

Software Engineering

CST 1b

Ross Anderson

Aims

- Introduce students to software engineering, and in particular to the problems of
 - building large systems
 - building safety-critical systems
 - building real-time systems
- Illustrate what goes wrong with case histories
- Study software engineering practices as a guide to how mistakes can be avoided

Objectives

- At the end of the course you should know how writing programs with tough assurance targets, or in large teams, or both, differs from the programming exercises done so far.
- You should appreciate the waterfall, spiral and evolutionary models of development and be able to explain which kinds of software development might profitably use them

Objectives (2)

- You should appreciate the value of other tools and the difference between incidental and intrinsic complexity
- You should understand the basic economics of the software development lifecycle
- You should also be prepared for the organizational aspects of your part 1b group project, for your part 2 project, and for courses in systems, security etc

Resources

- Recommended reading:
 - S Maguire, ‘Debugging the Development Process’
 - N Leveson, ‘Safeware’ (see also her ‘System Safety Engineering’ online)
 - SW Thames RHA, ‘Report of the Inquiry into the London Ambulance Service’
 - RS Pressman, ‘Software Engineering’

Resources (2)

- Additional reading:
 - FP Brooks, ‘The Mythical Man Month’
 - J Reason, ‘The Human Contribution’
 - MPP cases: O Campion-Awwad et al, ‘The National Programme for IT in the NHS – A Case History’
 - H Thimbleby, ‘Improving safety in medical devices and systems’
 - R Anderson, ‘Security Engineering’ 2e, ch 25–6
- And read widely in whichever application areas interest you!

Outline of Course

- The 'Software Crisis'
- How to organise software development
- Critical software
- Tools
- Large systems
- Guest lecture on current industrial practice (25th October)

The 'Software Crisis'

- Software lags far behind the hardware's potential!
- Many large projects are late, over budget, dysfunctional, or abandoned (LAS, CAPSA, NPfIT, DWP, Addenbrookes ...)
- Some failures cost lives (Therac 25) or billions (Ariane 5, NPfIT)
- Some expensive scares (Y2K, Pentium)
- Some combine the above (LAS)

The London Ambulance Service System

- Widely cited example of project failure because it was thoroughly documented (and its pattern has been frequently repeated)
- Attempt to automate ambulance dispatch in 1992 failed conspicuously with London being left without service for a day
- Hard to say how many deaths could have been avoided; estimates ran as high as 20
- Led to CEO being sacked, public outrage

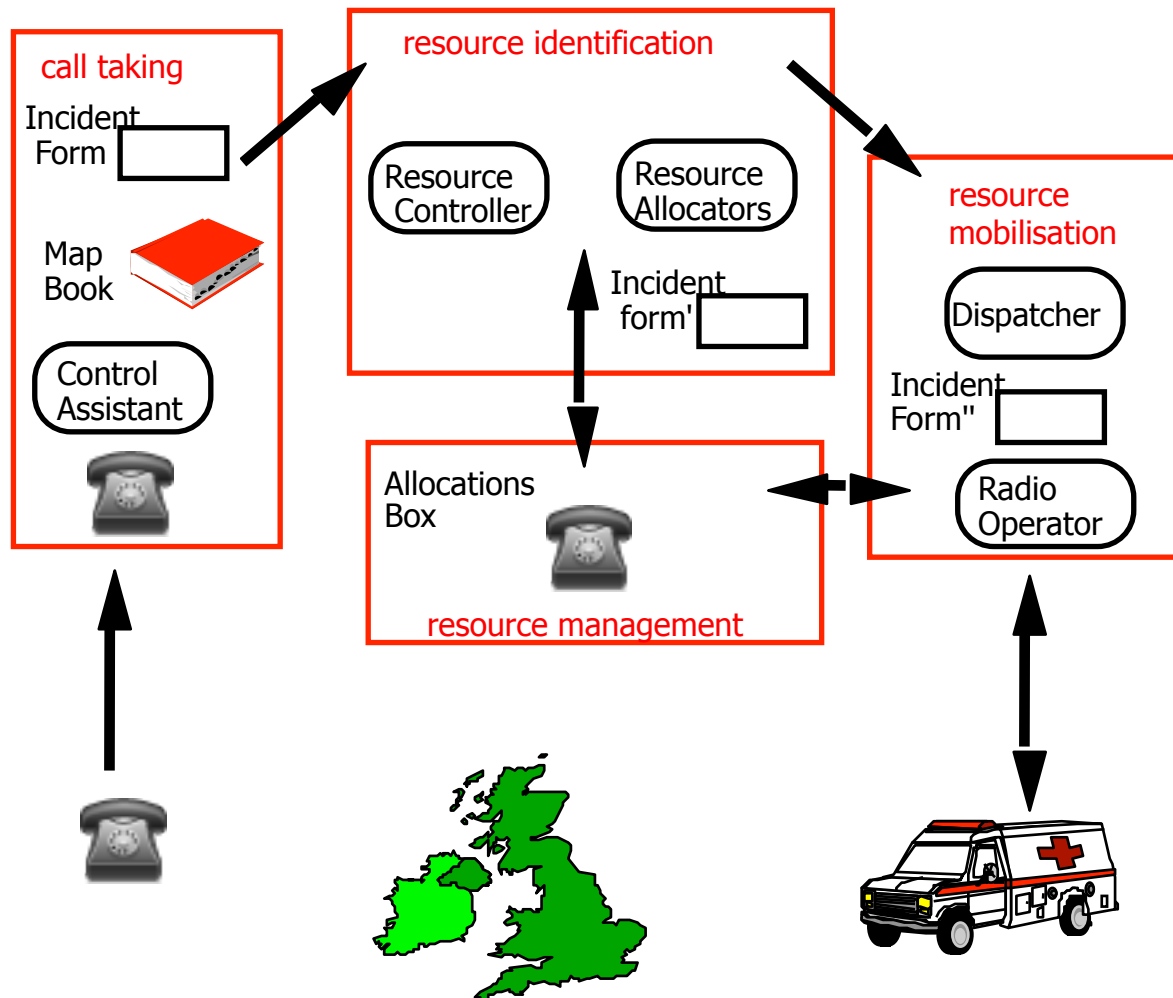
Original System

- 999 calls written on paper tickets; map reference looked up; conveyor to central point
- Controller deduplicates tickets and passes to three divisions – NW / NE / S
- Division controller identifies vehicle and puts note in its activation box
- Ticket passed to radio controller
- This all takes about 3 minutes and 200 staff of 2700 total. Some errors (esp. deduplication), some queues (esp. radio), call-backs tiresome

