In response to Extreme Programming

Extreme Programming (XP) seems to be the subject of choice for software's chattering classes, so I thought it was time I went to the source. This is not a book review of Kent Beck's "Extreme Programming explained", nor is it advocacy for the cause. This is my response to the issues and ideas raised in the book.

The book is worth reading at £20 and under 200 pages it will not break the bank or take you for ever to read – I wish more computing books could be so economical with my time and my money!

My reaction to this book was 75% like reading Design Patterns for the first time; "I've done that, that's the way I did it on project Y," the other 25% was scary, very scary in places. It did succeed in staring up feelings in me though and feeling lead to thought, which is good. Some Overload readers will know of my liking for modern art, well, that 25% of the book was like seeing Tracy Emin's infamous *Bed* in the Turner Awards. It's easy to simply reject it out of hand, but, if you delve deeper and think about it you can obtain and understanding. You may still not like it, but you can appreciate it. Well, 25% of the XP book was like that and it's probably why I feel the need to respond with this article.

Before jumping in a word about my qualifications to write this: I have never done XP, I have however, done a lot of the things described by Beck's XP and I therefore think I'm qualified to pass comment. Much of my personal reasoning on development processes comes from McCarthy (1995) and Maguire (1994), it's several years since I read anything that opened my ideas about the process as much as these two books, I consider Beck's XP a successor to these books[1].

In response to some XP ideas

Changing requirements

One of the dirty little secrets of software development is: specifications don't work. The *Victoria novel* approach to software development, so beloved by out-sourcing companies is fatally flawed. To write a comprehensive spec you have to implement most of the system. Performing comprehensive analysis is a sure fire way to get bogged down on the starting blocks. The only complete specification you will ever have is the program code.

XP copes with this issue in two ways. Firstly, specifications are presented as a set of stories, a story is half use case, half feature. You have a pile (literally, your advised to write these on CRC cards) and the top priorities come at the top and the lesser ones at the bottom. This also allows XP to cope with changes and additions to the spec, you simply add and remove stories (CRC cards) as required.

Secondly, XP accepts that the code is the best source of specification and documentation.

[1] Steve McConnell's *Code Complete* is also comes from the same, Microsoft, stable and although generally highly regards I've never found it says anything that Pressman, or any other of the staple of classic software engineering writers didn't say, albeit in a more academic style.

Iterative cycle

Like McCarthy, Beck is a big, big fan of iterative development. The "release early, release often" school has been gaining ground for several years and is widely used in the Open Source community. I long ago came to realise that large monolithic release just don't work. They are easy to understand and fit into a water-fall development strategy but increase the risks, if you miss a release everything in that release is lost. With a short release cycle, adding a few features at a time, you minimise these risks.

Short release cycles do have a downside: organisations which are change averse will require a major cultural shift. Also, every release carries over heads, these are usually fairly fixed for each release (e.g. customer sign off, release notes, source code labelling) and if you increase the number of releases you will increase the amount of time spent on these activities. Further, every release carries a risk factor, a series of small incremental releases could entail more cumulative risk than one big release.

Testing

Most developers accept testing as a necessary evil. XP has an interesting take on this: write your tests before you code and write them as automated tests. Can't say I disagree with either of these but I don't think it's as easy as Beck makes out. A large part of XP depends on you being able to write automated tests which can be run repeatedly and quickly. Sometimes this won't work: I've been on several projects where it just isn't possible to automate the tests e.g. data is not repeatable because it comes from a market feed or a environment sensor; and I've also written programs with run times of several hours.

Writing the tests before coding should confer some of the benefits of prototyping because you are forced to explore the problem before constructing the solution.

Planning game

Beck claims there are 4 variables in XP: Cost, time, quality and scope[2]: the objective of the planning game is to bring these into equilibrium for the next iteration cycle. Beck encourages a team approach, together with the customer, to the planning game. Beck doesn't pretend, as so many mangers do, that requirements are independent of the time it will take to implement them. I'm sure most developers have faced a high priority requirement X which will take N weeks to implement, and low priority requirement Y with will take N hours: sometimes a little jam today makes a long wait more attractive. This is not to say always we do the easy things first: Beck is right to say tackle the difficult bits first, this way you avoid hitting a show stopper later and have some easy work to look forward to.

