Applying the Lessons of eXtreme Programming

Pete McBreen
McBreen.Consulting, Canada
petemcbreen@acm.org
http://www.mcbreen.ab.ca/

Abstract

Although eXtreme Programming has been explained by Kent Beck[1], there are many benefits to adopting eXtreme Programming (XP) practices in other development processes. The benefits of adopting the complete XP approach are outside of the scope of this paper, what is discussed here are the lessons that XP offers to other development processes.

This paper initially discusses the benefits available from adopting XP style unit testing on a project, and then moves on to identify useful lessons that can be learned from other XP practices. The paper concludes by asking questions about the nature of process improvement in software development and how we can make our software serve society.

1. Introduction

Extreme Programming is a relatively new, somewhat controversial development process that takes many known software development practices and "turns all the dials up to 10" [1]. Initial experience with applying XP style Unit Testing using the JUnit testing framework [2] lead the author to experiment with incorporating other XP Practices into projects, not as a lead in to adopting XP, but as a useful process improvement step.

Experience to date with this approach indicates that XP has many useful lessons that can be applied to other development processes. The main personal lesson that I have learned from all of this is that it really pays off to look at the personal development process [3] and how this interacts with the team process. It is as if XP has it's roots back in the early work of Jon Bentley in his Programming Pearls books [4], it encourages us to think about what we are doing rather than just blindly following a process.

This paper initially looks at the effects of applying XP style Unit tests, and then explores other XP practices that have proven to be of value in other development processes. The paper concludes by asking questions about the nature of process improvement and how we can continually strive to make our software serve society.

2. Applying JUnit

JUnit is a deceptively simple testing framework that can create a major shift in your personal development process and the enjoyment of programming. Prior to using JUnit, developers typically have resistance to making changes late in a project "because something might break." With JUnit the worry associated with breaking something goes away. Yes, it might still break, but the tests will detect it. Because every method has a set of tests, it is easy to see which methods have been broken by the changes, and hence make the necessary fix. So changes can be made late on in a project with confidence, since the tests provide a safety net.

***Lesson**: Automated tests pay off by improving developer confidence in their code.*

In order to create XP style Unit Tests however, developers need to make sure that their classes and methods are well designed. This means that there is minimum coupling between the class being tested and the rest of the classes in the system. Since the test case subclass needs to be able to create an object to test it, the discipline of testing all methods forces developers to create classes with well-defined responsibilities.

***Lesson**: Requiring all methods to have unit tests forces developers to create better designs.*

It is interesting to compare classes that have unit tests with those that do not have unit tests. By following the discipline of unit tests, methods tend to be smaller but more numerous. The implementations tend to be simpler, especially when the XP practice of "Write the Unit Tests first" [5] is followed. The big difference I notice is that the really long methods full of nested and twisted conditional code don't exist in the unit tested code. That kind of code is impossible to write unit tests for, so it ends up being refactored [6] into a better design. As Kent Beck has reported elsewhere "Testing First Makes the Code Testable" [7]

***Lesson**: Design for testability is easier if you design and implement the tests first.*

Adopting XP style unit testing also drastically alters the minute by minute development process. We write a test, compile and run (after adding just enough implementation to make it run) so that the test will fail. As William Wake notes "This may seem funny - don't we want the tests to pass? Yes, we do. But by seeing them fail first, we get some assurance that the test is valid." [8] Now that we have a failing test, we can implement the body of the method and run the test again. This time it will probably pass, so now it's time to run all the other unit tests to see if the latest changes broke anything else. Now that we know that it works, we can now tidy up the code, Refactor as needed and possibly optimize this correct implementation. Once we have done this we are ready to restart the cycle by writing the next unit test.

***Lesson**: Make it Run, Make it Right, Make it Fast (but only if you need to).*

Early successes with JUnit encouraged me to experiment with other XP Practices. Just having the unit tests in place made programming fun again, and if that practice was valuable, maybe other were as well.

3. Useful lessons

This section details the major lessons learned from applying various XP practices on projects.