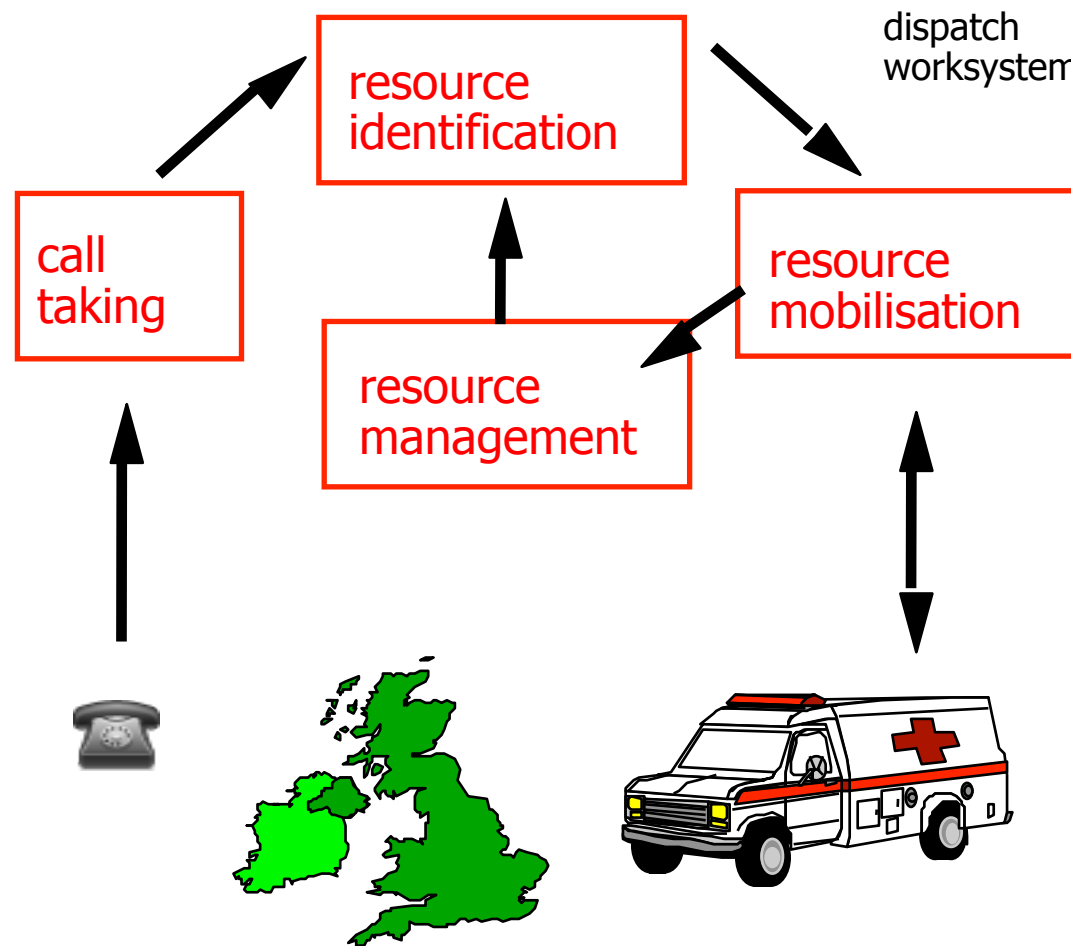
Project Context

- Attempt to automate in 1980s failed – system failed load test
- Industrial relations poor – pressure to cut costs
- Public concern over service quality
- SW Thames RHA decided on fully automated system: responder would email ambulance
- Consultancy study said this might cost £1.9m and take 19 months – provided a packaged solution could be found. AVLS would be extra

The Manual Implementation



Dispatch System



- Large
- Real-time
- Critical
- Data rich
- Embedded
- Distributed
- Mobile components

despatch domain

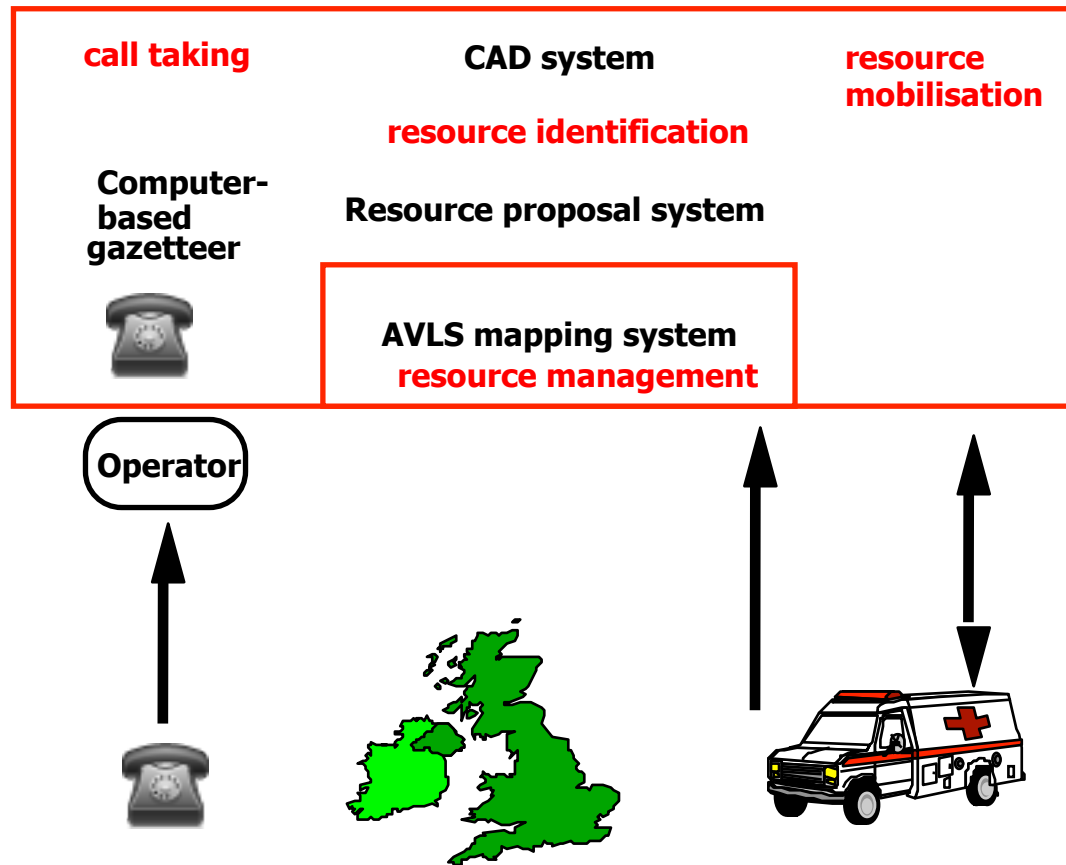
Bid process

- Idea of a £1.5m system stuck; idea of AVLS added; proviso of a packaged solution forgotten; new IS director hired
- Tender 7/2/1991 with completion deadline 1/92
- 35 firms looked at tender; 19 proposed; most said timescale unrealistic, only partial automation possible by 2/92
- Tender awarded to consortium of Systems Options Ltd, Apricot and Datatrak for £937,463 – £700K cheaper than next lowest bidder!

First Phase

- Design work 'done' July
- Main contract signed in August
- LAS told in December that only partial automation by January deadline – front end for call taking, gazetteer, docket printing
- Progress meeting in June had already minuted a 6 month timescale for an 18 month project, a lack of methodology, no full-time LAS user, and SO's reliance on 'cozy assurances' from subcontractors

The Goal



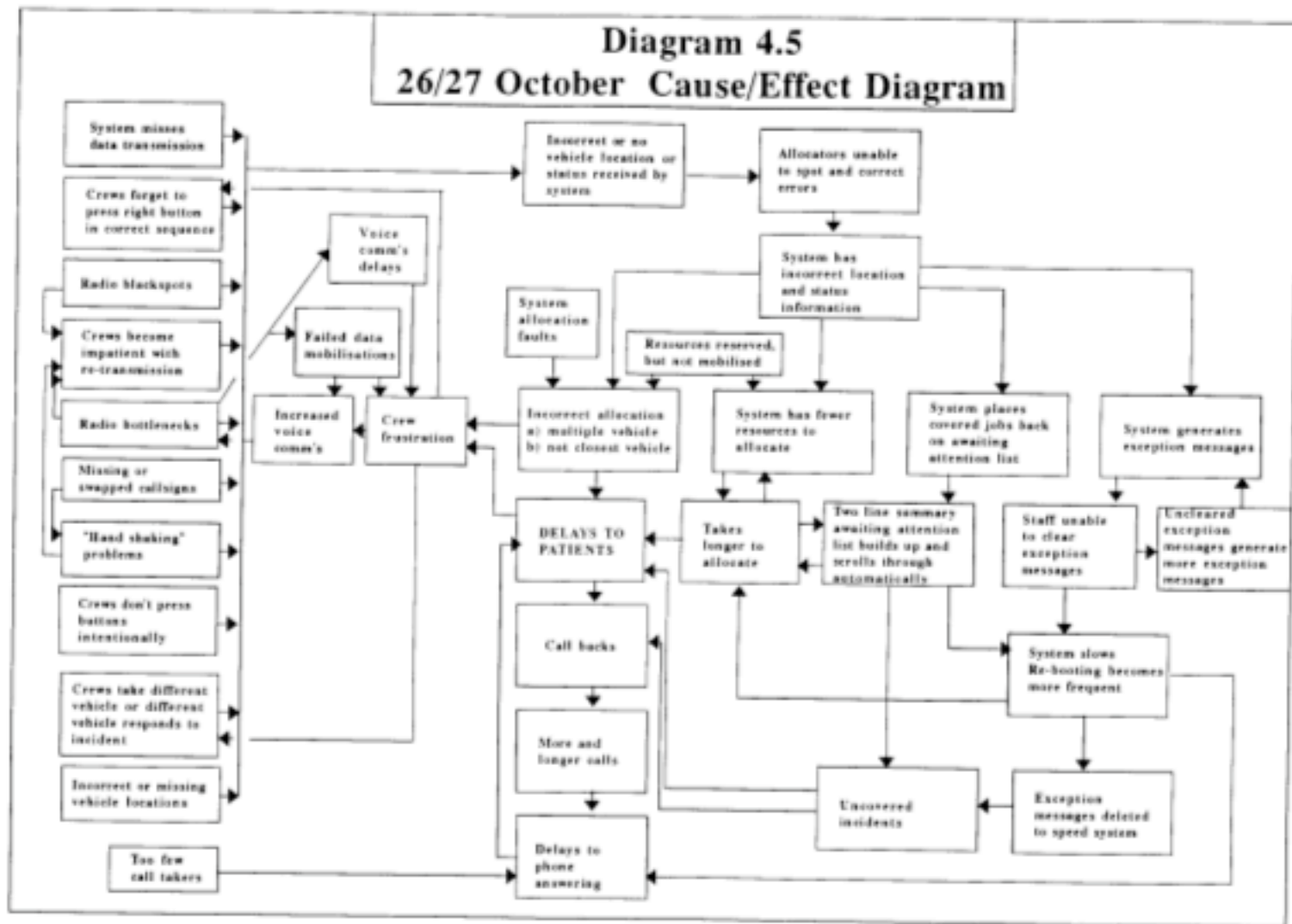
From Phase 1 to Phase 2

- Server never stable in 1992; client and server lockup
- Phase 2: radio messaging with blackspots and congestion. Couldn't cope with 'established working practices'
- Yet management decided to go live 26/10/92
- CEO: "No evidence to suggest that the full system software, when commissioned, will not prove reliable"
- Independent review had called for volume testing, implementation strategy, change control ... It was ignored!
- On 26 Oct, the room was reconfigured to use terminals, not paper. There was no backup...