Programming in pairs

This is probably the XP idea which gets the most attention and it is undoubtedly not without merit.

---

[2] It's interesting to contrast these with McCarthy's three: features, resources, time. I suspect a little bit of analysis would reconcile these into a single equation.

Personally, I'm not sure its always applicable. Sometimes we just require a good old think, maybe a fiddle and a re-think. I expect working in a pair would make you more prone to get something working without necessarily giving it in depth thinking. But as XP advocates simple design this isn't an issue.... In part I suspect it depends on the temperament of developers: for many programming in pairs will bring a discipline, for others it will seem like a bind. Maybe to pair, or not to pair, for any given piece of work, should be the decision taken at the time by a developer.

Simple design and refactoring:

While I agree with the sentiment of this idea, this is where I have my biggest problem with XP. Let me take it as two items to start of with.

Simple design: Beck says "produce the absolute simplest solution" and re-work it next time. For me this raises so many questions. What is the simplest design? Lets start throwing away some stuff:

- Why bother with namespaces? If we need one we'll add it later
- Why bother with virtual functions? We can make it virtual later.
- Why use an abstract base class? That only complicates things, put all the code in the base class, if we need to derive we'll re-work it.
- Exception handling : we'll add it when we need it.
- In fact, we don't need polymorphism or a class hierarchy at all, we can just add tags to the switch statements later.
- Databases: just add the tables as you need them, we'll normalise later

I can't accept that it is always in the best interest of the system to look the other way and do something very simple every time. Exception handling in particular is notorious difficult to retro-fit yet can, in the long run significantly enhance the understandability and hence maintainability of a system.

I strongly believe that if we pursuit the "simplest design possible" we will end up with code which lacks elegance and extendibility.

Potentially, this rule violates Bertrand Meyer's "open-closed" rule: the code will not be open to extension (as this is the absolute simplest possible why should it be?), and will not be closed to other modules because if another module needs to re-use it we can just refactor it then.

I'm please Beck and others are making refactoring respectable, indeed, almost a buzz word. However I'm not sure that refactoring is possible without some semblance of change to XP principles. Faced with several thousand lines of tag-and-switch code (done that way because it is simple, and lets face it, in a small program a switch statement is easier to follow than virtual methods) I may not be able to refactor to any significant degree.

I suspect that this simple design and refactor is a substitute for prototyping. It amounts to the same thing: write something then throw it away when we have learned about the problem. Compare the development cycles for both:

Prototyping	**Refactoring**
Write prototype	Write version 1
Compare prototype to problem	Release version 1 and see how it deals with the problem

Write the real thing	Refactor the whole thing

My biggest problem with all of this is one of my own rules of thumb built up over several years: Get it right first time, because no matter how much you or you boss thinks you can come back and sort it out later chances are you won't get the time.

To give Beck his due I suspect that his definition of *simplest design possible* may not be as absolute as mine. An individuals definition of *simplest possible* will depend on their experience and views.

Kelvin Henny has suggested that a solution which computes but is not architecturally elegant is not a solution that works. If we moderate our definition of *simplest possible* we may over come many of the problems raised here.

Beck is equally absolute on the matter of eliminating duplicate code as part of refactoring. It's worth noting that John Lakos makes a good case for the use of duplicate code!

Any team adopting XP would be well advised to prepare for these battles.

Collective Ownership

Collective ownership seems like a good idea but again I have some worries about it. Could it actually be an excuse for no-ownership? Where an entire team owns a product that is a good thing, when the new version ships the whole team have something to celebrate, if flaws occur then its a flaw in *the team's baby*.

But when it comes to collective ownership of the entire code base I see several problems:

? Some areas of code can be complex. Particularly where a knotty business problem is involved. Here it is quiet natural that one or two people become more involved with this aspect of the project. Sometimes they can end up understanding the problem better than the customer. Are we to hold back developers because they are moving ahead of the pack? Are we to demand that customers explain the same functionality to every developer on the team, even if this means them explaining it four, five, or even six times?

