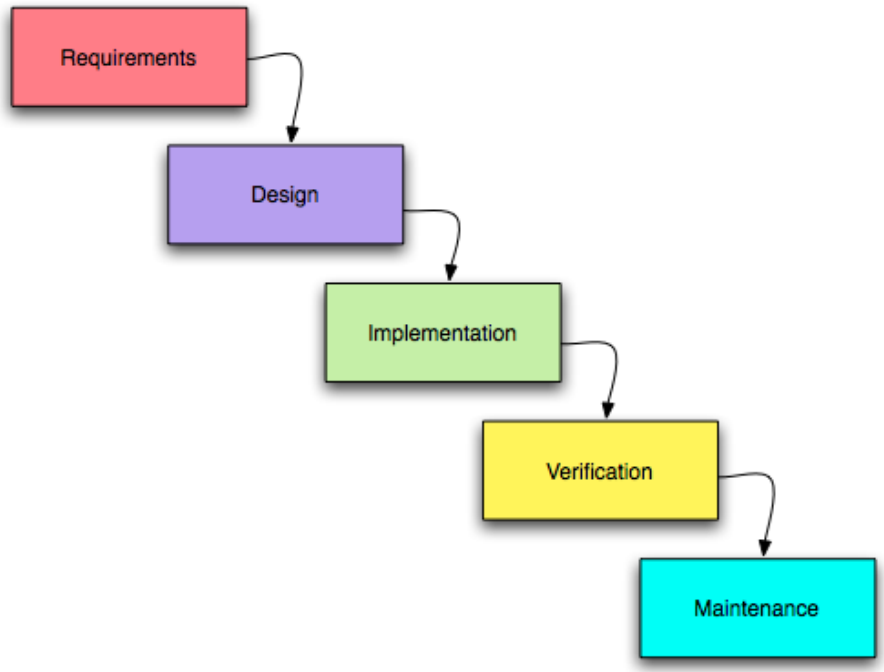
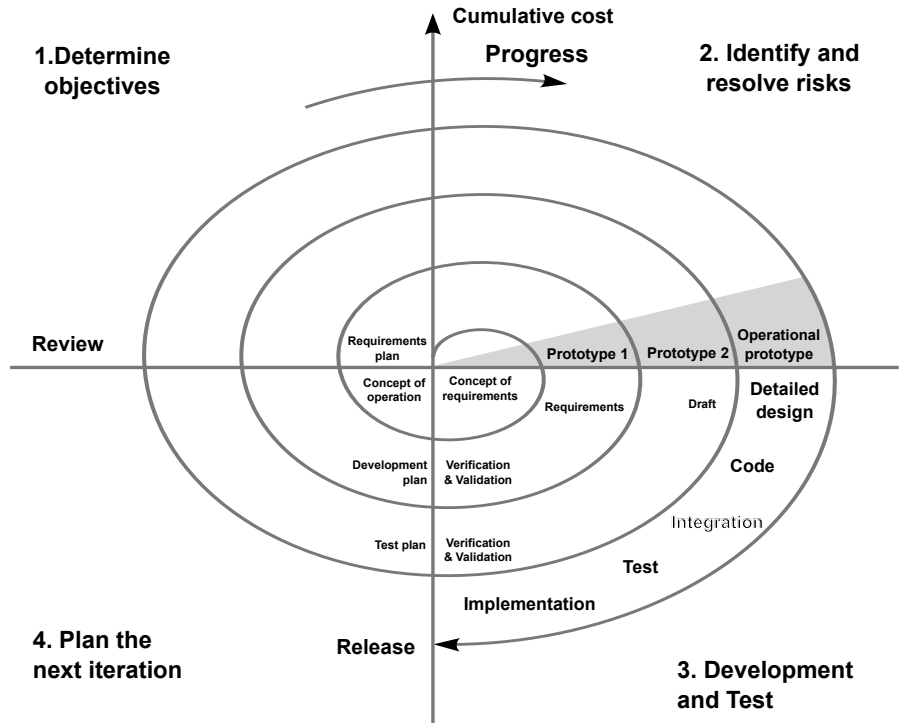
Extreme Programming, an agile software development process