LAS Disaster

- 26/7 October vicious circle:
 - system progressively lost track of vehicles
 - exception messages scrolled up off screen and were lost
 - incidents held as allocators searched for vehicles
 - callbacks from patients increased causing congestion
 - data delays → voice congestion → crew frustration → pressing wrong buttons and taking wrong vehicles → many vehicles sent to an incident, or none
 - slowdown and congestion leading to collapse
- Switch back to semi-manual operation on 26th and to full manual on Nov 2 after crash

Diagram 4.5
26/27 October Cause/Effect Diagram



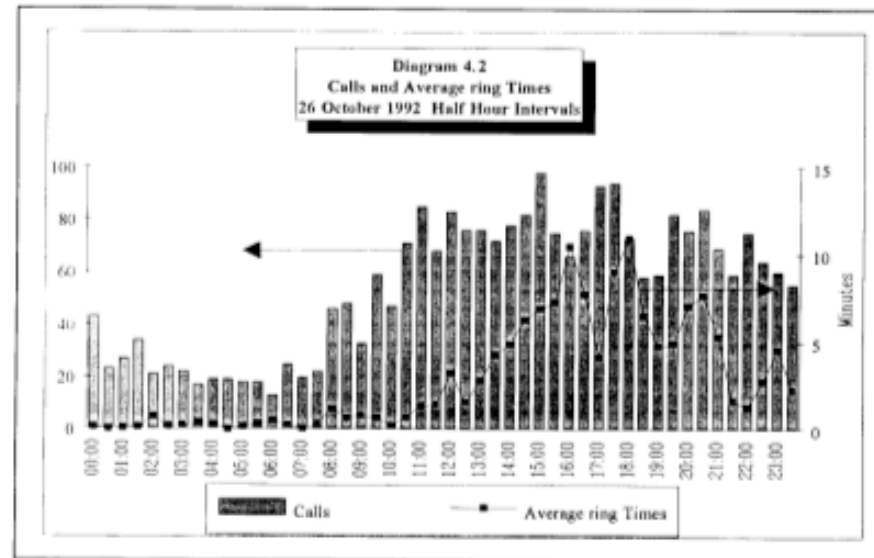
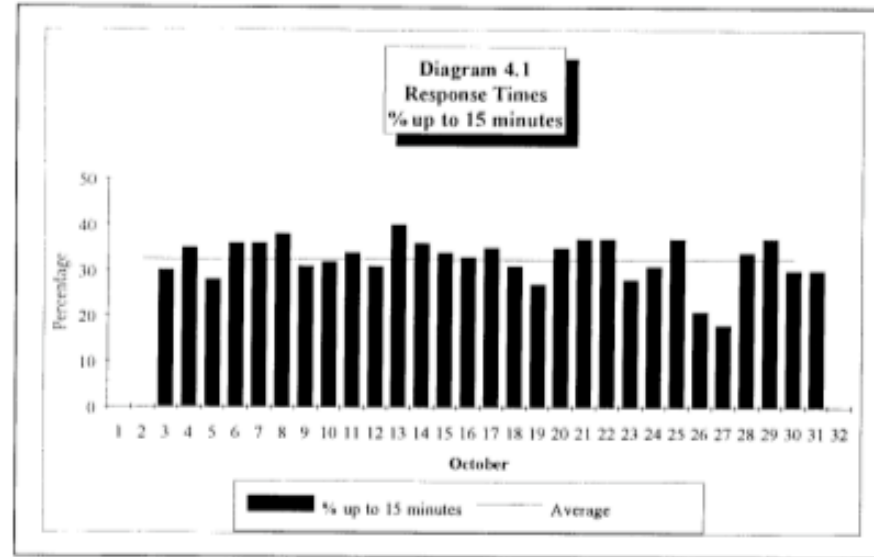


Diagram 4.3
Total A&E Patients Carried
October

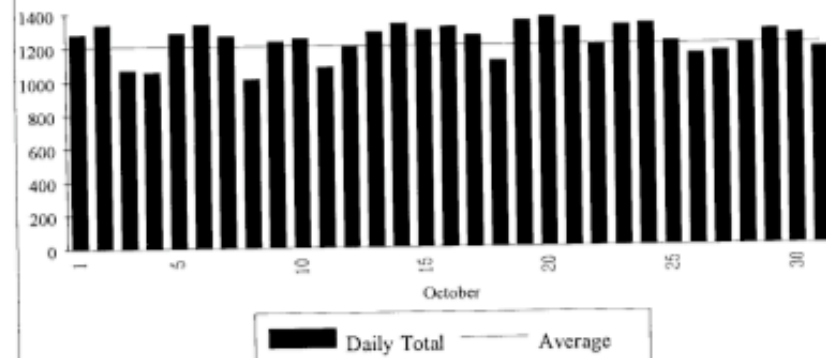
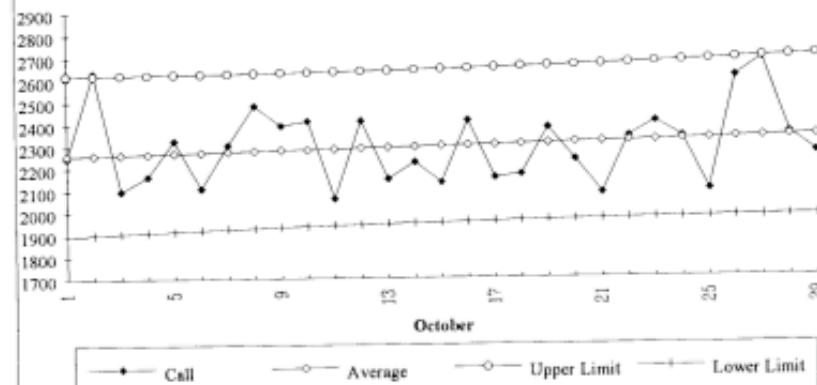


Diagram 4.4
Calls recorded by call logger



Collapse

- Entire system descended into chaos:
 - e.g., one ambulance arrived to find the patient dead and taken away by undertakers
 - e.g., another answered a ‘stroke’ call after 11 hours, 5 hours after the patient had made their own way to hospital
- People probably died as a result
- Chief executive resigns

What Went Wrong – Spec

- LAS ignored advice on cost and timescale
- Procurers insufficiently qualified and experienced
- No systems view
- Specification was inflexible but incomplete: it was drawn up without adequate consultation with staff
- Attempt to change organisation through technical system (3116)
- Ignored established work practices and staff skills

What Went Wrong – Project

- Confusion over who was managing it all
- Poor change control, no independent QA, suppliers misled on progress
- Inadequate software development tools
- Ditto technical comms, and effects not foreseen
- Poor interface for ambulance crews
- Poor control room interface

What Went Wrong – Go-live

- System went live with known serious faults
 - slow response times
 - workstation lockup
 - loss of voice comms
- Software not tested under realistic loads or as an integrated system
- Inadequate staff training
- No back up

NHS National Programme for IT

- Like LAS, an attempt to centralise power and change working practices
- Earlier failed attempt in the 1990s
- The February 2002 Blair meeting
- Five LSPs plus national contracts: £12bn
- Most systems years late and/or don't work
- Coalition government: NPfIT 'abolished'
- See case history written by MPP students in 2014 (linked from course materials page)

Next – Universal Credit

- Idea: unify hundreds of welfare benefits and mitigate poverty trap by tapered withdrawal as claimants start to earn
- Supposed to go live Oct 2013! Problems ...
- General: big systems take 7 years not 3
- They hoped ‘agile’ development would fix it ...
- Depended on real-time feed of tax data from HMRC, which in turn depended on firms
- Descended into chaos; NAO report

Next – Smart Meters?

