Software Engineering CS3003

Lecture 2: Software Maintenance and Evolution

Housekeeping/recap

- Lectures generally
- Lecture slides
  - On BBL on Monday morning
- Labs
  - On BBL Monday morning
  - Suggested worked through answers on BBL straight after the lab
- Coursework
  - Pass/fail
- Exam
  - Four questions from five – essay based

Lecture schedule

Week	Lecture Topic	Lecturer	Week Commencing
1	Introducing the module and Software Engineering	Steve Counsell	20th Sept.
2	Software maintenance and Evolution	Steve Counsell	27th Sept.
3	Software metrics	Steve Counsell	4th Oct.
4	Software structure, refactoring and code smells	Steve Counsell	11th Oct.
5	Test-driven development	Giuseppe Destefanis	18th Oct.
6	Software complexity **Coursework released Tues 26th Oct.**	Steve Counsell	25th Oct.
7	**ASK week**	**N/A**	**1st Nov**
8	Software fault-proneness	Steve Counsell	8th Nov.
9	Clean code	Steve Counsell	15th Nov.
10	Human factors in software engineering	Giuseppe Destefanis	22th Nov.
11	SE techniques applied in action	Steve Counsell	29th Dec.
12	Guest Lecture (tba) **Coursework hand-in 6th December**	Guest Lecture	6th Dec.

Lab schedule

Week	Labs	Week Commencing
1	**No labs**	20th Sept.
2	Lab (Introduction)	27th Sept.
3	Lab	4th Oct.
4	Lab	11th Oct.
5	Lab	18th Oct.
6	**No lab**	25th Oct.
7	**ASK week**	**1st Nov.**
8	Lab	8th Nov.
9	Catch-up Lab	15th Nov.
10	Work on coursework (no Lab)	22nd Nov.
11	Work on coursework (no Lab)	29th Nov.
12	**No lab**	6th Dec.

Structure of this lecture

- What is software maintenance and evolution?
- How is it managed?
- Some truisms
- Some relevant and related topics
  - Lehman's Laws of Software Evolution
  - Defensive programming
  - Mob programming