3.1 Be intolerant of process variations and working off process

I learned this lesson the hard way. While working on a C++ project using CppUnit, Michael Feathers' C++ implementation of JUnit[9], we mistakenly made writing unit testing optional. The people who used unit tests spent far too much time chasing down bugs in the code that didn't have unit tests. Eventually we managed to get automated functional testing implemented for the system, but this didn't help that much. Yes, we now had proof that the application didn't work, but we didn't know what was broken or how to fix it. Well we did have some idea, the 25% of the code that had unit tests was never broken for long, since the failing unit tests highlighted the methods that needed fixing. It was the rest of the code where we wasted lots of time running debuggers and logging intermediate values to the console -yeuch! At last I knew why one of the rules in XP is "*You are welcome to use your style. Just not on our project*" [10]

With good unit tests, debug time is almost non-existent. But this is only true if we have tests for ALL of the code, so all developers must develop using this style. As Alistair Cockburn has pointed out in his Software Development Manifesto, software development is a team game[11]. In an effective software team, everyone follows the same process and standards. Indeed, until the team aligns on these, they are not a team, just a bunch of developers muddling through.

***Lesson**: Effective working together on a process requires we all work the same way.*

This is not to say that there is no room for creative genius and experience, just that developers need to choose to express their individualism in things other than their coding style. Resistance is, however, inevitable. Most (all?) developers have had the unfortunate experience of being told to follow an unworkable or ill-defined process or standard. This is where the "High Discipline, Low Ceremony" [13] part of XP shines through in the intelligence with which the rules are applied, since after all, "they're just rules" [12], and the rules can be changed.

***Lesson**: Rather than work off process, change the process to make it workable for the team.*

So if a process or standard is broken, fix it. Do some carefully controlled experiments to investigate alternatives, and let the team choose the most appropriate option. In this way the process and standards get debugged really quickly, since a team of vocal, empowered developers will soon fix the problems in any process that they are required to follow.

***Lesson**: Detecting broken processes is hard if people are not required to follow the process.*

3.2 Use a coach to keep the team on process and improve individual skills

Given a choice between following a process and doing whatever we feel like, few humans follow the process. This is a normal part of being human. Unfortunately, one possible result of this is "shelfware" methodologies, high ceremony, with lots of formal deliverables and sign-offs. The grim little secret of many projects is that they continually have to spend time going back to create deliverables for sign-off purposes, and none of these deliverables have added any value to the overall development of the system. One company has even gone so far as to create a process for working "off-process", and it is not possible to find a single project that follows their official methodology.

XP has taken this to a different extreme. Rather than have lots of things that the team is "*supposed to do*", instead it mandates a minimum number of things that the team "*has to do*". So instead of having a high ceremony, low discipline process (that nobody follows), XP has a low ceremony, high discipline process that everyone follows. Compliance with the process is achieved in XP by having a coach whose main jobs are to keep the team on-process and to spot when the process needs to be adjusted.

***Lesson**: Nominate a coach to keep the team on process*

With a coach as part of the team, there is always someone with the responsibility of ensuring that everyone is working at the top of their game. (Sports teams have managers and coaches in case anyone thinks that this is the manager's job.)

Although many developers are resistant to this idea, there are two groups of developers who almost always are open to it: developers at the start of their career and really good developers who have the ability to make a team really "jell". Interestingly enough, when a team gets mentoring (typically because they are learning a new technology), most of what the mentor does is be a coach for the development team. The main problem with mentoring is that developers rarely get enough mentoring in their development process and the people skills side of software development.

For those rare developers who fail to integrate into a team, an important role for the coach is to ensure that this person is transferred to another team where they can contribute their gifts. Over the years, too many authors to mention have pointed out the benefits of teamwork and the hazards of a "Lone Ranger" developer. In a "highly tuned" process like XP, the effects of a Lone Ranger are detectable earlier, but the outcome is the same of all processes; the team is less effective than it would be without the Lone Ranger. Most managers know a project where they wish they had removed somebody sooner than they did.