- Idea: expose consumers to market prices, get peak demand shaving, make use salient
- EU Electricity Directive 2009: 80% by 2020
- Labour 2009: £10bn centralised project to save the planet and help fix supply crunch in 2017
- March 2010: experts said we just can't change 47m meters in 6 years. So excluded from spec
- Coalition government: wanted deployment by 2015 election! Planned to build central system Mar–Sep 2013 (then: Sep 2014 ...)
- Contracts tendered while spec still fluid...
- Spec still fluid, tech getting obsolete, despair ...

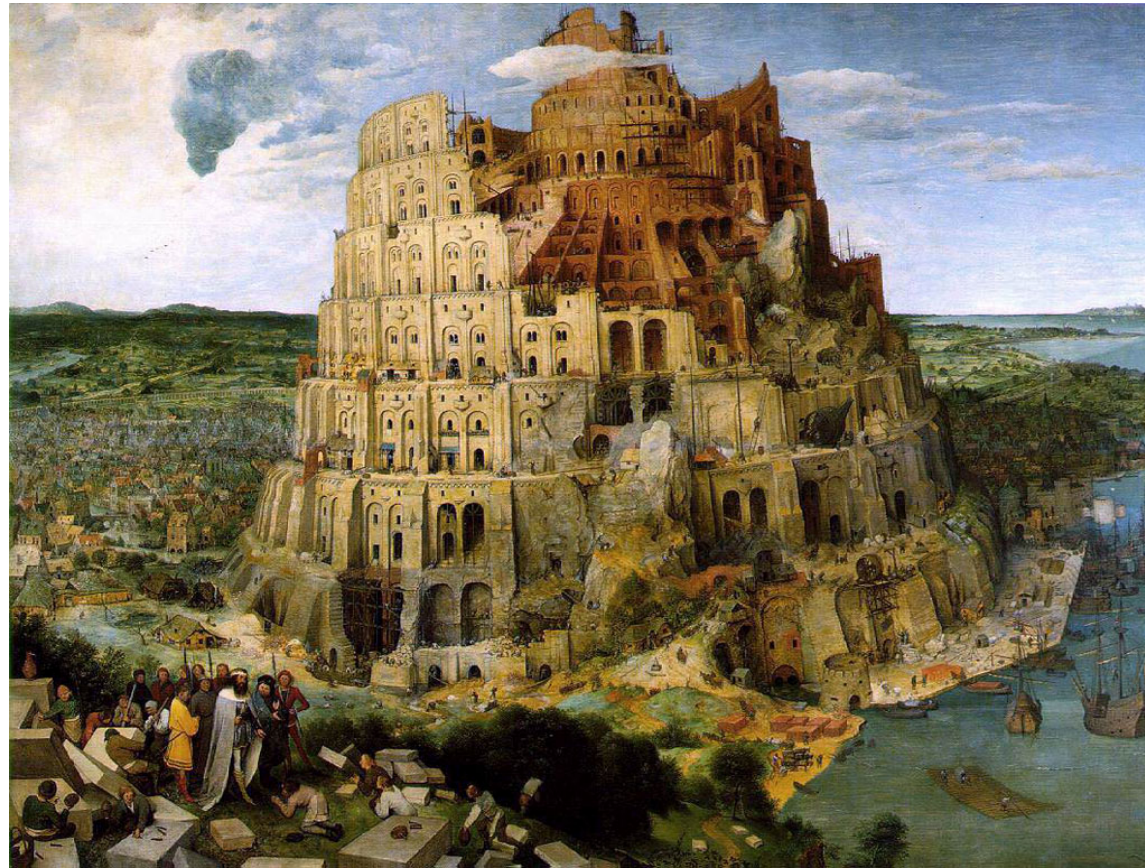
CAPSA

- Cambridge University wanted 'commitment accounting' scheme for research grants etc
- Oracle Financials bid £9m vs next bid £18m; VC unaware of Oracle disasters at Bristol, Imperial,...
- Target was Sep 1999 (Y2K fix); a year late
- Old system staff sacked Sep 2000 to save money
- Couldn't cope with volume
- 15 years later, still can't supply the data that grant holders or departmental administrators want
- We had to write our own scripts to scrape and mash it to make it even halfways workable

Managing Complexity

- Software engineering is about managing complexity at a number of levels
 - At the micro level, bugs arise in protocols etc because they're hard to understand
 - As programs get bigger, interactions between components grow at $O(n^2)$ or even $O(2^n)$
 - ...
 - With complex socio-technical systems, we can't predict reactions to new functionality
- Most failures of really large systems are due to wrong, changing, or contested requirements (see paper by Curtis, Krasner and Iscoe on web page)

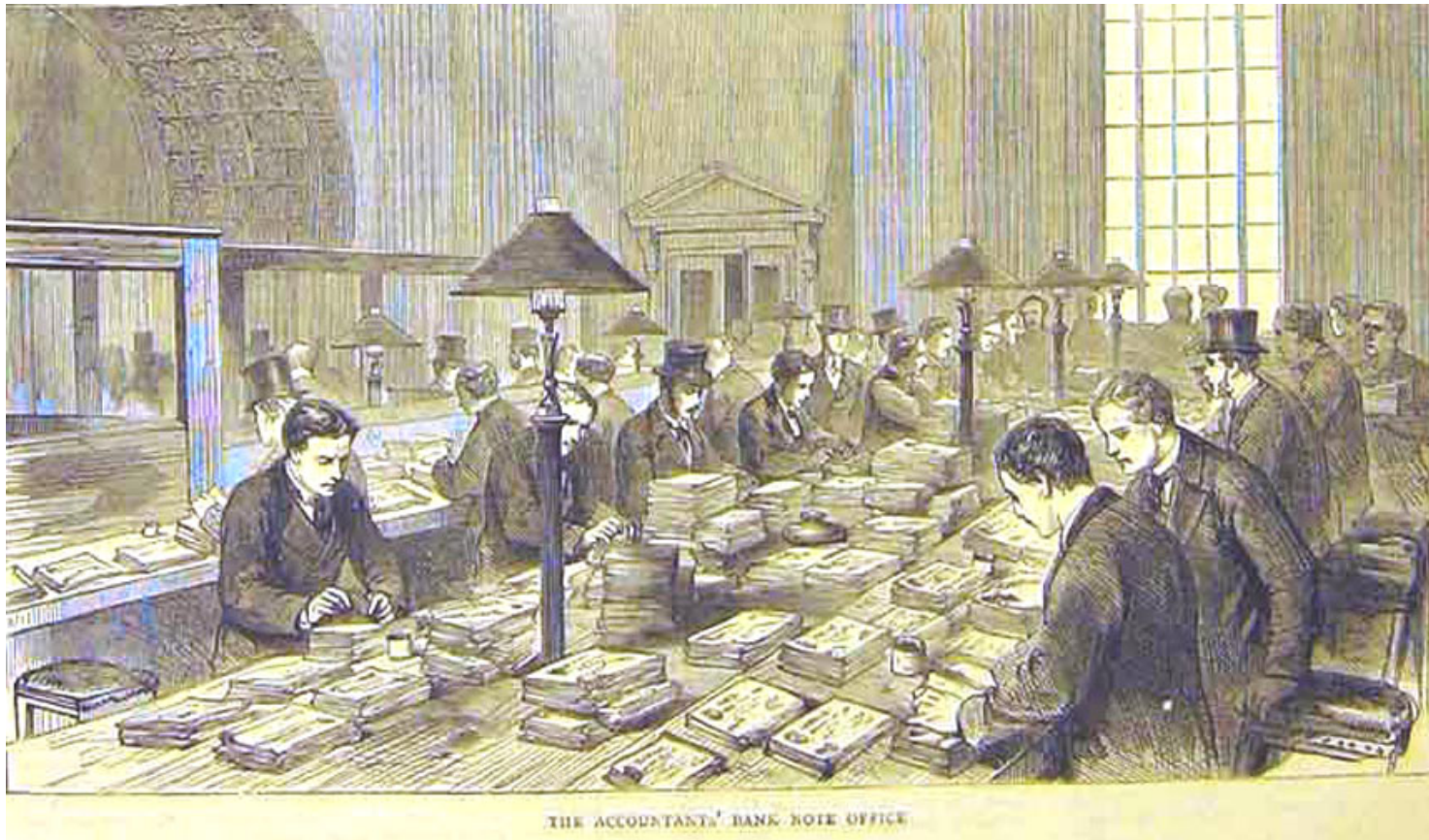
Project Failure, c. 1500 BC



Nineteenth Century

- Charles Babbage, 'On Contriving Machinery'
 - “It can never be too strongly impressed upon the minds of those who are devising new machines, that to make the most perfect drawings of every part tends essentially both to the success of the trial, and to economy in arriving at the result”

Complexity, 1870 – Bank of England



Complexity 1876 – Dun, Barlow & Co



Complexity 1906 – Sears, Roebuck



- Continental-scale mail order meant specialization
- Big departments for single bookkeeping functions
- Beginnings of automation

Complexity 1940 – First National Bank of Chicago



1960s – The Software Crisis

- In the 1960s, large powerful mainframes made even more complex systems possible
- People started asking why project overruns and failures were so much more common than in mechanical engineering, shipbuilding...
- ‘Software engineering’ was coined in 1968
- The hope was that we could things under control by using disciplines such as project planning, documentation and testing

How is Software Different?