Software Maintenance: definition

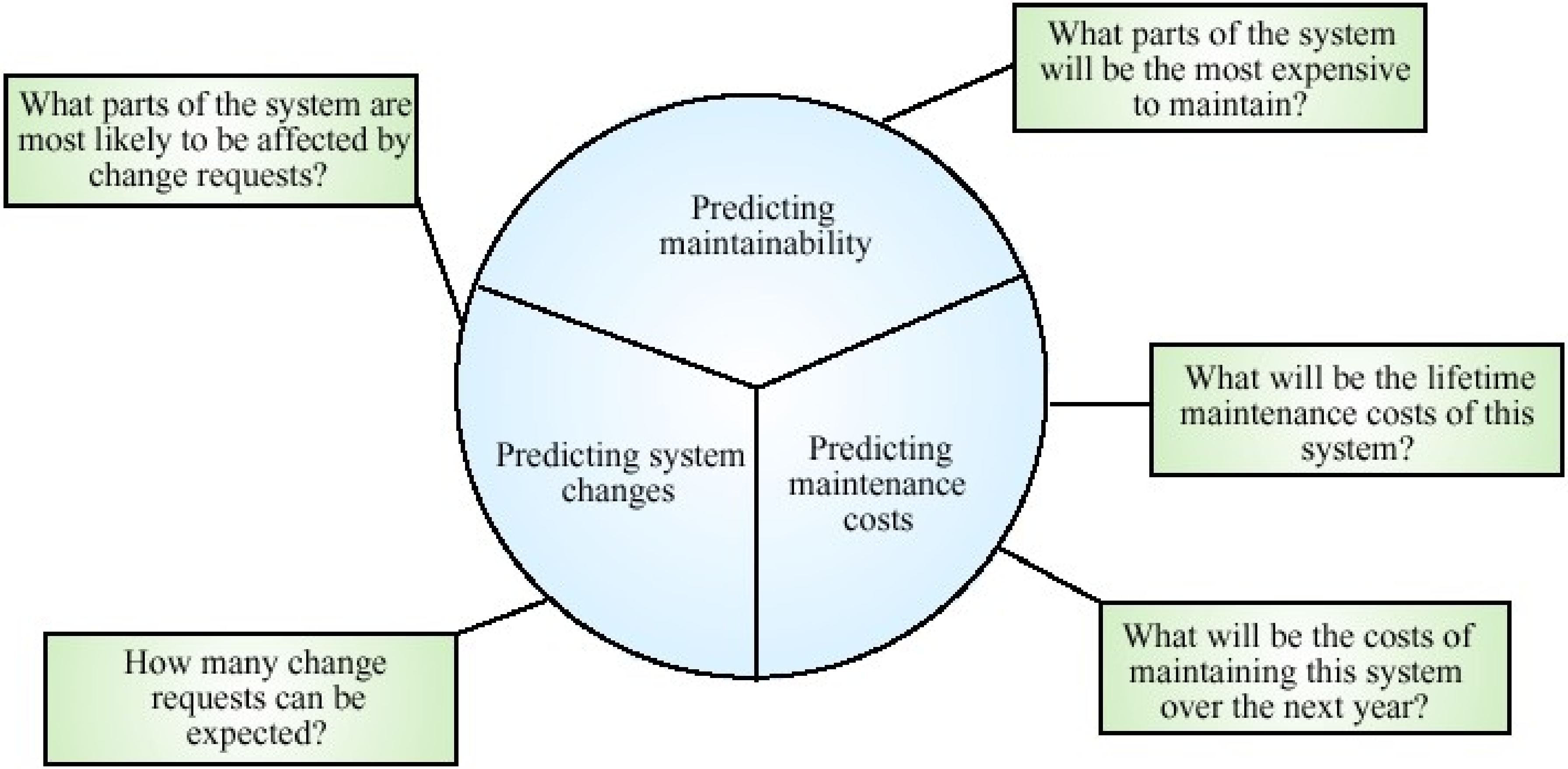
The process of **modifying** a software system or component to correct **bugs**, improve **performance** or other **attributes**, or adapt to a **changed** environment

Definition (cont.)

Process of developing software **initially**, then repeatedly **updating** it for various reasons

- Maintenance of evolving software is unavoidable, continuous and essential
- Important to *design for maintenance*
- You need to think about why software maintenance is so difficult

Maintenance prediction (slide taken from Somerville)


Predicting maintainability
Predicting system changes
Predicting maintenance costs
What parts of the system are most likely to be affected by change requests?
What parts of the system will be the most expensive to maintain?
What will be the lifetime maintenance costs of this system?
What will be the costs of maintaining this system over the next year?
How many change requests can be expected?

Types of Maintenance?

- **Adaptive** maintenance
  - Adapting to changes in the environment
    - Both hardware and software
- **Corrective** maintenance
  - Correcting errors/bugs
  - Error vs Bug vs fault vs failure
- **Perfective** maintenance
  - Making what’s there “better”
- **Preventative** maintenance
  - “Future-proofing” the code for later

Maintenance during development

- Higher quality code means less corrective maintenance
- Anticipating changes means less adaptive maintenance
- Better development practices means less perfective maintenance
- Better tuning to user needs means less maintenance overall
    - Less maintenance overall:
        - Avoid code ‘bloat’
        - Avoid code smells emerging

Some harsh facts

- Maintenance is **expensive**:
  - Sommerville reports 70% software costs is due to maintenance
  - Costs **vary** across application domain (e.g. real time, embedded, games)
  - Classic example: US air force project: £20 per line code to develop and £2,500 per line code to maintain
- Maintenance is **difficult**:
  - 47% effort goes into understanding the system and identifying **dependencies** (Pfleeger)
  - Making changes is also **risky** because of **ripple effects** and dependencies
  - Can be a low morale job

