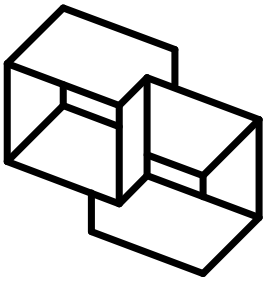
©2000 McBreen.Consulting

Practical Objects



Applying the lessons of eXtreme Programming

Pete McBreen, McBreen.Consulting
petemcbreen@acm.org

What is eXtreme Programming?

eXtreme Programming is a humanistic discipline of software development

XP is a ***deliberate*** and ***disciplined*** approach to software development

XP is *Low Ceremony* without much formality, but projects run with *High Discipline*

Although there are few ***prescribed elements*** the process is followed without anyone working "off-process"

XP has been used effectively by

DaimlerChrysler, Bayerische Landesbank, Credit Swiss Life, First Union National Bank, Ford Motor Company and UBS

eXtreme Programming, pictures are nice but tested code is better

eXtreme Programming is a developer scale, lightweight methodology based on four values

Ultra high frequency ***Communication***

Simplicity of both design and process

Feedback driven Activities

Courage - embrace change as a way of life, rather than build processes to limit change

When we come to realize that change is the only reality, life is much easier **- Anon**

XP is a challenge to the standard ***one size fits all*** software engineering view of the world

XP has many lessons that can be applied to software development projects in general

Steering **projects**

Using feedback to improve development processes

Separating Business and Technical responsibilities

Applying the *Quality First* strategy

Incremental development means *incremental requirements capture*

Big design up front **does not work**

Unit Testing **and really meaning it**

Make it run, make it right, make it fast

Avoiding *Integration Hell*

Refactoring **to improve design quality**

Experimentation requires a *high discipline process*

Use a process coach to tune and tailor your process

Steering Projects means that the project plan has to be a living document

eXtreme Programming uses a *Driving Metaphor*

Driving is not about getting the car pointed in the right direction

Driving is about constant adjustment

A little this way, a little that way, ***forever***

You always ***pay attention***

You are always ***prepared to change*** things

Unfortunately, the driving paradigm is at odds with the *car pointing paradigm* prevailing in much of the corporate world

Using feedback to improve the software development process

When driving, you need to keep your eyes open, so process feedback is essential

Track ***actuals against estimates*** and learn from variation

Make the tasks short so that feedback is immediate

Keep progress visible through ***graphs***

Use ***short increments*** so that the team can celebrate delivery every few weeks

Use ***Relentless Testing*** so that developers know find out immediately if they have broken anything

Have an onsite customer so that you never have to wait for a decision

Only did deep into your bag of tricks if a simpler solution has been proven not to work

Manager and Customer Rights

1. **You have the right to an overall plan, to know what can be accomplished, when, and at what cost**
2. **You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify**
3. **You have the right to change your mind, to substitute functionality, and to change priorities**
4. **You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date. *You can even cancel at any time and be left with a useful working system reflecting investment to date***

Developer Rights

1. **You have the right to know what is needed, via clear requirement stories, with clear declarations of priority**
2. **You have the right to say how long each story will take you to implement, and to revise estimates given experience**
3. **You have the right to identify risky stories, to have them given higher priority, and to experiment to reduce risk**
4. **You have the right to produce quality work at all times. You have the right to peace, fun, and productive and enjoyable work**

Business makes *Business decisions*, Developers make *Technical decisions*

Projects are driven by Business decisions...

***But* these business decisions must be *informed* by technical decisions about cost and risk**

So keep Technical people focused on Technical Problems and Business people focused on the Business

Projects fail* when either party always *"wins"

With the ***Business In Charge***, projects take on too much effort and risk without matching rewards

With the ***Developers In Charge***, projects take on too much technical risk and effort with too little return

A balance of power is needed

Use the four project variables to balance Business and Technical power

You ask your boss what he wishes you would do with the project. He answers, in his own words, "Use fewer people to get more done, increase quality, and get done sooner"

You're sorry you asked

***Resources, Scope, Quality, Time* - These are the dimensions of any project**

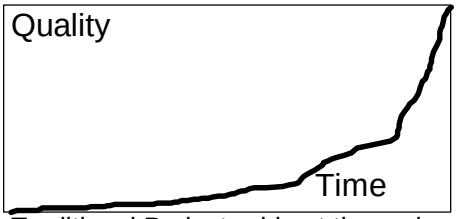
These are the four variables that control any project