- Many things that make writing software fun also make it complex and error-prone:
 - joy of solving puzzles and building things from interlocking moving parts
 - stimulation of a non-repeating task with continuous learning
 - pleasure of working with a tractable medium, ‘pure thought stuff’
 - complete flexibility – you can base the output on the inputs in any way you can imagine
 - satisfaction of making stuff that’s useful to others

How is Software Different? (2)

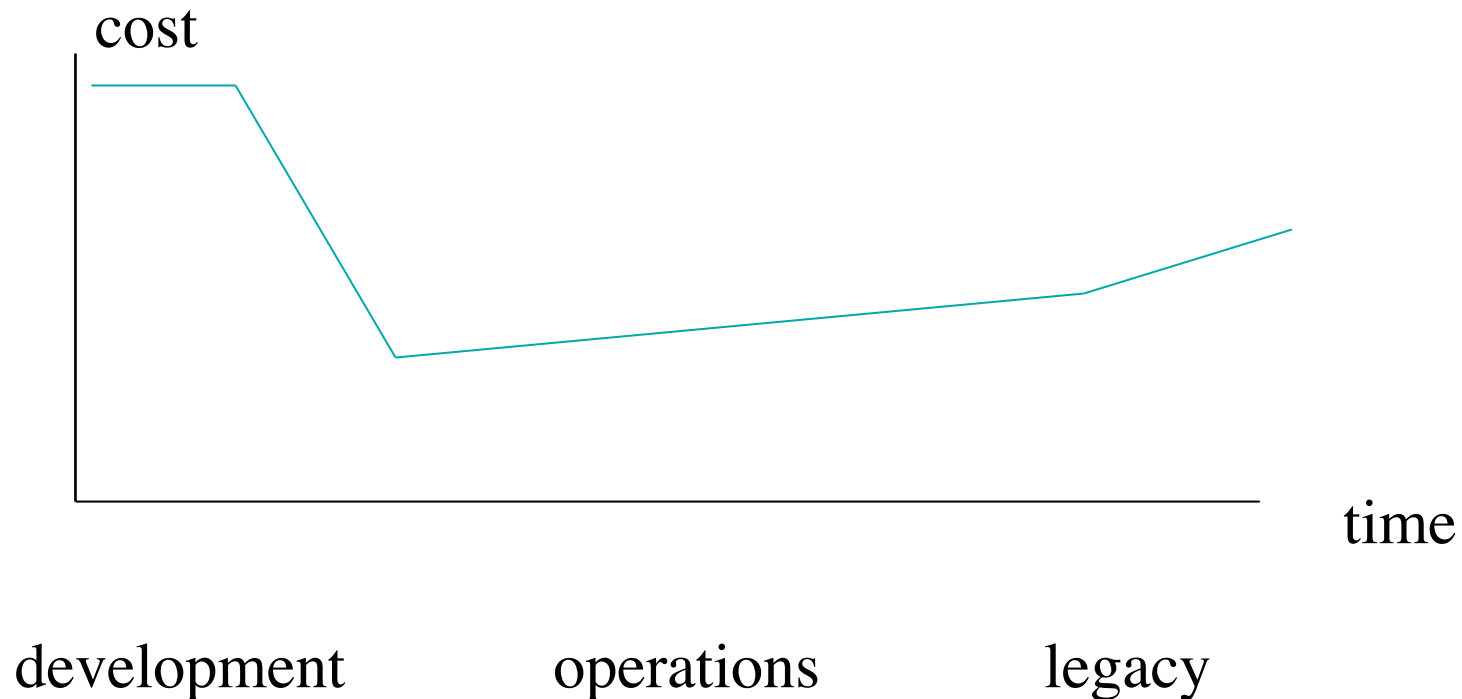
- Large systems become qualitatively more complex, unlike big ships or long bridges
- The tractability of software leads customers to demand 'flexibility' and frequent changes
- This makes systems more complex to use over time as 'features' accumulate, and interactions have odd effects
- The structure can be hard to visualise or model
- The hard slog of debugging and testing piles up at the end, when the excitement's past, the budget's spent and the deadline's looming

The Software Life Cycle

- Software economics can get complex
 - Consumers buy on sticker price, businesses on total cost of ownership
 - vendors use lock-in tactics
 - complex outsourcing
- First let's consider the simple (1950s) case of a company that develops and maintains software entirely for its own use

Cost of Software

- Initial development cost (10%)
- Continuing maintenance cost (90%)



What Does Code Cost?

- First IBM measures (60s)
 - 1.5 KLOC/developer year (operating system)
 - 5 KLOC/dev yr (compiler)
 - 10 KLOC/dev yr (app)
- AT&T measures
 - 0.6 KLOC/dev yr (compiler)
 - 2.2 KLOC/dev yr (switch)
- Alternatives
 - Halstead (entropy of operators/operands)
 - McCabe (graph entropy of control structures)
 - Function point analysis

First-generation Lessons Learned

- There are huge variations in productivity between individuals
- The main systematic gains come from using an appropriate high-level language
- High level languages take away much of the accidental complexity, so the programmer can focus on the intrinsic complexity
- It's also worth putting extra effort into getting the specification right, as it more than pays for itself by reducing the time spent on coding and testing

Development Costs

- Barry Boehm, 1975

	Spec	Code	Test
C3I	46%	20%	34%
Space	34%	20%	46%
Scientific	44%	26%	30%
Business	44%	28%	28%

- So – the toolsmith should not focus just on code!

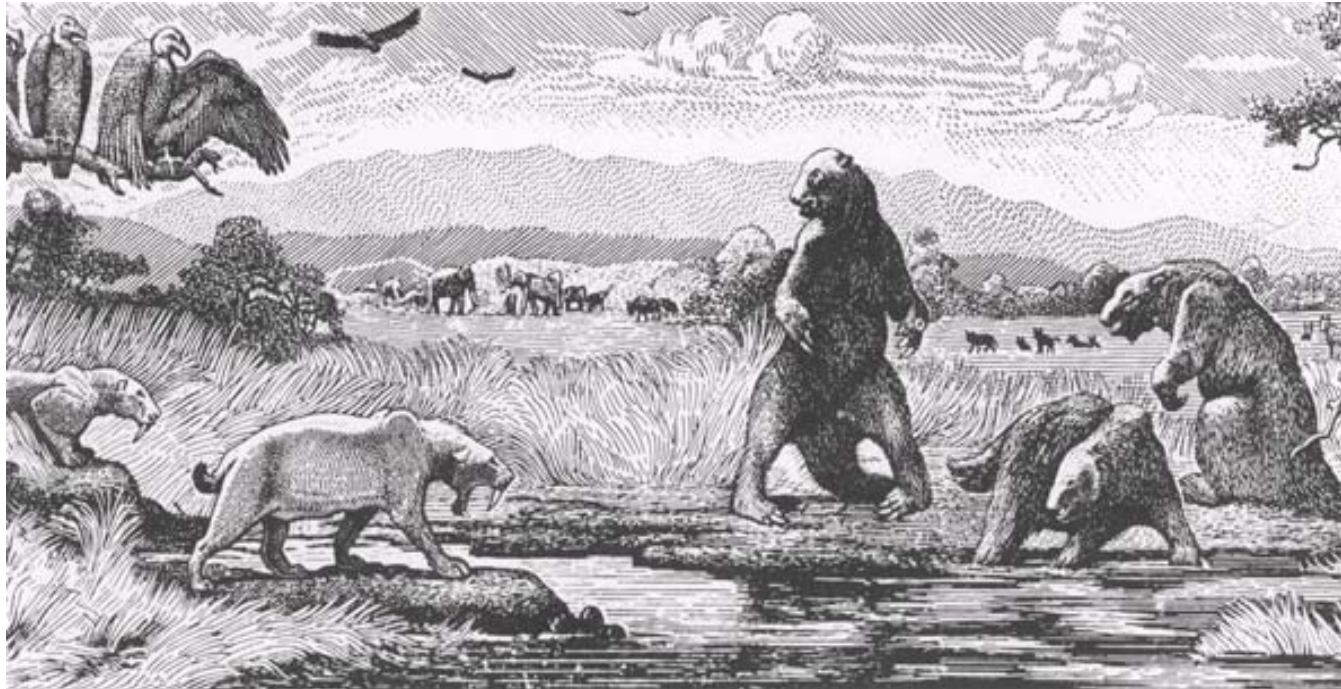
'The Mythical Man-Month'

- Fred Brooks debunked interchangeability
- Imagine a project at 3 developers x 4 months
 - Suppose the design work takes an extra month. So we have 2 months to do 9 dev mth work
 - If training someone takes a month, we must add 6 devs
 - But the work 3 devs did in 3 months can't be done by 9 devs in one! Interaction costs maybe $O(n^2)$
- Hence Brooks' law: adding manpower to a late project makes it later!