? Collective ownership can also be an excuse to ignore issues in the code. Suppose we have a bit of ugly code, suppose we have a real hack somewhere – which is quiet possible because some piece of simple design grew like Topsy – then who is going to sort it out? You may find that each of the four collective owners has a higher priority piece of work, and after all "if it works why fix it?" Knowing that if a change comes up the developers only have a one in four chance of needing to change that piece of code the temptation is to play Russian roulette. A developer who has to live with such a piece of code and answer questions about it is far more likely to try and take preventative action.

I think some degree of collective ownership is a good thing. It's worth having an under-study for each developer[3] - if only so we can take holidays! - spreading knowledge round is a good thing but I think the idea that every developer is responsible for every line of code is, erh, too extreme.

[3] Fred Brooks talks about a co-pilot who works with the surgeon (don't blame me, Brooks mixed the metaphors!) – the co-pilot is not responsible for the work but can take over if need be.

This is one of the key points of XP which limits it's scalability. If every developer must be able to maintain every line the maximum number of lines in the project is equal to the number of lines the least able developer can handle.

Onsite customer

Lawyers have been trying to write precise English which covers all situations for a few hundred years and yet we still have arguments in the courts about what the law means. Actually having a real live customer who you can just ask things of is a real boom. However, this requires a change in how the company operates and bills. If every time the customer tells you something you have to raise a change request you will not get any where. You must also be able to have a frank discussion with a customer – I remember one project where after a discussion with the "architect" I went talk to an onsite customer about a problem but had to ask questions so that they did not become aware that we'd uncovered a major deficiency!

If companies really want use an on-site customer they often have to change their way of thinking and billing. This may actually be impossible unless the developers and customers actually work for the same company.

Production is the normal state

Most of my professional career has been spent in a production state. I have no problems with this rule, it imposes a certain discipline and is not without it's draw-backs. For example, you may not get the time to perform a major refactoring, or, to explore a new technology which is potentially useful[4] Sometimes this is not possible: I recently worked on a system where we had six months to prepare for a new software release: both our software and the clients went live on the same day, releasing ours before theirs was pointless, but to release later carried a very real prospect that customer would use a competitor.

People and teams

Beck advocates a strict 40 hour week[5]: Developers are, to some degree, brought up in a culture of long hours, all night hacks, and racing deadlines. Sometimes, these are necessary, sometimes, just sometimes they can make a difference. On the whole though, doing much more than 40 hours a week on a regular basis takes its toll. On the subject of holiday's he's right again, a break for a week or two can do wonders for the creative process. Just because I'm away from the code-face doesn't mean my brain is diminishing, if anything, I've used these opportunities to look at my problems in a different light and seek inspiration.

---

[4] McCarthy answers this problem by proposing "Scouts" : developers who are allowed to move away from production development for a period of time to explore a new technology or experiment with a different approach.

[5] Maguire advocates the same policy.

Beck talks of the importance of a team which fits together, he's right about this, a team which thinks alike, shares a common vision and respects one another will be an order of magnitude more productive than a team of individuals – the whole being more than the sum of the individuals and all that.
However, Beck seems too quick to fire people from the team, his method of getting a team to gel is to fire the people who don't fit. This is the wrong way of approaching the problem, albeit, it is extreme! Not only does much employment law prevent this it's not the right way to treat people. Sometimes it's the only way to deal with a situation, but generally, if you go round firing people who don't fit your practising management by terror. Beck doesn't talk about recruitment, surely it's better to hire people who fit in? Have the whole team involved in the interview process, is everyone happy with hiring this person?
Beck recognises that best projects are all seated physically close together, and sometimes the best way to achieve this is physically move the desks yourself. However, in the bigger companies I've worked for this just isn't on, period. Yes, companies with this attitude may deserve to fail.
Beck also advocates developers taking responsibility for the process: ownership of the process is an important part of owning the final product. But this is not without problems.
Some organisations have a process, and they can be very attached to the process, especially if confers ISO900X or CMM level N status to the organisation. Secondly, this could lead to navel gazing. Is the program broken or the process?
Small companies can usually be more flexible with this kind of thing while bigger companies less flexible, after all: you may not be the only development team in the organisation and what works for one may not work for another, large companies frequently prefer lowest-common-denominator approaches. Changing the process can bring you into all sorts of office politics which potentially kill the project before you start.
Finally here, I agree on his view on toys and food but I don't think you can legislate for them.
Moving the furniture, owning the process, playing with toys are all part of binding a team building and empowering. The shared vision / shared goals concepts which run through Beck and McCarthy's work is where I believe the silver bullet is be found.