1. Resources - who is on the project, the tools available
2. Scope - what the system is supposed to do
3. Quality - how good the system is
4. Time - how much calendar time has elapsed

The Quality First strategy falls out of the customer rights

Most projects "*add quality*" as time progresses

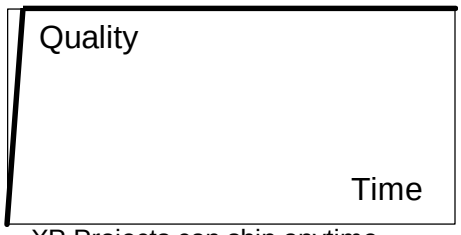
- Testing "***inspects quality***" into the software
- ***Integration hell*** is normal
- ***It's feature complete, time to get the bugs out*** ☺



Traditional Projects ship at the end

XP gets quality to 100% as early as possible

- The system always runs with zero bugs
- Developers **incrementally capture requirements** and add features



XP Projects can ship anytime

Minimizing the time between Requirements Capture and Delivery improves *feedback*

The time between *Requirement Identification and Delivery* may be significant

- This will reflect the overall resources committed to a project and the amount of parallel working that is possible

But the time between *Requirement Capture and Delivery* can be short *within an Increment*

- Reducing the effort of tracking ***Requirement Changes***, since in a short period, requirements do not change
- Changes may have occurred since ***Identification***, but the project is not responsible for tracking those changes
  - That is a ***Business Responsibility***
- The business also needs to track changes in Priority

Delaying Requirements Capture as long as possible results in an *improvable process*

Learning Research shows that the best learning occurs when there is rapid response

This means a short lag between Experiment and Feedback

The Business is *learning* how to explain what they need in the form of Requirements

The resulting functionality is their learning ***feedback***

Learning cannot occur if there is no delivery before all requirements are captured in exquisite detail

Developers are *learning* how the design works

Learning cannot occur if there is no delivery before all of the design detail has been "worked out"

Big Design Up Front does not work, since it reduces the opportunity for learning

***Big Design Up Front* works when little learning is required in order to deliver the system**

This sometimes occurs with ***experienced development teams*** and ***sophisticated customers***

But when there is significant learning involved, *Designs need to be tested* by running code

I've seen people on projects spend four weeks getting the pictures right for a design which half a day of coding could have convinced them would not work. -David Harvey

Big Design Up Front* is a return to the Waterfall development process, it is *pointing* not *driving

OO works best with an ***Incremental, Iterative process***

Unit Testing and really meaning it... *Testing everything* that could possibly break

Relentless Testing means that the Quality of the complete system is always known

The benefit of ***no integration surprises*** is only available if you have a complete set of tests that you always run ***no matter how small or insignificant the change***

Tests fall into one of two categories

Unit Tests are written by the Developers, ***so that they know that every method works as intended***

Functional tests are owned by the Business, ***so that they know that every requirement has been satisfied***

XP style Unit Tests focus on system ***Verification*** and the functional tests handle system ***Validation***

The *Write the Unit Tests First* strategy changes the feel of the overall process

Writing the Test Cases further *validates the design* and *adds precision to the Methods*

So when you code the Class, the ***Method Signatures*** are already specified and no more design is needed

The sequence is code the Tests, implement the methods, run all the tests, fix and retest

To ensure that the test cases are complete, provide minimal method bodies initially to ensure ***coverage***

The idea is to always take the smallest possible next step and never go too far before testing

Iterate round the ***Fix and Retest*** loop until all tests pass, then start the next increment by writing more tests

Developing using these *Nano-Increments* has a different feel to traditional programming

***Iteration is very rapid*, developers run the Unit tests every couple of minutes**

So practice ***running the tests after every change***

***Debugging is less common* because each method has it's own set of tests**

The test results point to where the problem lies

If you think you need to use the debugger, ***write a test case instead*** for the variable you want to inspect

Developers get a *real sense of progress* from the increasing number of test passes

There is no more uncertainty as to ***will it work or not?***

Make it run, make it right, make it fast

Ideally developers should *make the design executable first* by writing the code

Running the test cases will then identify any problems with the implementation or requirements

Then developers *make the design as simple as possible and correct*

Rewrite for clarity, to reflect what has been learned so far and to meet project standards

Only make it fast if performance *measurements* indicate that it will be too slow for use

Measure and profile to find where the system is spending time, and ***if possible apply a hardware fix***

Simple design is not a new idea, but simple is not always easy

There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R. Hoare, *Turing Award Lecture* 1980

Continuous Integration is a way of avoiding *Integration Hell*

The longer you wait to integrate, the longer it takes, and the more painful it is

THEREFORE: don't wait at all!