Software Engineering Economics

- Boehm, 1981 (empirical studies after Brooks)
 - Cost-optimum schedule time to first shipment
 $T = 2.5(\text{dev-months})^{1/3}$
 - With more time, cost rises slowly
 - With less time, it rises sharply
 - Hardly any projects succeed in less than $3/4 T$
- Other studies show that if people are to be added, you should do it early rather than late
- Some projects fail despite huge resources!

The Software Project ‘Tar Pit’

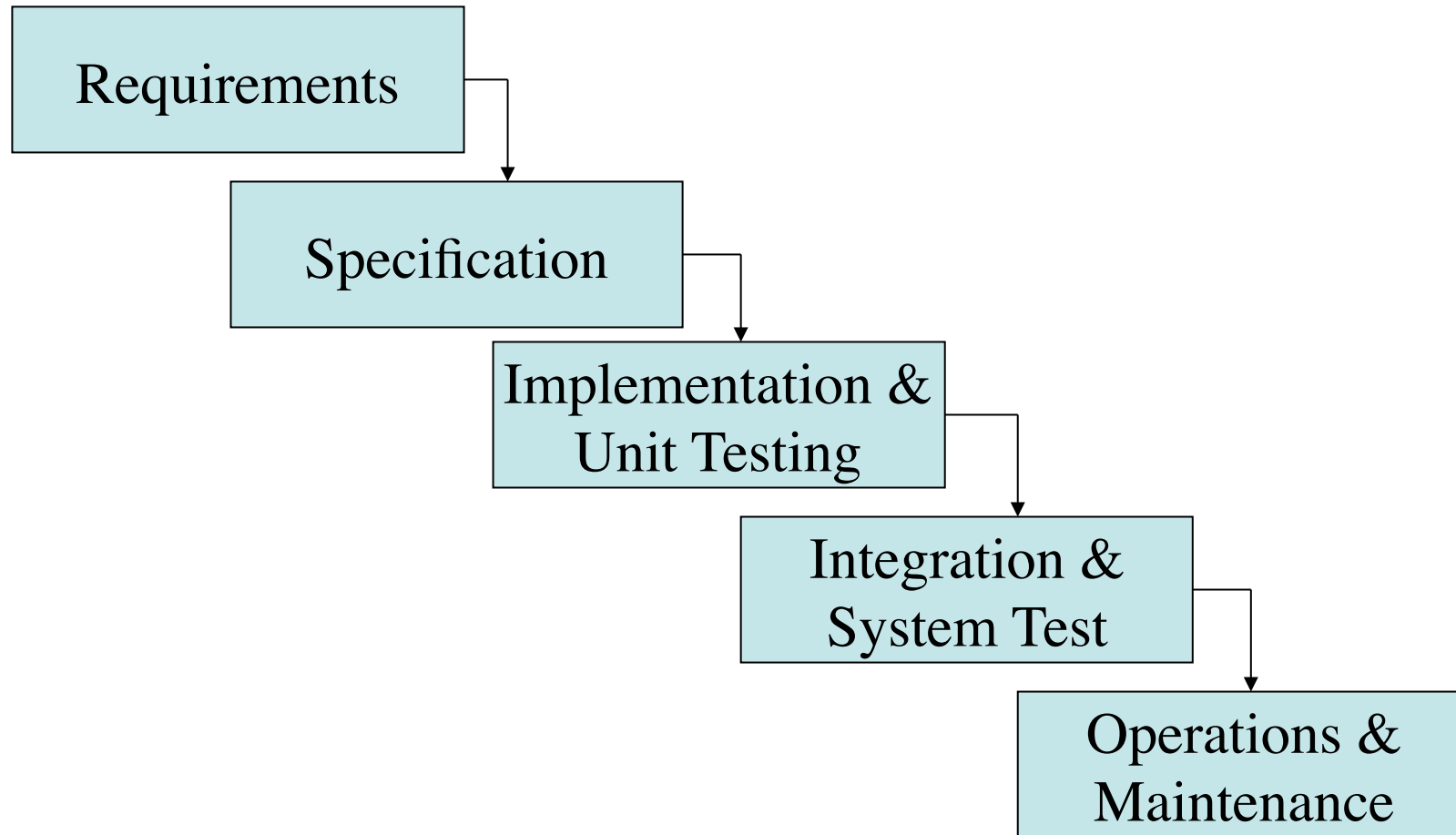


- You can pull any one of your legs out of the tar ...
- Individual software problems all soluble but ...

Structured Design

- The only practical way to build large complex programs is to chop them up into modules
- Sometimes task division seems straightforward (bank = tellers, ATMs, dealers, ...)
- Sometimes it isn't
- Sometimes it just seems to be straightforward
- Quite a number of methodologies have been developed (SSDM, Jackson, Yourdon, ...)

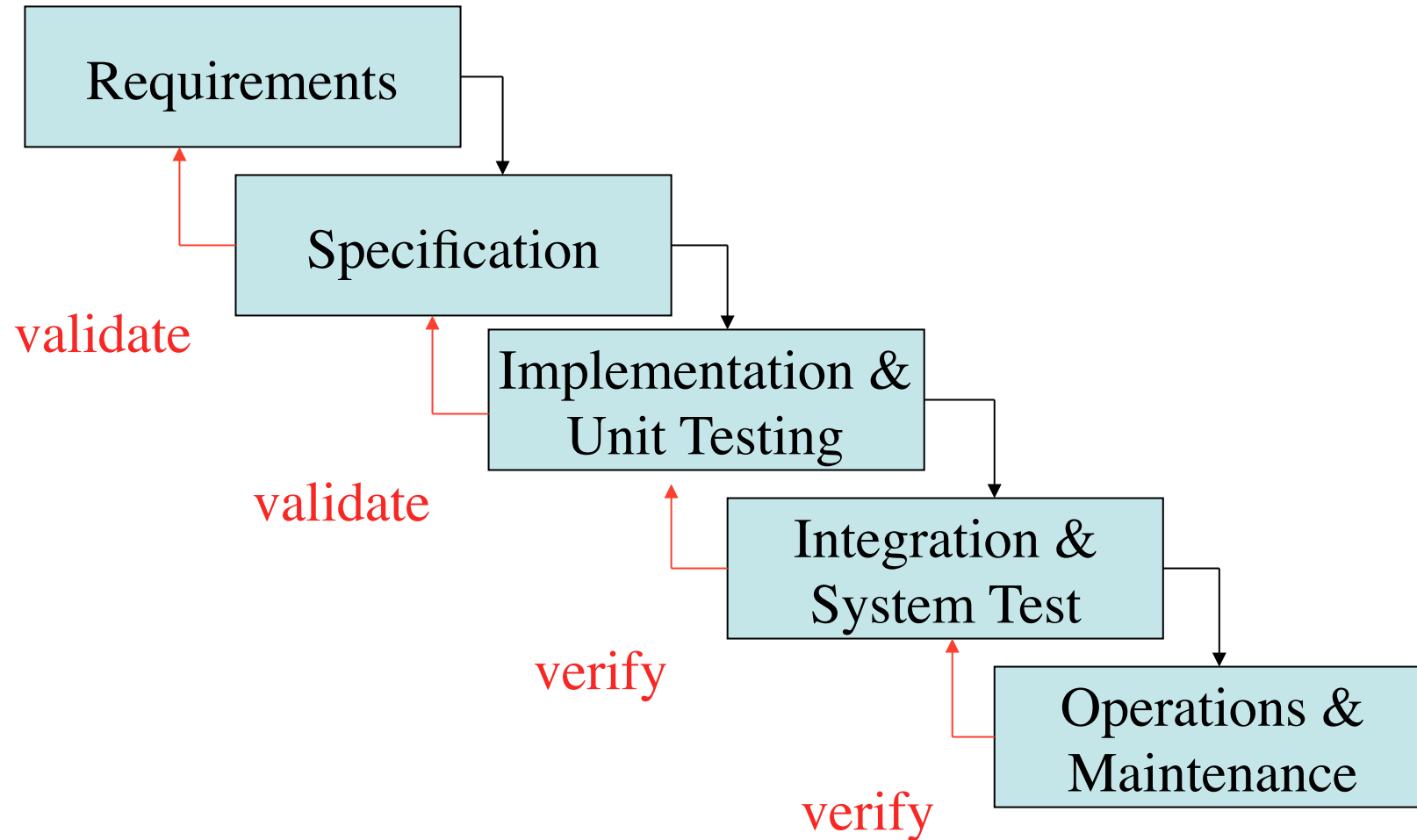
The Waterfall Model



The Waterfall Model (2)

- Requirements are written in the user's language
- The specification is written in system language
- There can be many more steps than this – system spec, functional spec, programming spec ...
- The philosophy is progressive refinement of what the user wants
- Warning – when Winton Royce published this in 1970 he cautioned against naïve use
- But it become a US DoD standard ...

