow you to go back and recover the extra code if ever you need it.

Refactoring

As previously mentioned in Simple Design, if you spot duplication in code it is expensive to carry that duplication around with you. Refactoring is the practice that encourages you to remove this duplication. In its simplest terms, refactoring is defined as:

A series of small steps, each of which changes the program's internal structure without changing its external behaviour

The reason to refactor can actually be more than just removing duplication it is also a useful technique for improving the readability of code. Furthermore, refactoring as a technique of small steps, is also a useful way of avoiding big sweeping changes that can cause big code breakages of days or more.

It is important to note that refactoring is **NOT** a re-design activity whose changes will impact the whole team. If something is large enough to require discussion with the whole team then it is dangerous to call this refactoring. It is better to write a story for the redesign and discuss with the customer why you need to change to the design to accommodate their requirements. As refactoring is critically important to keep the change curve flat you don't want to get in a situation where customers are questioning why you are spending time refactoring.

Retrospectives

Once you get a feeling of how all of the practices are related to each other, it is important to take time to check that there are not any problems with the balance of each of them. As described at the beginning of this article, if everyone is concentrated on a weaker practice then there is a high probability that something else is suffering and causing the platform to tilt in another direction. One practice that acts as rebalancing agent, is that of holding frequent retrospectives. This is a new practice not mentioned in the original white book, but one which the XP community has unanimously embraced, originally defined as:

> *retrospective (rèt´re-spèk-tîv) -- a ritual held at the end of a project to learn from the experience and to plan changes for the next effort.*

In XP terms, this practice is typically used monthly (rather than at the end of a project) to asses how well the team is working with regards to its process. In an efficient team it is quite common for issues to build up that need a release, and the act of taking time to discuss "what has gone well", "what is not going so well" and "what puzzles us" in a structured manner is extremely helpful for the team to redistribute its resources effectively. Normally a team will build up a series of actions which they will prioritize and select a few of for implementation in the weeks following the retrospective.

Safe Deployment

On larger teams, especially those in a larger corporation, deploying a finished application to "live" becomes a big deal. While the balance of original practices normally covers getting software built and delivered we have noticed that deploying is a big enough problem to warrant its own practice. Often this problem is made worse because there is another deployment team involved. The key to safely deploying an application is to ensure that you frequently practice deployment like steps in an automated way. As such we adjust the continuous build mechanism to build additional targets for UAT and Production environments along with a mechanism to switch configuration files relevant for these environments. For this practice to work best it is essential to integrate deployment staff with the team (and rebalance the Whole Team practice). While it may be difficult to have member of this other team permanently around, building up a rapport and having them visit the team to pair on migration verification or script changes is a more successful way to get them comfortable with an automated deployment that they will sanction.

Along with scripts, you may also have to produce some form of support documentation for these staff, and again the trick is to just provide what is strictly necessary, which may be lessened if they understand the build scripts used to create the software. Finally, issue tracking systems such as Jira might also be involved. We have found that these systems can integrate quite well on the team, and any bugs raised can be simply carded with a relevant title and bug tracking number (from that system) that can be used to get full details when the story is worked on (there is normally no real need to rewrite full bug details or make printouts).

Lessons Learned

Having worked with many XP and Agile teams, there are some important lessons that can be passed on.

Hire people who are XP compatible

As with any job, it’s expensive to correct hiring mistakes, so make a point of identifying if a candidate is comfortable with:

- Collective ownership
- Iterative design
- Cultural Fit

Good developers will recognize what XP is solving, and if possible its worth getting them to pair with members of the team so that they can see how well they might fit in.

Observable Process

Make sure the XP practices are visible; you should be able to observe:

- Stand-up meetings around a Planning Board
- Visible Cards with checked off items
- Audible Progress, the “mooing cow” effect of work being completed in short intervals
- Metrics that matter, like velocity graphs
- Planning Games with Customer involvement

While this is just the beginning of the possible observations that you can make, there is something that is very obvious about teams that are practicing XP. If a team has the balance of practices right, then you will definitely notice that they stand out, especially because they work well together.

Summary

There are a lot of misconceptions about eXtreme Programming and the Agile methods that choose to wrap it. Many of these are simply from people who have not taken the time to understand how all of the prescribed practices balance each other out. While it might sound outrageous to take some of them to the extreme, if you do this with the reinforcing practices graph in mind you understand just how an extreme approach can actually work. Far from being undisciplined, the practices that XP embraces are extremely relevant to organisations that want to improve the quality of software and its application. This emphasis on discipline and repeatability is particularly relevant to organisations like TickIT and it would seem that with some careful and pragmatic consideration of what XP is trying to achieve (like those proposed in [2]), that two different communities might well find themselves working on the same team.

In closing, **don't be afraid of XP!**

> *"Fear leads to anger, anger leads to hate, hate leads to suffering"*
> *(Yoda: Star Wars Episode 1)*

Acknowledgements

The author acknowledges the contributions of many people who have helped derive this information over the years: the developers and founders of Connextra, the attendees of XtC, Duncan Pierce, Rachel Davies, Steve Freeman, John Nolan and finally ThoughtWorks, in particular James Davison and Mike Royle.

About The Author

At the time this paper was written Tim Mackinnon was a senior developer at ThoughtWorks, where he coached teams learning Agile Development as well as facilitating workshops and retrospectives.

Tim is now the founder of Iterex Ltd, a company specialising in Iterative Excellence and the creator of the Iterex Professional software. Recognized as an early pioneer of agile development, his contributions include the well recognized techniques: Mock Objects, Gold Cards, Heartbeat retrospectives and Futurespectives. He has worked with numerous organisations, developing, teaching and mentoring teams in effective agile methods.

References

[1] http://www.softwarereality.com/lifecycle/xp/key_rules.jsp last visited march 2004

[2] Southwell, Keith. TickIt October 2002.

[3] http://c2.com/cgi/wiki?WikiWikiWeb, last visited march 2004

[4] http://www.xprogramming.com/publications/distributed_computing.htm, last visited march 2004

[5] Beck, K. *Extreme programming explained: embrace change*. Reading Mass. Addison-Wesley, 1999

[6] Stephens M., Rosenberg D., *Extreme Programming Refactored: The Case Against XP* . Apress. 2003

[7] http://www.agilealliance.org/home, last visited march 2004

[8] http://www.acm.org/sigplan/oopsla/oopsla96/how93.html, last visited march 2004

[9] Boehm, Barry, "Get Ready For Agile Methods With Care", IEEE Magazine, Jan. 2002