***Lesson**: If you do not want to be part of the team, don't work on the project.*

3.3 Apply the Quality First Strategy

XP has been the source of many interesting sayings, and Don Wells' VCAPS project [14] provided one that illuminated a fundamental difference between XP and Traditional processes. JangIt - "Just Add New Garbage Ignoring Tests" highlights the XP mindset that unless the code is supported by extensive, relentless testing, the code is almost certainly useless. Thus in XP, there is always a focus on small increments, so that the minimum calendar time elapses before the code can be tested by real end to end functional tests.

Which is more useful to a project. One feature that passes all functional tests and is supported by unit tests, or ten features that just need to be debugged and tested? I see no contest here at all. Proven quality beats untested code every time. Hence the focus in XP is on getting something running and keeping it running correctly as new features are added. This is close to the advice that Fred Brooks gave "One always has, at every stage in the process, a working system. I find that teams can *grow* much more complex entities in four months than they can *build*" [15].

***Lesson**: Make sure the system always runs and passes all automated tests.*

Unless you enjoy running the debugger and reading through core dumps, never add any new functionality until all of the existing functionality works correctly. Quality First works, it's practically impossible to debug your way to a working system. A corollary to this is that developers should never, ever check in code unless ALL of the unit tests pass. Rather than institute penalties for "Breaking the Build" as some large corporations have done, change the process to ensure that the master source only ever contains a version that works.

***Lesson**: Only check in code when all tests pass.*

3.4 Continuous Integration is the only way to avoid Integration Hell

A core part of the Quality First Strategy is to make sure that the entire system works, not just your small part of it. Life is far too short to waste time trying to debug your way out of Integration Hell, so keep the system integrated all of the time. After all, we can scarcely claim that the system works end to end if we haven't connected the ends yet. Painful experience on projects has taught me that whichever parts of the system we fail to integrate early are the ones that cause the big schedule surprises.

***Lesson**: Keep the system completely integrated and automate the tests that verify it works.*

3.5 Tired humans produce lousy software

Despite what we may have thought we learned at college, humans do not do their best work in the small hours of the morning. An early XP project had a rule that limited developers to a Forty Hour Week "Overtime is time spent in the office when you don't want to be there. We think overtime is bad."[16]. It is as if collectively we do not understand the real nature of software and what it takes to create really great software. Howard Baetjer's view is that "Software is embodied knowledge"[17], and to do the creative, mental work that this entails requires minds that are fresh and alert.

***Lesson**: You can always try to sprint, but this is a long distance race, so use an even pace.*

3.6 Incremental Development requires Incremental Requirements Capture

An interesting feature of the Human 1.0 operating system is that Humans work better with closure. What this means is that they need to see the fruits of their labor, and the more immediately visible the results are the better. This is probably one of the reasons that Fred Brooks noted that ***Growing*** a system was easier than ***Building*** a system. When growing a

system developers get a sense of closure from seeing the added features work, whereas when building a system, initially progress is only visible by looking at the ever larger documentation and design models. XP does a good job of giving the team closure by using 2-3 week increments, changing the conversation from "software development" to "software delivery".

***Lesson**: Frequent, incremental delivery allows a team to learn how to deliver software.*

In order to build in this frequent delivery cycle, teams must learn how to incrementally capture requirements as well as incrementally develop software. Without this the team will spend a lot of time before they can deliver the first increment, and hence run the risk of falling asleep in analysis paralysis. Overall a win for everybody, all achievable by separating the concepts of Capturing Requirements for Prioritization from Capturing Requirements for Implementation. XP does this by User Stories, but the same effect can be achieved by initially only capturing low precision use cases until after the business has made the necessary prioritization decisions.

***Lesson:** Prioritization of requirements is faster than getting all implementation details.*

3.7 Simple Designs are easier to maintain and evolve

Simple design is one of the key tenets of XP. The way that this works is that developers are constrained to short design sessions following which they must prove their design ideas out in running, tested code. For really tough technical challenges, developers are encouraged to take a day or so to create a throwaway prototype or "Spike Solution". Then once the issues are know, use what was learned to create a simple production design. A good simple design passes all tests, expresses the intention clearly, contains no duplication and requires minimal of classes and methods[18].