The Waterfall Model (3)



The Waterfall Model (4)

- People often suggest adding an overall feedback loop from ops back to requirements
- However the essence of the waterfall model is that this isn't done
- It would erode much of the value that organisations get from top-down development
- Very often the waterfall model is used only for specific development phases, e.g. adding a feature
- But sometimes people use it for whole systems

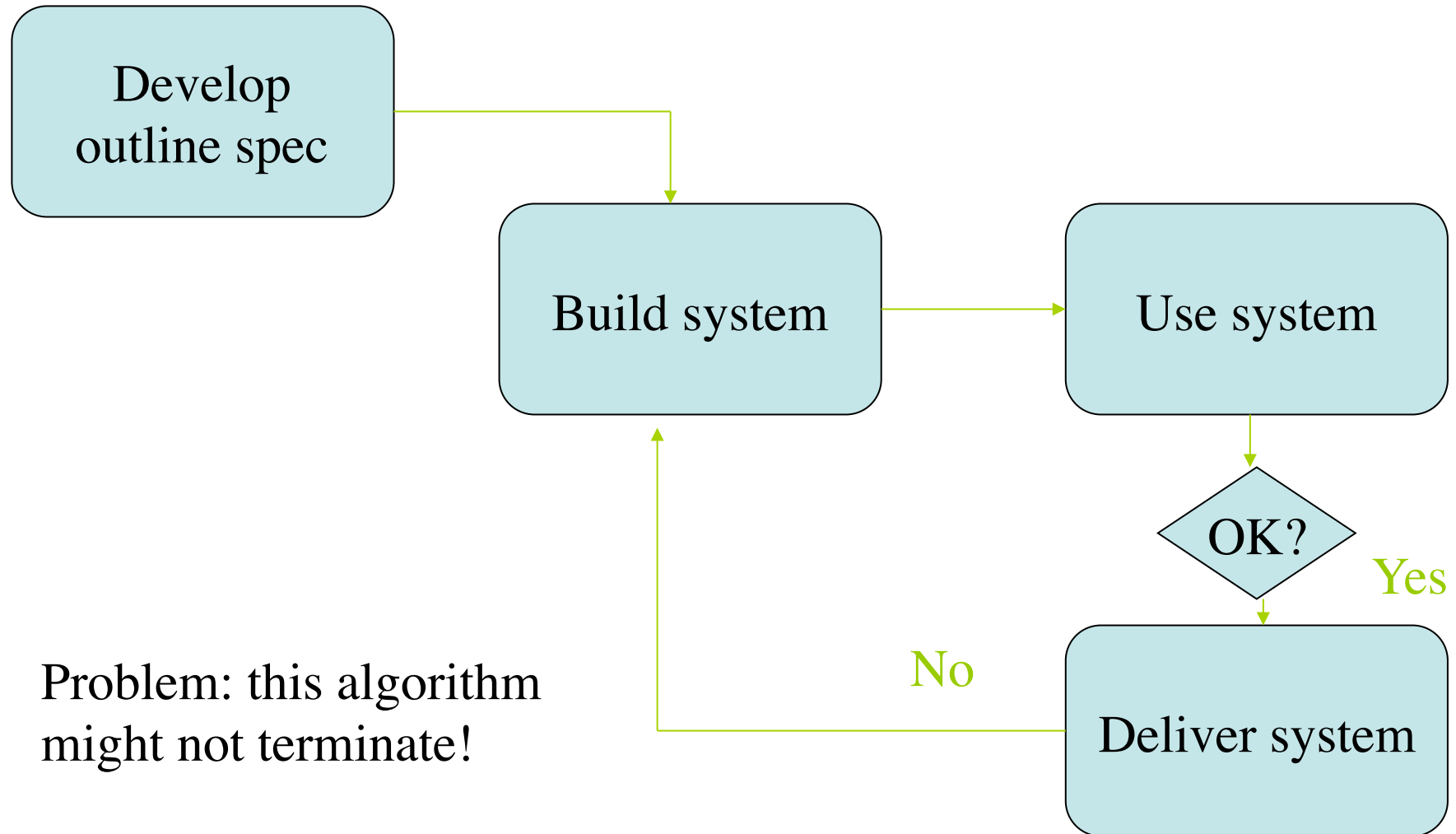
Waterfall – Advantages

- Compels early clarification of system goals and is conducive to good design practice
- Enables the developer to charge for changes to the requirements
- It works well with many management tools, and technical tools
- Where it's viable it's usually the best approach
- The really critical factor is whether you can define the requirements in detail in advance. Sometimes you can (Y2K bugfix); sometimes you can't (HCI)

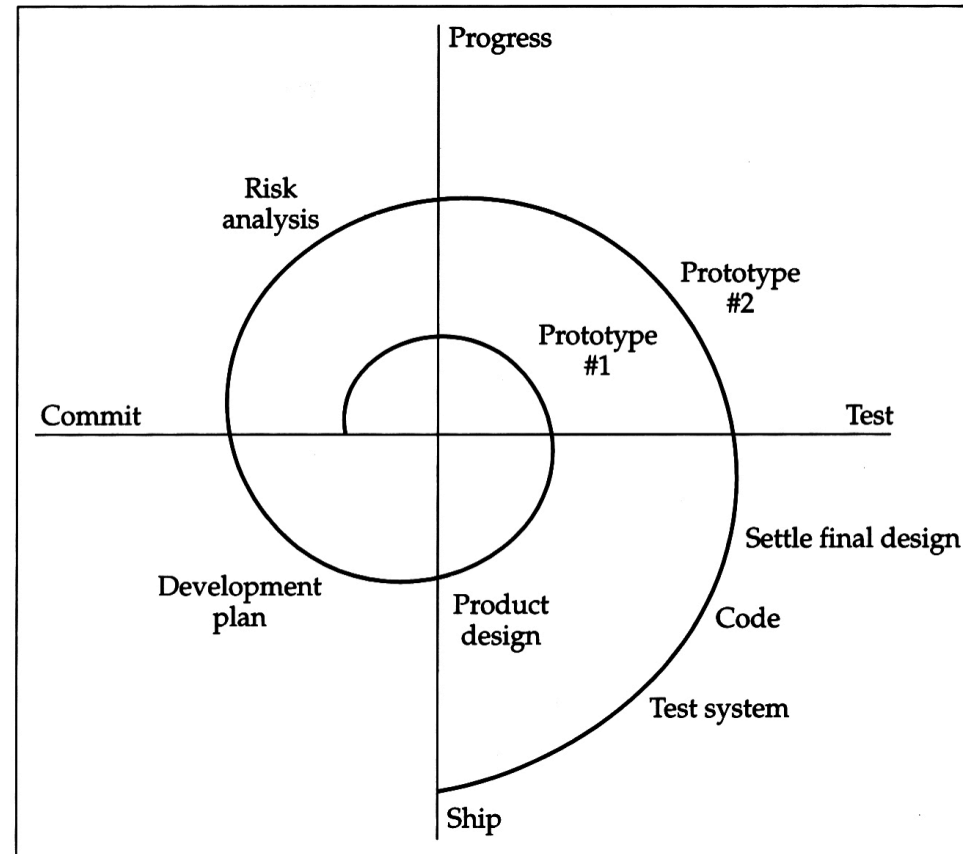
Waterfall – Objections

- Iteration can be critical in the development process:
 - requirements not yet understood by developers
 - or not yet understood by the customer
 - the technology is changing
 - the environment (legal, competitive) is changing
- The attainable quality improvement may be unimportant over the system lifecycle
- Specific objections from safety-critical, package software developers

Iterative Development



Spiral Model



Spiral Model (2)

- The essence is that you decide in advance on a fixed number of iterations
- E.g. engineering prototype, pre-production prototype, then product
- Each of these iterations is done top-down
- “Driven by risk management”, i.e. you concentrate on prototyping the bits you don't understand yet

Evolutionary Model

- Products like Windows and Office are now so complex that they evolve (MS tried to rewrite Word from scratch and failed)
- The big change that's made this possible has been the arrival of automatic regression testing
- Firms now have huge suites of test cases against which daily builds of the software are tested
- The development cycle is to add changes, check them in, and test them

Evolutionary Model (2)

- A modern integrated development environment has several components
 - Code and documentation version control (git)
 - Code review (gerrit)
 - Automated build (make)
 - Continuous integration (Jenkins)
- The guest lecture will discuss the effect this tech has had on the industry
- Think how you set up your group project!

