Think about it…

Luas Engineering Project:

What are the different phases in its lifecycle?

Why have different phases?

Problems/benefits of different phases?

Risks?

Software Engineering Projects:

What makes software engineering projects different from traditional engineering projects?

Lifecycles

A software project goes through a number of distinct stages from conception to decommissioning

"Software lifecycles"

- Bring some structure to software's evolution
- General organising principles of software development

In this lecture we'll look at some popular (and not-so-popular) lifecycle models

Stages - 1

Domain analysis

- The wider business context for the system

Requirements

- What the client wants built

Specification

- What the client is going to get
- Hopefully pretty close to the requirements…

Architecture

- The full system context

Design

- Each module, component, data structure and algorithm
- User interface (graphic design)

Stages - 2

Implementation

- Realise each module, component, data structure and algorithm
- Unit testing

Integration

- Bring the system together as a whole
- Whole-system testing
- Verification and validation (acceptance testing)

Operation and maintenance

- Bug detection and fixes
- New features, new platforms, new uses, …

What do we want in a lifecycle?

Should cover the whole story
- From conception through realisation to operation

Must reflect reality
- “Formally specify the whole thing from scratch, get the client to prove it correct, build it, then formally validate it” methods are probably not practical….
- ….although that’s not to say that formal methods aren’t useful in certain situations
- Many projects have to proceed without an expert mentor

Traceability
- Document and justify all the decisions taken

Models

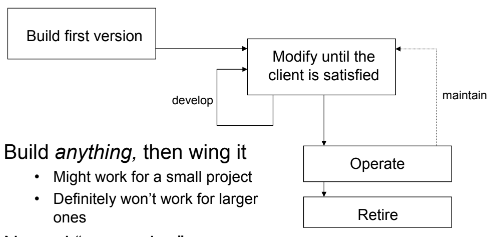
Over the years lots of models have been tried

We’ll look at seven
- Build and fix
- Xtreme Programming
- Waterfall
- Incremental
- Rapid prototype
- Spiral
- WinWin spiral

Increasingly better suited to larger and more complex systems

Each has its good points – there’s no “right” model for all situations
- Balance the complexity of the project against the complexity of the model to find the most appropriate approach

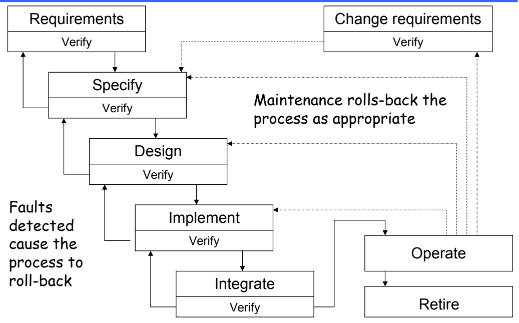
Build and fix



Build *anything,* then wing it
- Might work for a small project
- Definitely won’t work for larger ones

No real “game plan”
- Deficiencies propagate
- No documents = no maintenance

Waterfall model



Waterfall model - analysis

Not a bad model, really

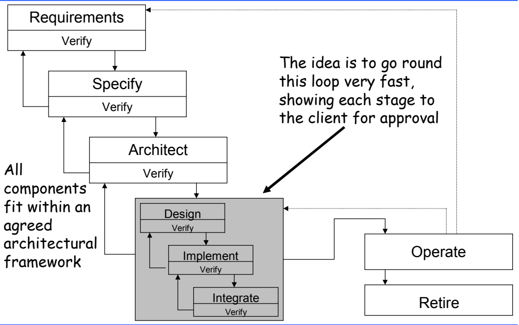
- Each stage gets verified before the process proceeds
- Faults cause the process to roll-back a stage (or more)
- Maintenance causes changes at some stage, and the same process is pursued from that point again

Main problems

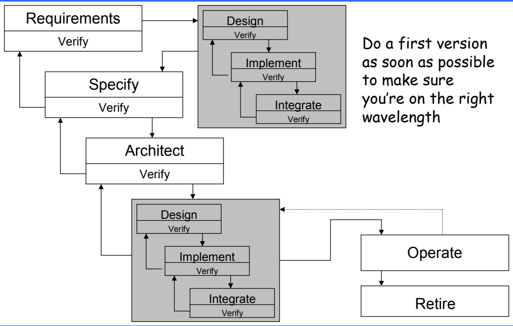
- Faults discovered late are *extremely* expensive to fix
- There's no requirement to show the client anything but documents until the end
  - good excuse for building a correct but useless system!!
- Why aren't all the problems found in verification?