Is it a methodology?

Somewhere in the book, Beck states that XP is likened to "observing programmers in the wild." As I said at the top, 75% of the book has me saying "I've done that" so I agree with this comment.
However, simply observing and documenting does not a methodology make.
This is by big problem with XP as it stands. I feel it's a bunch of observations which Beck has tried to fit into an over reaching methodology. The glue which hold this together are his tales an anecdotes.
I can tell a good tale myself, and I've been around software long enough to have a lot of anecdotes. I could also try to come up with my own philosophy based on this. Beck and I are not unique here, I think most developers could, Beck and I differ from most because we're prepared to put it on paper and stick our necks out.
But I don't think this makes a methodology.

In Jim McCarthy gives "54 rules for delivering great software." I'd much of preferred Beck's thoughts if they where presented in a similar fashion. McCarthy speaks a lot of sense and injects a lot of ideas but he doesn't try to elevate this to a methodology. Steve Maguire takes a less itemised approach to McCarthy but has a similar approach. As Maguire and McCarthy's books come out of the same epoch of Microsoft history they are quiet complementary but they never claim to be a methodology with the answers I wish Beck could of taken more of an item by item approach, rather than try to capture the intellectual high ground of a methodology.

(I picked up a copy of *The Pragmatic Programmer* at JaCC and although I have only just started to read it, hope it can be added to this short list.)

One lesson I don't find in XP explained but forms part of my personal lore is that "methodologies don't work", or rather, they don't work straight out of the box, they must be adapted to any given environment.

Get out clauses....

I don't think anyone will ever prove that Beck's XP works, or doesn't work. Beck has provided too many "get out clauses" to prevent it being quantified. XP will appeal to the hacker in us all – if we can just loose those pesky unit tests! These get out clauses include:

- Chapter 23: "the full value of XP will not come until all the practices are in place.... you can get significant benefits from the parts of XP... there is much more when you put all the pieces in place" : in other words, if you measure an "XP" project and it doesn't show the claimed benefits it may just be you missed some little bit. (I never worked out how squares this with the preface which says "XP is lightweight... low-risk, flexible...." if it is lightweight and flexible why must I implement 100%? And how can that possibly be low-risk?)
- Chapter 24: "accepting responsibility for the process" sometimes you don't have control of the process. Sometimes you are not allowed to change it – ever worked for an ISO9001 approved organisation?
- Chapter 25 is full of "when not to use XP" : I feel like I'm reading a tautology, you know the "this is not true" type of thing. I think Beck sets out to define a very narrow domain space for XP and of cause, if you set the parameters so tight it's going to work.
- So much of XP seems to centre on Kent Beck and/or Ward Cunningham's personality and personal style : I'm sure they are great people to work with and real thinkers but I can't help thinking that makes it difficult to really duplicate it in other teams[6].
- XP is intended for small to medium projects. I hate to say this, but these are not the ones where you need lots of methodology, the real test of a methodology is large projects. Having said this, I actually think large projects are impossible and should not even be attempted – another rule of my lore "inside every big project is a small project struggling to get out."

[6] Kevlin Henney suggested on the ACCU-General mailing list that perhaps every XP team needs a Ward or Kent, I think he may be right here, maybe XP as described is a Beck/Cunningham thing, but other teams can do similar things if a capable leader emerges.

How applicable must a methodology be to be worth of the name? By placing all these restrictions on XP is it a methodology or just a recipe for Beck's own projects?
Beck also includes a get out clause to dismiss those, like me, who are frightened by XP: in the preface he acknowledges this and states that these are all old techniques. He's right of course, but nobody has taken them to such "extremes" before, many things which are good are bad when taken excessively never mind taken to extremes.

Will XP work?