Factors that affect maintenance & evolution

(1) Team stability

(2) Poor development practice (**quality**)

(3) Staff skills

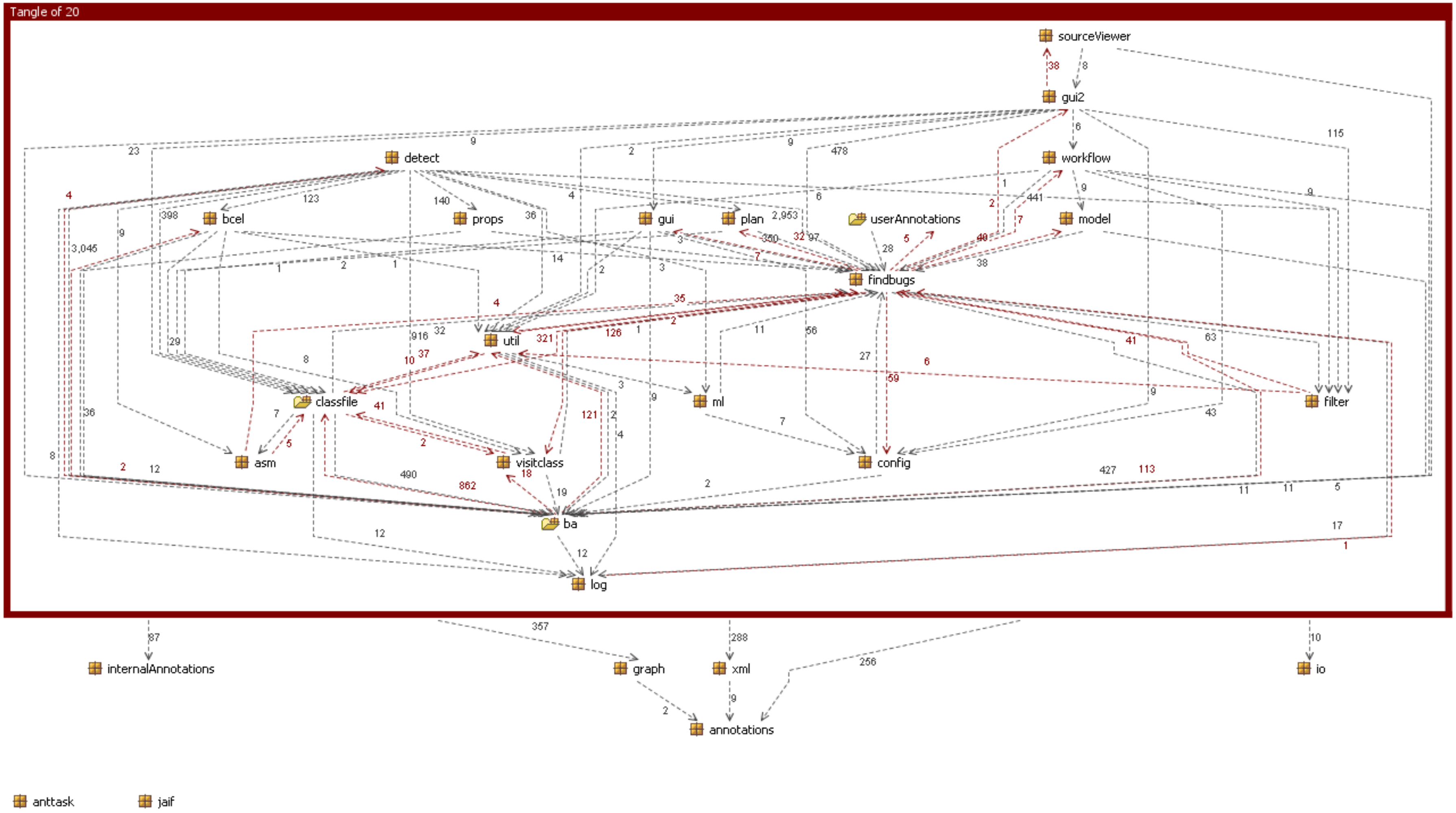
(4) Program age and structure

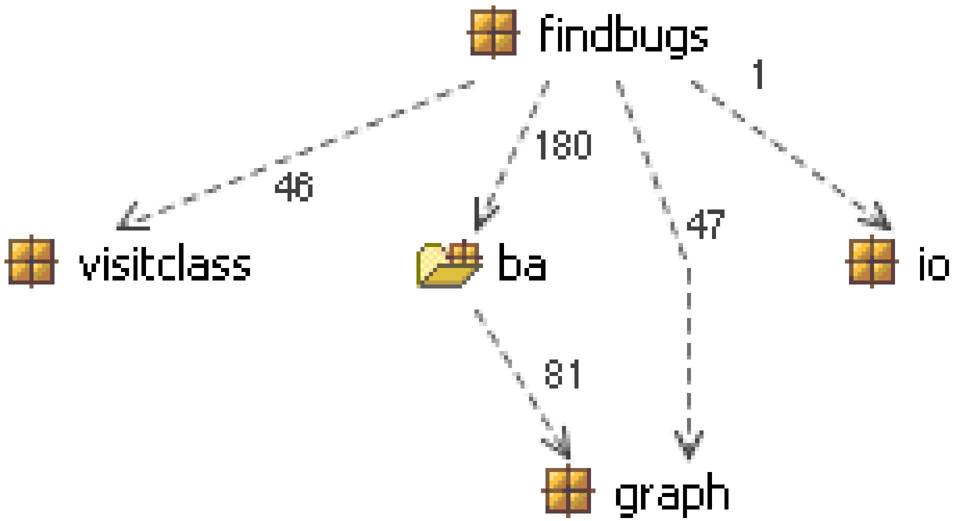
(5) The amount of “technical debt” in a system

    (1) The result of not doing maintenance when you should

    (2) That “ignored” maintenance will come back to haunt you later

https://www.linkedin.com/pulse/technical-debt-infinite-mortgage-your-system-julien-dollon/


Tangle of 20
sourceViewer
38
8
gui2
23
9
2
9
478
6
115
detect
workflow
4
123
140
4
6
1
441
9
9
398
bcel
props
36
gui
plan
2,953
userAnnotations
model
9
2
7
3,045
350
32
97
28
5
40
1
2
1
14
2
3
7
38
findbugs
4
35
2
126
1
11
56
41
63
29
916
32
util
321
10
37
8
27
6
59
classfile
3
9
ml
9
filter
36
7
41
121
2
4
43
5
2
asm
visitclass
config
8
2
12
490
862
18
19
2
427
113
11
11
5
ba
12
17
12
1
log
87
357
288
256
10
internalAnnotations
graph
xml
io
2
9
annotations
anttask
jaif



https://structure101.com/2008/11/27/software-erosion-findbugs

Two factors that help manage software evolution

1. Change management

- Many **change requests** are generated for most systems
  - Fall into the different maintenance categories (see earlier slide on types of maintenance)
- Need to ensure that change is implemented *rationally*
- Factors considered include:
  - ❑ Urgency, Value, Impact, Benefit, Cost
- Usually companies have a team who **analyse** (and **prioritise**) change requests
- In XP **customers** involved in prioritising changes

2. Version control

- A repository of the **first version** of the system and all subsequent **changes** made to it.
- Subsequent versions stored in the form of ‘**diffs**’ (deltas)
- All software has multiple **versions (branches?)**
  - Different platforms
  - Different customer variants
  - Different stages in the lifecycle
- Need to be able to:
  - Recreate old versions
  - Keep track of and control changes
  - Support independent development

Why is version control important?