Microsoft integrates code daily to weekly

Unfortunately they do not have complete testing so all that they prove is that ***it builds***, not that ***it actually works***

eXtreme Programming projects integrate multiple times PER DAY

Take small steps and always ensure that the complete system still passes all of the tests

Deleting tests to make your code pass the tests will soon be made a capital offense

Refactoring is a process of *Restructuring Code* to support Design, Evolution and Reuse

Refactoring are *Behavior Preserving Changes* to the source code that improve *Code Quality*

Refactoring ***Transformations*** do not impact Functionality

***Coding* activities fall into one of two categories**

Modifying Functionality, adding features or fixing bugs

- Adding ***new methods*** to classes
- Bugfixing
- Adding support for a new subclass

Refactoring Code, making it clearer to allow another ***Episode*** of Modifying Functionality

- Moving misplaced features
- Simplifying long convoluted methods
- Simplifying conditional code

Refactoring relies on the ability to be able to *recognize code that needs restructuring*

The ability to detect *Code Smells* is developed with *Practiced Intolerance*

"*A **code smell** is a hint that something has gone wrong somewhere in your code*" - Kent Beck

As developers, *we are too kind to bad code*

Rather than fixing it up, ***we comment it***, making it likely that the practice will continue

Don't comment bad code, *fix it*

This is the lesson that can be learned from Refactoring, it is possible to create good, readable code

eXtreme Programming is about *Controlled Experimentation* in a disciplined process

Projects experiencing difficulties need to first check if they are still following the process

eXtreme Programming is a ***highly tuned methodology***, it is easy to fall off process and lose productivity

The XP Coach is a crucial role to ensure the team stays *on process*

Experimentation is only safe if the baseline performance is known and predictable

The Coach may have to employ a ***rolled up newspaper*** to ensure that people stay on process

Most companies take a relaxed approach to following a defined process

Yes, the project has a methodology, but the team is not required to follow it

For most projects, there is a big difference between the *espoused* and the *actual* process

XP is a processes that developers can actually follow

Few developers can follow a high discipline process, *people are not built that way*

To get compliance, extra mechanisms are required

A process coach is the way XP tackles this

Controlled Experimentation is how a Coach applies the *Scientific Method* to software

The whole XP process is designed to get clear, concrete feedback about the process

XP has definite rules, but they are *just rules*

Developers are required to follow the rules, or change the rules

Nobody is allowed to work ***off process*** when working on the main production source code

Spike Solutions and prototypes are encouraged

Changes are only made when there is hard data from experiments to justify the change

The Coach needs to keep individuals on track with respect to the development process

This is a hard task that calls for people skills

It requires a balance of respect for the individual within the context of the overall team goals

The coach has to notice when things are not working and to get the project back on track

Experimentation is only required if the team is following the process and things are still not working out

Debugging the development process is hard

Especially since few teams capture any useful data

Closing Thoughts and Question Time

Although XP as a package is new, *most of the practices have been around for a long time*

Few projects pay enough attention to *coaching their developers to improve performance*

***Writing the unit tests first* results in higher quality, tested code and *productive* developers**

***Accurate estimating is possible* if you break down a deliverable into small enough tasks**

Remember that *software development is meant to be fun*, if it isn't, the process is wrong

Links and references

The eXtreme Programming Website
http://www.xprogramming.com/

Manifesto for software development
http://members.aol.com/acockburn/manifesto.html

Software Development as a Cooperative Game
http://members.aol.com/humansandt/papers/asgame/asgame.htm

Uncertainty in Software Development
http://www.cadvision.com/roshi/Uncertain.html

Incremental Requirements
http://www.xprogramming.com/xpmag/incremental_req1.htm