I think there are a some circumstances where XP will work. I don't think there is anywhere that XP will work exactly as Beck describes it, unless of course, you hire Beck and Cunningham. Given his get-out-clauses I think this makes it almost impossible to reproduce and measure XP.
In the financial sector it is very common to have developers and customers working in tandem. I have literally see trading floors with developers sitting next to traders. The developer codes something, and when it does what the trader wants it's released[7]. It is possible to have a team of developers working with a team of traders. The pressure on developers can be intense, and it will never satisfy the 40-hour week rule but I think it satisfies most of the other points: always in production, simplest possible design (because they will throw most things away), coders all being familiar with the code (because there is not a lot of it), etc.
I can also see some small consultancy teams working in this fashion. The parachute in type consultants who arrive with their kit-bags, and set up shop for six months in a strange town. Again, while they may only work 40 hours a week, since they spend 4 hours travelling on Monday morning and Friday afternoon they must do long days.
What worries me more is that some consultancies will believe that XP is another methodology they must provide to their clients; so a few people will "learn XP" and it will sit on the list of available methodologies next to RAD, SSADM, Prince and the ISO9001 badge. The same managers will be sent XP and SSADM projects, as most of the actual developers are just hired guns these people will change, and hence the consultancy will miss the entire point of XP!
There are also places where XP will never work. Some organisations are change averse, if you have an organisation that will make you document every line of code you change then XP isn't going to work, the reliance on refactor means that organisations must accept the code base will change frequently.
Generally, I'd prefer to cherry-pick XP in the same fashion I cherry-pick McCarthy and Maguire's books. I don't think I'm alone here, I think others would agree with this.

It's all in the name

Originally I didn't believe extreme programming was extreme. It didn't make any sense to me, at best it sounded like hacking, "lets be extreme... let's _only_ code!". Having read the book, my opinion has

[7] Although I haven't read the Nick Leeson case in detail I understand there was an element of this in his ability to defraud and bring down Bearings Bank.

changed, there is something there, it is extreme, because "if something is good we do extreme, 100%, no prisoners, no compromise".
I don't think you can develop software like that. So much software development is about judgement calls, "do we optimise now or later", "do we take the refactoring hit now or later", "do I implement a 20% solution now to keep them working or bite the bullet."
But more than Extreme Programming as a technique, Extreme Programming is it's own God, trying to give a select programmers the kudos of joining the *Extreme Sports club*. XP would never of received the hype and publicity it was called "Beck Development Methodology", or "Chrysler style programming." Nor would the book of sold so many copies if it had the title "27 ideas to spice up your development" which I think would be a much more accurate title.
(Even the abbreviation panders to our culture fascination with the letter "X", X-programming like X-Files, or Microsoft X-box, X-wing bomber, XR3i, etc.)

Conclusion

So in conclusion I have come to believe that XP is all about XP, not so much about programming, more about a good name which someone thought up. Beck could of written a very interesting book about the Chrysler C3 team's experience, or about his experience in the financial programming world[8]. As a book about development I think it's good, as a bunch of tales with parables it's good, as a set of heuristics it's good.
But, I feel the name came first, and having decided to hang a methodology off this name Beck fails to present any conclusive and quantifiable results. So much of this book is a collection of tales, stories about him learning to drive, imaginary conversations between programmers, and so on.
Now that sounds negative, I'm sorry, I actually like the sound of XP. In some ways, Beck's book reminds me of Fred Brooks *Mythical Man month*: both present an *Emperors news clothes* view of current software development techniques.
Before I rush to adopt XP I want some changes: first, lets call the XP describe in Beck's book "Beck XP", second lets define XP as "programming centric development", putting developers in the driving seat if you like. Third, I'd like to relax the get-out clauses, XP must adapt to what ever environment it is in, after all "developers must control the process." With these modifications I'm happy with XP, actually I'm very happy. One final request: can we please recognise the books by McCarthy and Maguire as pre-XP books about XP?

References

? Jim McCarthy : Dynamics of Software Development, Microsoft Press, 1995

[8] Programming for financial markets is, I believe, a much neglected subject, The City in London, Wall Street in New York and countless financial institutions and software providers the world over service a market which is massive. Yet most of our documented software projects are drawn from, well computing itself. If anyone wants to collaborate on this give me a call, I've been round this market myself.

? Steve Maguire : Debugging the Development Process, Microsoft Press, 1994
? Steve McConnell : Code Complete, Microsoft Press, 1993
? Fred Brookes : Mythical Man Month, second edition, 1999
? Kent Beck : Extreme Programming explained, Addison-Wesley, 1999
? Andrew Hunt and David Thomas: The Pragmatic Programmer, Addison-Wesley, 1999