- Stops changes being made to the wrong version
- Prevents the wrong version being delivered to a user
- Controlling change is a major issue
  - Many change requests will be generated
- Allows the right files to be associated with the right version/release
- A change can be rolled back
- Concurrent changes by developers controlled
- The evolution of the system tracked
- New releases, versions, families of software can be developed in a more organised way
- Can help in fault identification
- Who and when made a particular change is recorded

Eight Relevant Topics to Maintenance (and system evolution)

1. Software Evolution Theory

- Lehman's Laws of Software Evolution
  - 1. Continuing Change: A system must be continually adapted else it becomes progressively less satisfactory in use
  - 2. Increasing Complexity: As a system evolves its complexity increases unless work is done to maintain or reduce it
  - 3. Continuing Growth: The functional capability of systems must be continually increased to maintain user satisfaction over the system lifetime
  - 4. Declining Quality: Unless rigorously adapted to take into account changes in the operational environment, the quality of a system will appear to decline as it is evolved
  - 5. Feedback System: Evolution processes are multi-level, multi-loop, multi-agent feedback systems

2. Defensive programming

- Definition: A technique where you assume the worst for all inputs
- Helps maintenance because you are insuring for the future
- Rules of defensive programming
    - Rule 1: Never assume anything about the input
        - Most problems in code come from unexpected input
            - A negative age or an age >150!
    - Rule 2: Use standards
        - Use a proper coding standard and stick to it
            - Even things like the position of brackets in Java
    - Rule 3: Keep your code as simple as possible
        - Reuse wherever possible because it is usually more trusted and documented

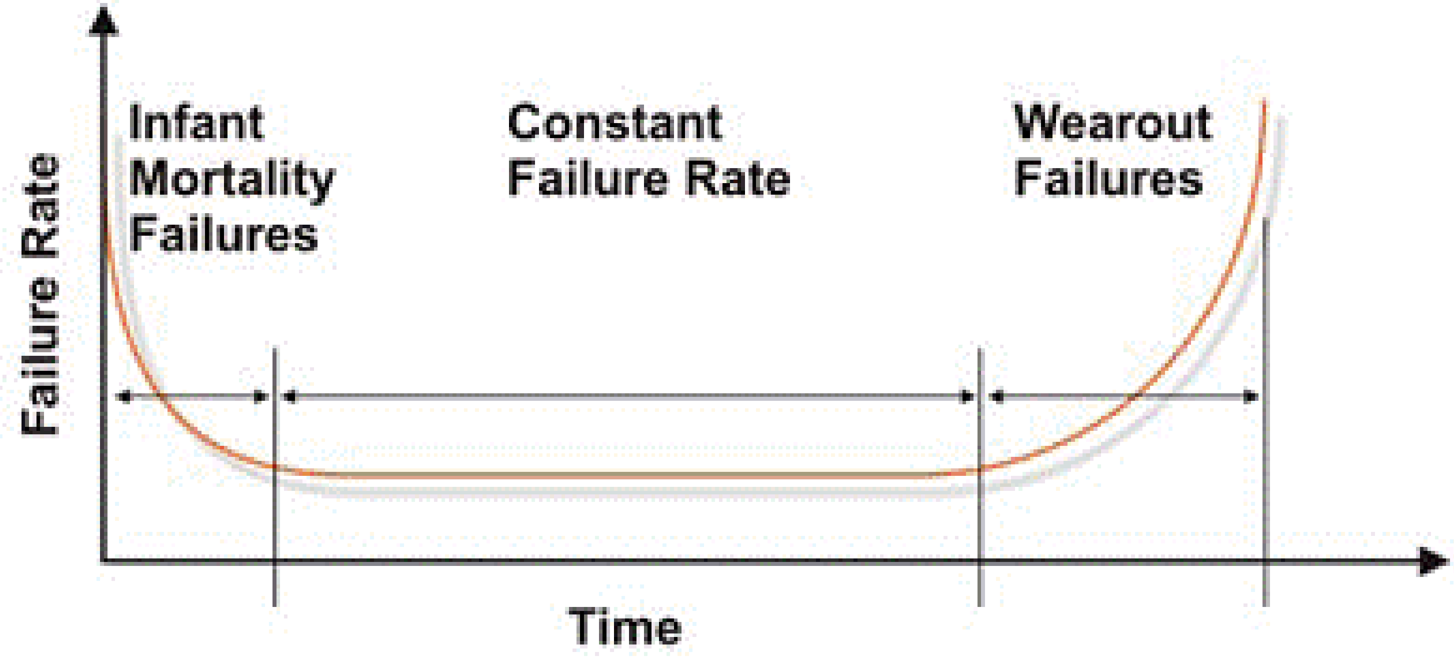
Defensive programming (cont.)

