Extreme Programming, an agile software development process

Nigel Goddard

School of Informatics
University of Edinburgh

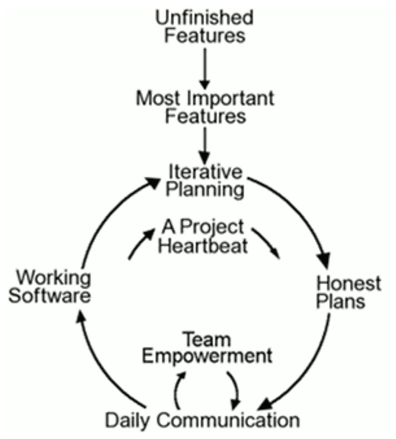
Agile processes

What the spiral models were reaching towards was that software development has to be *agile*: able to react quickly to change.

The Agile Manifesto `http://agilemanifesto.org`:

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

Agile flowchart



12 principles of Agile

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity- The art of maximizing the amount of work not done - is essential
- Self-organizing teams
- Regular adaptation to changing circumstances

Extreme Programming

One variant: Extreme Programming (XP) is

"a humanistic discipline of software development, based on values of communication, simplicity, feedback and courage"

People: Kent Beck, Ward Cunningham, Ron Jeffries, Martin Fowler, Erich Gamma...

More info: `www.extremeprogramming.org`,
Beck "Extreme Programming Explained: Embrace Change"

Risk: The Basic Problem

- schedule slips
- project cancelled
- system goes sour
- defect rate
- business misunderstood
- false feature rich
- staff turnover

"Use XP when requirements are vague or changing"

Key insight of XP

Traditional methodologies are that way because

the cost of coping with a requirements change or correcting a defect rises exponentially through the development lifecycle

– but what *if* it needn't be so? *Then* it is possible to be much more flexible.

Keeping that cost down is partly luck (i.e. being in the kind of project where it's possible to do so) and partly judgement (e.g., following those of XP's practices, like refactoring, which help to make it so).

XP classification of software development activities

- coding
- testing
- listening
- designing

Illuminating exercise: map these onto "standard" activities and contemplate implications of differences.

XP Practices

The Planning Game
Small releases
Metaphor
Simple design
Testing
Refactoring
Pair programming
Collective ownership
Continuous integration
40-hour week
On-site customer
Coding standards

The Planning Game

- Release planning game – customer and developers.
- Iteration planning game – just developers

Customer understands scope, priority, business needs for releases: sorts cards by priority.

Developers estimate risk and effort: sorts cards by risk, split cards if more than 2-4 weeks.

"Game" captures, e.g., that you can't make a total release in less than the sum of the times it's going to take to do all the bits: that's against the rules.

On-site customer

A customer – someone capable of making the business's decisions in the planning game – sits with the development team (maybe doing their normal work when not needed to interact with the development team), always ready to clarify, write functional tests, make small-scale priority and scope decisions.

Small releases

Release as frequently as is possible whilst still adding some business value in each release. This ensures that you get feedback as soon as possible and lets the customer have the most essential functionality asap. (May be talking about every week to every month – outside XP each 6 months would be more usual even in an iterative project, longer not uncommon.)

Metaphor

Is basically XP's word for part of what other people call architecture – it avoids the word architecture to emphasise that it doesn't *just* mean the overall structure of the system. "Metaphor" is intended to suggest an overarching coherence, easily communicated.

Continuous integration

Code is integrated and tested at most a few hours or one day after being written. E.g. when a pair wants to checkpoint they go to an integration machine, integrate and fix any bugs against the latest full build, add their changes to the central CM database.

Simple design

Motto: *do the simplest thing that could possibly work*. Don't design for tomorrow: you might not need it.

Testing

Test everything that could break. Programmers write unit tests using a good automated testing framework (e.g. JUnit) to minimise the effort of writing running and checking tests. Customers, with developer help, write functional tests.

Refactoring

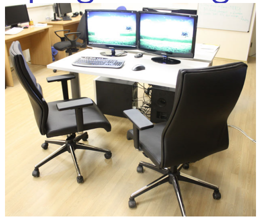
As we discussed before: but here refactoring is especially vital because of the way XP dives almost straight into coding. Later redesign is vital. A maxim for not getting buried in refactoring is "Three strikes and you refactor": For example, consider removing code duplication.

1. The first time you need some piece of code you just write it.
2. The second time, you curse but probably duplicate it anyway.
3. The third time, you refactor and use the shared code.

i.e. do refactorings that you *know* are beneficial

(NB you have to know about the duplication and have "permission" to fix it... ownership in common)

Pair programming



All production code is written by two people at one machine. You pair with different people on the team and take each role at different times.

> *There are two roles in each pair. The one with the keyboard and the mouse, is coding. The other partner is thinking more strategically about:*
>
> - *Is this whole approach going to work?*
> - *What are some other test cases that might not work yet?*
> - *Is there some way to simplify the whole system so the current problem just disappears?*

Collective ownership

i.e. you don't have "your modules" which no-one else is allowed to touch. If any pair sees a way to improve the design of the whole system they don't need anyone else's permission to go ahead and make all the necessary changes. Of course a good configuration management tool is vital.

Coding standards

The whole team adheres to a single set of conventions about how code is written (in order to make pair programming and collective ownership work).

Sustainable pace

aka **40 hour week**, but this means not 60, rather than not 35!

People need to be rested to work effectively in the way XP prescribes. There might be a week coming up to deadlines when people had to work more than this, but there shouldn't be two consecutive such weeks.

Mix and match?

Can you use just some of the XP practices?

Maybe... but they are *very* interrelated, so it's dangerous.

E.g., if you do collective ownership but not coding standards, the code will end up a mess;

if you do simple design but not refactoring, you'll get stuck!

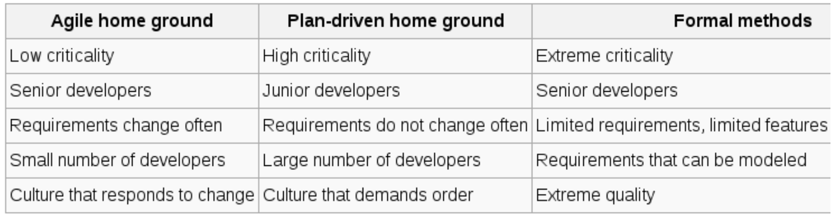
Where is XP applicable?

The scope of situations in which XP is appropriate is somewhat controversial. Two examples

- there are documentated cases where it has worked well for development in-house of custom software for a given organisation (e.g. Chrysler).
- A decade ago it seemed clear that it wouldn't work for Microsoft: big releases were an essential part of the business; even the frequency of updates they did used to annoy people. Now we have automated updates to OSs, and Microsoft is a Gold Sponsor of an Agile conference

XP does need: team in one place, customer on site, etc. "Agile" is broader.

Relating different processes



Agile home ground	Plan-driven home ground	Formal methods
Low criticality	High criticality	Extreme criticality
Senior developers	Junior developers	Senior developers
Requirements change often	Requirements do not change often	Limited requirements, limited features
Small number of developers	Large number of developers	Requirements that can be modeled
Culture that responds to change	Culture that demands order	Extreme quality