Paul Jackson

School of Informatics
University of Edinburgh

Recall: Waterfall and Spiral Models





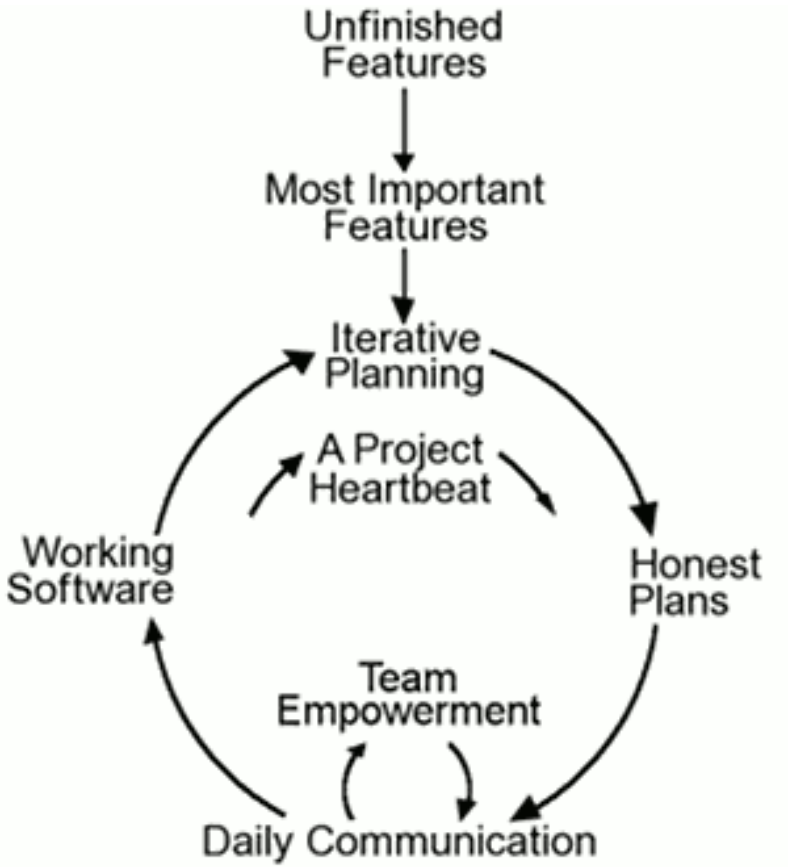
Agile processes

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
- ...

12 principles of Agile (continued)

- ...
- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted to get job done, given right support
- Continuous attention to technical excellence and good design
- Simplicity – the art of maximizing the amount of work not done – is essential
- Best requirements and designs from self-organizing teams
- Regular reflection on process and tuning of behaviour

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"

Example risks and the XP responses

- schedule slips:
  Short iterations give frequent feedback; features prioritised
- project cancelled after many slips:
  Customer chooses smallest release with biggest value
- release has so many defects that it is never used:
  Tests written with both unit-level and customer perspectives
- system degrades after release:
  Frequent rerunning of tests maintains quality
- business misunderstood:
  Customer representative embedded in development team
- system rich in unimportant features
  Only highest priority tasks addressed
- staff turnover:
  Programmers estimate task times; new programmers nurtured

XP classification of software development activities

- Coding
  Central. Includes understanding, communicating, learning
- Testing
  Embodying requirements, assessing quality, driving coding
- Listening
  Understanding the customer, communicating efficiently
- Designing
  Creating structure, organising system logic

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 always ready to

- clarify,
- write functional tests,
- make small-scale priority and scope decisions.

Customer maybe does their normal work when not needed to interact with the development team.

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
- lets the customer have the most essential functionality asap.

Every week or every month

Outside XP releases commonly every 6 months or longer.

Metaphor

- About an easily-communicated overarching view of system. E.g. *Desktop*
- Encompasses concept of software architecture.
- Provides a sense of cohesion
- Often suggests a consistent vocabulary

Continuous integration

Code is integrated, debugged and tested in full system build at most a few hours or one day after being written.

- Maintains a working system at all times
- Responsibility for integration failures easy to trace
- If integration difficult, maybe new feature was not understood well, so integration should be abandoned

Simple design

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

Testing

Any program feature without an automated test simply doesn't exist.

Test everything that could break.

Programmers write unit tests

- use 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

Refactoring is especially vital for XP because of the way it dives almost straight into coding.

Later redesign is essential.

A maxim for not getting buried in refactoring is "Three strikes and you refactor". 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 fresh, creative, careful and confident 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. Other Agile processes include Scrum and DSDM.

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

Reading

Suggested : *Extreme Programming explained: Embrace change*. Kent Beck.

Suggested : Sommerville.
Significantly more in 9th and 10th Eds (Ch 3). than 8th Ed. (17.1, 17.2).
Good for balance.