- Defensive programming conforms to the "fail fast" principle
  - Get the bugs out of the systems asap
- Techniques:
  - Use diagnostic code
  - Standardise error handling
  - Use exception handling and assertions
  - Always test external API and library references
  - Assume nothing about the input!

3. Mob programming

- Definition: a software development approach where the whole team works on the same thing, at the same time, in the same space, and at the same computer.
    - Builds on principles of lean manufacturing, extreme programming, and lean software development
- Covers definition of user stories or requirements, designing, coding, testing, deploying software, and working with the customer and business experts.
- Work is handled in working meetings or workshops:
    - all involved in creating the software are considered to be team members, including the customer and business experts.
    - Also works for distributed teams in the same virtual space using screen sharing

4. Bathtub curve



Often criticized (unjustly) the bathtub curve is more of a conceptual tool than a predictive tool.

What affects the shape of the bathtub?

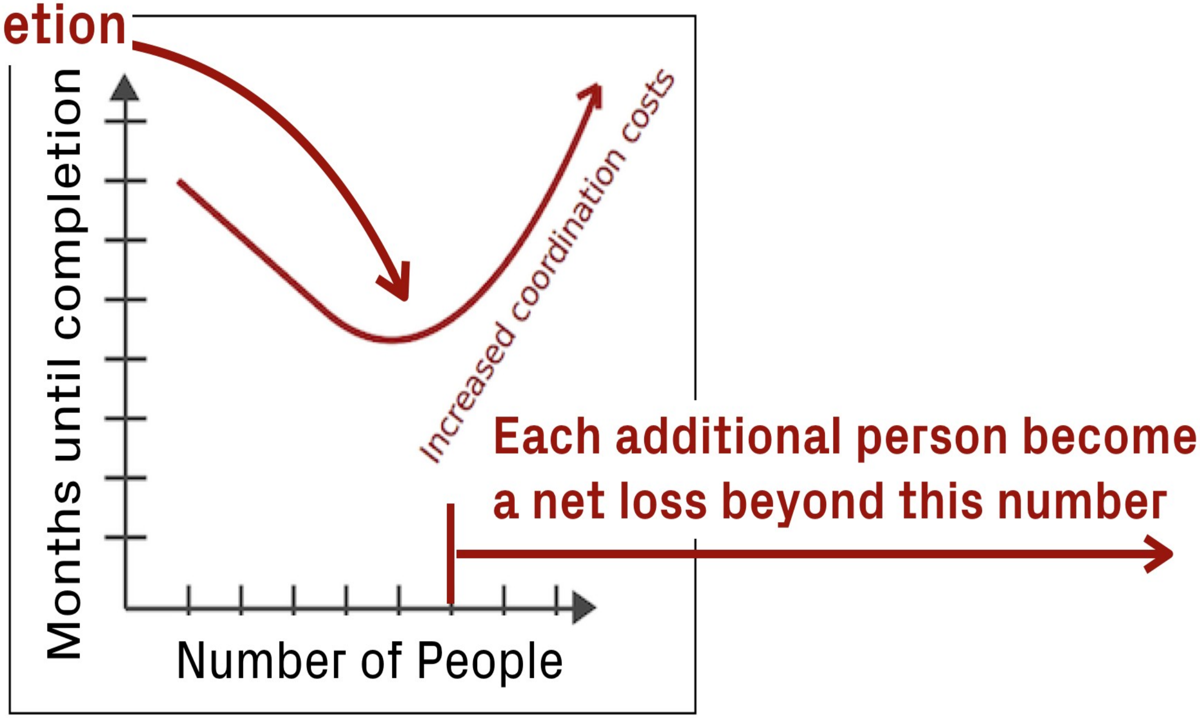
- If the system is poor:
  - There will be high numbers of problems at the start and it will take longer to reach the stability period
  - The period of ‘constant failure rate’ in the middle will be shorter
  - The decline will be much quicker
- So, what would a good system’s bathtub look like?
  - Low numbers of problems at the start
    - Reach the ‘constant failure rate’ stage quickly
  - Long period of ‘constant failure rate’
  - A slow decline