***Lesson**: Simple does not mean short, simple means straightforward and easy to understand.*

This definition of simple design is supported by XP style Unit Tests, since complex designs are harder (impossible?) to test. The initial design only has to be good enough to make the code run, after that, testing can make sure that the design is correct. Refactoring is then used to make the design is right. This refactoring is essential to reduce Technical Debt[19] and hence maintain the current project velocity[20].

***Lesson**: The longer you wait to Refactor, the harder the task, so don't delay, do it today.*

Another part of simple design is that of ensuring that the simple design is still simple when we look at the code. Well factored code that uses intention revealing method names[21] does not need comments, indeed comments are a sign that the author of the code thought that they needed to explain what the code is intended to accomplish.

***Lesson**: Don't comment bad code, fix it!*

With simple designs and well factored code, maintenance and evolution of a system are much easier because the developers can understand the system. New features can be added by composing sequences of existing methods and by adding the few new methods and classes that are needed. It is rare to have to rewrite and change an existing method in order to add new functionality.

3.8 Adjust your development process slowly and measure the effects of the change

For most projects, it is better to have a less process and ceremony that everyone follows than it is to have more process that is only occasionally used. In most cases it is not that the process is broken, it's just that people are not following the process. This is where the XP practice of Pair Programming really pays off. Working by myself, I might "forget" to write a test for a method, but I'm sure not going to let you forget when you are working with me.

By paying attention to the minute by minute process of software development, XP has managed to raise the level of the entire development process. Rather than needing formal design

reviews and signoffs, by encouraging continual reviews and communication between developers and customers, quality and productivity are simultaneously improved. Best of all, XP has managed to put the fun back into software development. One of the leading indicators that can be used on any project is whether people are smiling, or as Eric Raymond[22] said "Enjoyment predicts Efficiency".

***Lesson**: Software Development is meant to be fun, if it isn't the process is wrong.*

When changing any development process, make small changes and observe the results to see if they match the predicted outcome of the change. In order for these measurements to be valid, the team must follow the new, experimental process with high discipline. This is the only way that we can successfully apply the scientific method to software development processes. The other part of this is that the experiment needs to run long enough for the expected results to appear. This is another reason why XP is popular with developers. All of the practices are geared towards immediate or near term results. The effects of non-compliance show up quickly so the team can correct itself quickly. In contrast, many other processes allow activities to be pushed towards the end of a project where they can conveniently be ignored due to schedule pressure. Small wonder then that many projects ship untested, unmaintainable systems.

***Lesson**: Humans ignore future costs in favor of immediate, short term gains.*

3.9 Many projects would benefit from a good dose of Reality Therapy

Communication is an essential part of all XP projects. Many mainstream projects fail due to lack of communication. Most XP practices provide a means of communication between developers, the customers and the system that is being created. The unit and functional tests provide a healthy dose of reality to the discussion about "when will you be able to deliver this feature?" It's hard to claim to be "nearly done" when none of the functional tests pass.

***Lesson**: Hard numbers are needed to prevent the "90% done for 90% of the time" problem.*

A key part of Reality Therapy is the XP practice of "Business people make business decisions, Technical people make technical decisions." Realistically, the business can either choose scope or the delivery date, or as Ron Jeffries[23] has stated, Resources, Scope, Quality, and Time are the dimensions of a project, "Projects under stress often squeeze quality because no one can see it happening." So when the business people select a scope, the technical people get to say when that scope can be delivered, or if the business chooses a date, the technical people get to say how much scope can be delivered in that time. Using short development increments (2-3 weeks) then gives direct feedback about the actual project velocity.

***Lesson**: Defend against schedule pressure by making project velocity public knowledge.*

With a known project velocity it becomes important for developers to actually follow business priority when implementing features. Hence the need for incremental requirements capture and simple designs (after all, why spend time now allowing for a feature that might be needed in the future when there are other, more valuable features to work on right now). Developers get their own dose of Reality Therapy at the end of an increment if they do attempt to design for future features when they get to explain why the project velocity has dropped.