Adequate for something you really know how to build

Incremental model


Requirements
Verify
Specify
Verify
Architect
Verify
Design
Verify
Implement
Verify
Integrate
Verify
Operate
Retire
The idea is to go round this loop very fast, showing each stage to the client for approval
All components fit within an agreed architectural framework

Rapid prototyping

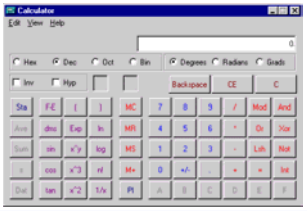


Do a first version as soon as possible to make sure you're on the right wavelength

What's in the rapid prototype?

The application painted with a broad brush

- Screen shots
- Key algorithms
- No fine details or finesse



No functionality behind menus

Full look and feel

Only basic maths

Comparison of lifecycle models

Model	Strengths	Weaknesses
Build and fix	Fine for short programs that will not require any maintenance	Totally unsatisfactory for nontrivial programs
Waterfall	Disciplined approach Document driven	Delivered product may not meet client's needs
Incremental	Maximises early return on investment Promotes maintainability	Requires open architecture May degenerate into build and fix
Rapid prototype	Ensures that delivered product meets client's needs	Needs lots of client involvement and enthusiasm Initial prototype cannot be used

Inspired by Stephen Schach, *Classical and object-oriented software engineering*, Addison-Wesley (1999)

Waterfall *vs* incremental *vs* rapid

Key idea is to get changes approved by the client on a short timescale

- ✓ Avoid mis-understandings propagating down the process
- ✓ Change can give the illusion of progress…
- ✗ Needs lots of client involvement and enthusiasm
- ✓ …but constantly having new things to show will help do this
- ✗ Getting the architecture wrong is disastrous – but that's true of any process

Rapid prototyping means you don't proceed without an implementation having been seen by the client

- Client knows what they'll get – or at least what it'll look like
- No route to later stages in the process except through concrete demonstration

However…

HEALTH WARNING

You MUST dump the first prototype

DON'T use it as the basis for the real development

The reasons

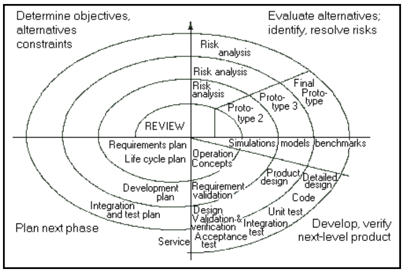
- Built without an architecture – badly designed (if at all)
- Lashed together in a hurry – badly designed (if at all)
- Features added willy-nilly – badly designed (if at all)

It's incredibly tempting to use it, but it's disastrous

- Essentially rapid prototyping then collapses to build-and-fix

Go back and do it properly

Spiral model

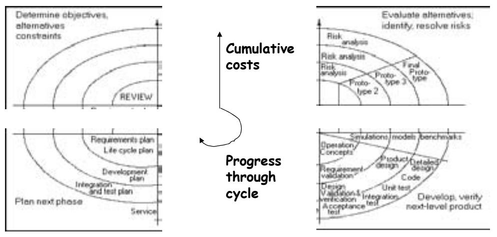


From Barry Boehm, *A spiral model of software development and enhancement*, ACM Software Engineering Notes (August 1986), pp.14-24.

Spiral model detail

Sector 1. Look for constraints, trawl thoroughly for alternatives

Sector 2. Assess the risks and, if acceptable, build



Sector 4. Decide what's to be done based on the maturity of the project

Sector 3. Examine the results and take any lessons to the next iteration

Spiral model - analysis

A bit of everything

- Rapid prototyping, incremental development, client involvement, plenty of documentation
- Maybe a bit *too* flexible for some peoples' tastes

Big on risk management

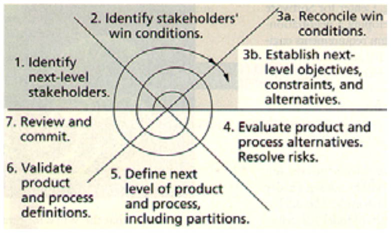
- No commitment without assessment – which at the very least passes the buck up to management…
- Most programmers aren't all that good at spotting risks
- Even if an unacceptable risk *is* detected, it may not be contractually possible to back out

Big on re-use – at all stages, not just code

- Plenty of opportunity to spot similarities between this and other projects

WinWin spiral model

Modification of the original spiral with added (!) activities


1. Identify next-level stakeholders.
2. Identify stakeholders' win conditions.
3a. Reconcile win conditions.
3b. Establish next-level objectives, constraints, and alternatives.
4. Evaluate product and process alternatives. Resolve risks.
5. Define next level of product and process, including partitions.
6. Validate product and process definitions.
7. Review and commit.