5. Brooks' Law

- "Adding human resources to a late software project makes it later"
- Why?
  - Ramp-up
    - Getting staff trained up on "what's happening" with the project
  - Complexity of communication
    - With two people in a team, there is only one communication channel (between person x and person y)
    - With five people there are ten!

Graphically

Persons vs Time to Completion



Time to completion

Each additional person becomes a net loss beyond this number

https://codescene.com/blog/visualize-brooks-law

6. Death march

- A project which is believed by participants to be destined for failure, or that requires a stretch of unsustainable overwork.
  - The project marches to its death as its members are forced by their superiors to continue the project against their better judgment
- I have some experience of this at the Gas company where I worked
- Yourdon wrote a book on the topic “Death March”

7. The software crisis

- Projects running over-budget
- Projects running over-time
- Software was very inefficient
- Software was of low quality
- Software often did not meet requirements
- Projects were unmanageable and code difficult to maintain
- Software was never delivered

8. Scope and feature creep

- Scope creep of a project
  - Arises when the boundaries of what a system is meant to do are changed after the project has been launched
  - Caused by:
    - Poor design
    - Poor communication during design and development
- Feature creep:
  - Arises when new features are continually being asked for after a project has been launched
    - From any "stakeholder"
  - Some domains are notoriously bad (notably games development)

Questions

- **Question 1:** What is Brooks' Law and why is it important?
- **Question 2:** It is impossible to change the shape of the Bathtub Curve. Discuss.
- **Question 3:** Appraise the use of defensive programming.
- **Question 4:** What are the advantages and disadvantages of Mob Programming?

Reading for the week

- *Sommerville Eds. 9 & 10, Chapter 9*
- *Sommerville Eds. 9 & 10, Chapter 25*
- Most popular OSS version control tools reviewed: http://www.smashingmagazine.com/2008/09/18/the-top-7-open-source-version-control-systems/
- Software Systems as Cities: A Controlled Experiment, Richard Wettel, Michele Lanza, and Romain Robbes, In Proceedings of ICSE 2011 (33rd International Conference on Software Engineering), pp. 551 - 560, ACM Press, 2011
- Death march: https://www.informit.com/articles/article.aspx?p=169512

Reading for the week (cont.)

Brook’s Law:

https://codescene.com/blog/visualize-brooks-law/

https://stevemcconnell.com/articles/brooks-law-repealed/

Bathtub Curve for systems:

https://www.linkedin.com/pulse/20140723115956-15133887-the-software-bathtub-curve-understanding-the-software-systems-lifecycle

Lehman’s Laws

https://researcher.watson.ibm.com/researcher/view_group.php?id=7296

Reading for the week (cont.)

For a much longer and in-depth read:

https://www.researchgate.net/publication/259979752_An_Empirical_Study_of_Lehman's_Law_on_Software_Quality_Evolution/link/02e7e52ee52794d397000000/download

Technical Debt:

https://martinfowler.com/bliki/TechnicalDebt.html

Software Crisis

https://en.wikipedia.org/wiki/Software_crisis