***Lesson**: How does a project get to be a year late? ... One day at a time. [15]*

4. Improving the software development process

When reading about the role of Coach in XP, it is easy for developers to make the assumption that Big Brother will be watching them. My understanding of this role is different. The real purpose of the coach role is to have someone always paying attention to process improvement. However, as part of that role, they need to be able to communicate directly with team members

who are following sub-optimal development practices. My experience has been that many software teams effectively operate without any adult supervision on a day to day basis (and few of these operate effectively). Just having someone on a project always pushing for process improvement can have dramatic effects on the productivity of the team.

***Lesson**: Staying awake pays off.*

A key part of staying awake is ensuring that the team brings all of themselves to work every day. This ensures that they can apply all of their intelligence to the software they are creating. Although XP practices support this to some degree, within the context of "Software Serving Society', I think that XP is not extreme enough. As mentioned earlier, XP practices a division of responsibility for decisions "Business people make business decisions, Technical people make technical decisions." While this is good as far as it goes, it makes the assumption that there are only two types of decisions that matter on a project, business and technical decisions.

In order to improve the software development process we need to consider the complete social and ecological system before we can determine whether our software is really serving society. In doing this we have to look beyond the systems we are developing to look at the overall impact of computer technology on society[24]. XP is a good starting point for this in that it focuses attention on people and their experience in developing the software. XP encourages developers to question their development process, now as developers we need to start questioning the systems we are building.

References

[1] Kent Beck, "Extreme Programming Explained", Addison-Wesley, 2000
[2] Erich Gamma & Kent Beck, "Test Infected" http://members.pingnet.ch/gamma/junit.htm
[3] Andrew Hunt & David Thomas, "The Pragmatic Programmer", Addison-Wesley, 2000
[4] Jon Bentley, "Programming Pearls", Addison-Wesley, 1985
[5] Ron Jeffries, "Unit Tests" http://www.xprogramming.com/Practices/PracUnitTest.html
[6] Martin Fowler, "Refactoring : Improving the design of existing code", Addison-Wesley 1999
[7] Kent Beck, "Testing First Makes the Code Testable" http://www.extremeprogramming.org/stories/testfirst.html
[8] William Wake, "The Test/Code Cycle in XP" http://users.vnet.net/wwake/xp/xp0002/index.shtml
[9] Michael Feathers, "CppUnit C++ Ver 1.5" http://www.xprogramming.com/software.htm
[10] Ron Jeffries, "Coding Standards" http://www.xprogramming.com/Practices/PracCodingStandards.html
[11] Alistair Cockburn, "Manifesto for software development" http://members.aol.com/acockburn/manifesto.html
[12] Ron Jeffries, "They're just rules" http://www.xprogramming.com/Practices/justrule.htm
[13] Martin Fowler, "UML Distilled", Addison-Wesley 1997
[14] Don Wells, "Lessons Learned" http://www.extremeprogramming.org/lessons.html
[15] Fred Brooks, "No Silver Bullet" in "The Mythical Man Month", Addison-Wesley, 1995
[16] Anon, "Forty Hour Week" http://c2.com/cgi/wiki?FortyHourWeek, Portland Pattern Repository 15-May-00
[17] Howard Baetjer Jr., "Software As Capital", IEEE Computer Society, 1998
[18] Anon, "Simple Enough" http://c2.com/cgi/wiki?SimpleEnough, Portland Pattern Repository 3-Apr-00
[19] Anon, "Technical Debt" http://c2.com/cgi/wiki?TechnicalDebt, Portland Pattern Repository 5-Mar-00
[20] D. Hinchcliffe, "Project Velocity" http://c2.com/cgi/wiki?ProjectVelocity, Portland Pattern Repository Apr-00
[21] Kent Beck, "Smalltalk Best Practice Patterns", Prentice Hall, 1997
[22] Eric S. Raymond, "The cathedral and the bazaar", O'Reilly, 1999
[23] Ron Jeffries, "Four Variables" http://c2.com/cgi/wiki?FourVariables, Portland Pattern Repository 18-May-00
[24] Computer Professionals for Social Responsibility, http://www.cpsr.org/