Get everybody involved to say how they win with the project, and make sure you explicitly account for these conditions

From Barry Boehm *et al*, *Using the WinWin Spiral Model: A Case Study*, IEEE Computer **31**(7) (July 1998), pp.33-44.

Tutorial

Read:

- Barry Boehm *et al*, *Using the WinWin Spiral Model: A Case Study*, IEEE Computer 31(7) (July 1998), pp.33-44.
- See course page

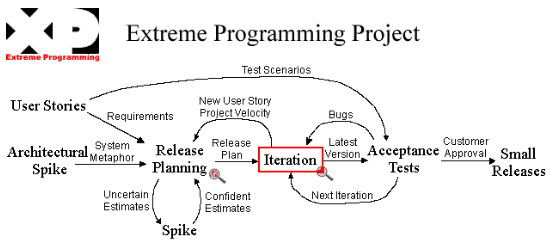
Extreme Programming Model

The Rules and Practices of Extreme Programming are broken into 4 main stages:

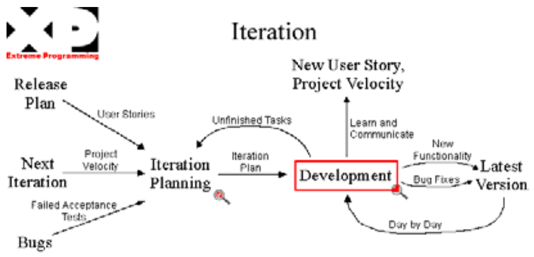
Planning
Designing
Coding
Testing

Reference:
http://www.extremeprogramming.org/rules.html

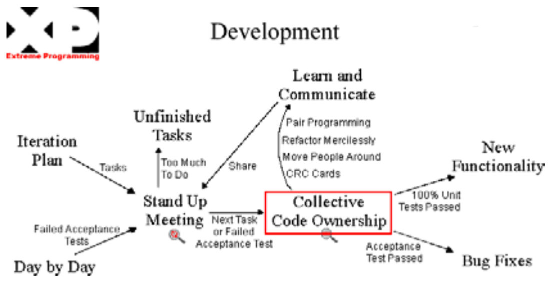
Overview of XP



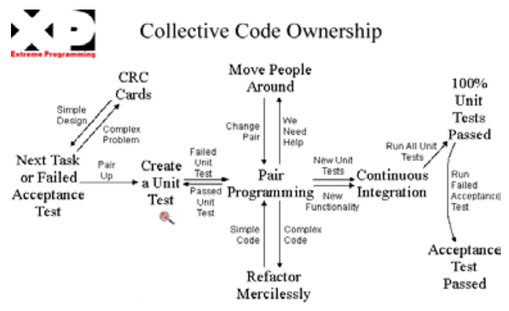
Iteration in XP



Development



Collective Code Ownership



Planning in XP

Write User stories

- Similar to Use Cases, but are used to create time estimates for the release planning meeting.

Release planning creates the schedule.
Make frequent small releases.
The Project Velocity is measured.
The project is divided into iterations.
Iteration planning starts each iteration.
Move people around.
A stand-up meeting starts each day.
Fix XP when it breaks.

Designing in XP

Simplicity.
Choose a system metaphor.
Use CRC cards for design sessions.
Create spike solutions to reduce risk.
No functionality is added early.
Refactor whenever and wherever possible.

Coding in XP

The customer is always available.
Code must be written to agreed standards.
Code the unit test first.
All production code is pair programmed.
Only one pair integrates code at a time.
Integrate often.
Use collective code ownership.
Leave optimization till last.
No overtime.

Testing in XP

All code must have unit tests.

All code must pass all unit tests before it can be released.

When a bug is found tests are created.

Acceptance tests are run often and the score is published.

So which lifecycle is correct?

Does everyone understand the goals?
- Do we need a prototype to get things straight?
- Do the users need to see what they might get in order to decide what they want?

How dynamic is the organisation?
- Will requirements change over the course of development?
- Will they *ever* be fixed?

Will the project be long-lived?
- Does it justify large-scale investment of time on management?

What level are the users?
- Can they do their own maintenance?

Is there substantial risk in decisions?
- Do we have the information to make them?
- What is their impact?

Summary

Different views of the software lifecycle
- Suited to different project complexities
- Structuring the process of development

Different views of what the main deliverables are
- Code? Design? Involved users? Justified risks?

Provide a lens through which to view and plan a project, but no real guidance on the individual stages

Next we'll look at the stages of the lifecycle in more detail, starting with domain analysis