Critical Software

- Many systems must avoid a certain class of failures with high assurance
 - safety critical systems – failure could cause, death, injury or property damage
 - security critical systems – failure could allow leakage of confidential data, fraud, ...
 - real time systems – software must accomplish certain tasks on time
- Critical computer systems have much in common with critical mechanical systems (bridges, brakes, locks,...)
- Key: engineers study how things fail

Tacoma Narrows, Nov 7 1940



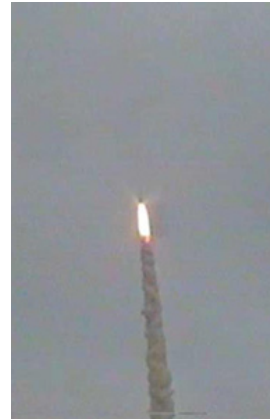
Definitions

- Error: design flaw or deviation from intended state
- Failure: nonperformance of system, (classically) within some subset of specified environmental conditions
- Reliability: probability of failure within a set period of time (typically mtbf, mttf)
- Accident: undesired, unplanned event resulting in specified kind/level of loss

Definitions (2)

- Hazard: set of conditions on system, plus conditions on environment, which can lead to an accident in the event of failure
- Thus: failure + hazard = accident
- Risk: probability of bad outcome
- Thus: risk is hazard level combined with danger (probability hazard \rightarrow accident) and latency (hazard exposure + duration)
- Uncertainty: risk not quantifiable
- Safety: freedom from accidents

Ariane 5, June 4 1996



- Ariane 5 accelerated faster than Ariane 4
- This caused an operand error in float-to-integer conversion
- The backup inertial navigation set dumped core
- The core was interpreted by the live set as flight data
- Full nozzle deflection → 20° angle of attack → booster separation

Real-time Systems

- Many safety-critical systems are also real-time systems used in monitoring or control
- Criticality of timing makes many simple verification techniques inadequate
- Often, good design requires very extensive application domain expertise
- Exception handling tricky, as with Ariane
- Testing can also be really hard

Example – Patriot Missile



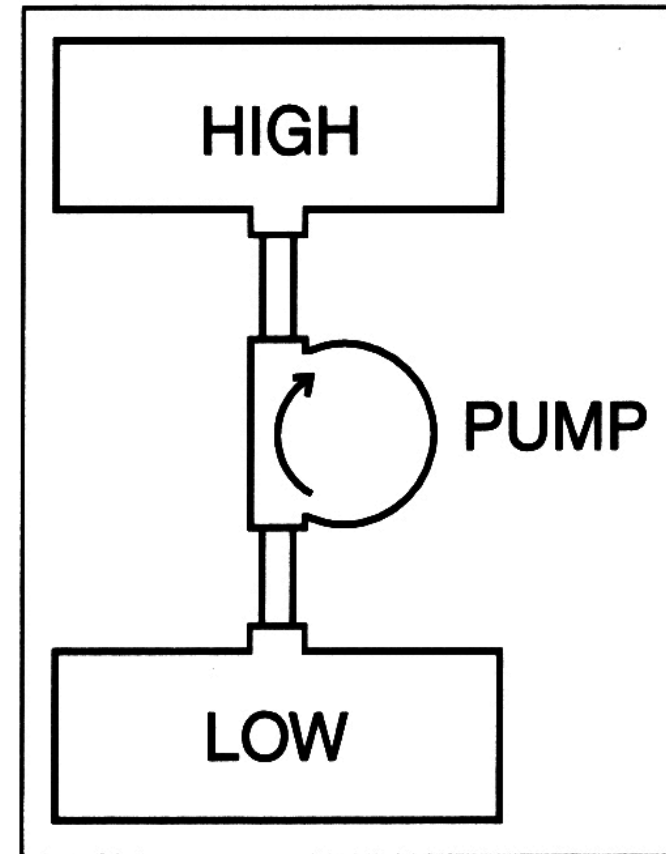
- Failed to intercept an Iraqi scud missile in Gulf War 1 on Feb 25 1991
- SCUD struck US barracks in Dhahran; 28 dead
- Other SCUDs hit Saudi Arabia, Israel

Patriot Missile (2)

- Reason for failure
 - measured time in 1/10 sec, truncated from .0001100110011...
 - when system upgraded from air-defence to anti-ballistic-missile, accuracy increased
 - but not everywhere in the (assembly language) code!
 - modules got out of step by 1/3 sec after 100h operation
 - not found in testing as spec only called for 4h tests
- Critical system failures are typically multifactorial: “a reliable system can’t fail in a simple way”

Security Critical Systems

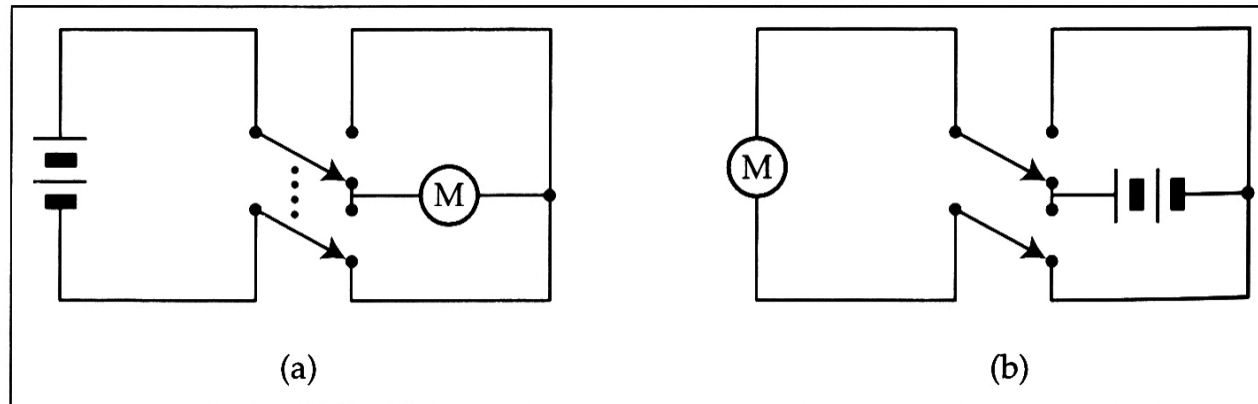
- Usual approach – try to get high assurance of one aspect of protection
- Example: stop classified data flowing from high to low using one-way flow
- Assurance via simple mechanism
- Keeping this small and verifiable is often harder than it looks at first!



Building Critical Systems

- Some things go wrong at the detail level and can only be dealt with there (e.g. integer scaling)
- However in general safety (or security, or real-time performance) is a system property and has to be dealt with there
- A very common error is not getting the scope right
- For example, designers don't consider human factors such as usability and training
- We will move from the technical to the holistic ...

Hazard Elimination



- E.g., motor reversing circuit above
- Some tools can eliminate whole classes of software hazards, e.g. using strongly-typed language such as Ada
- But usually hazards involve more than just software

The Therac Accidents

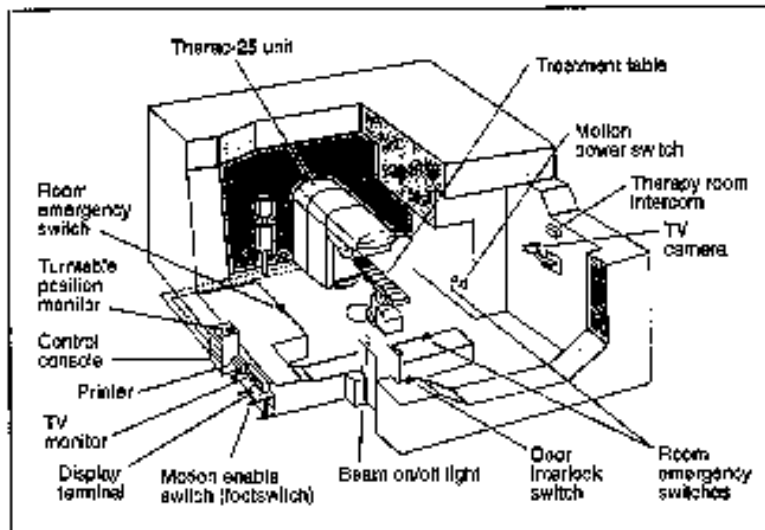
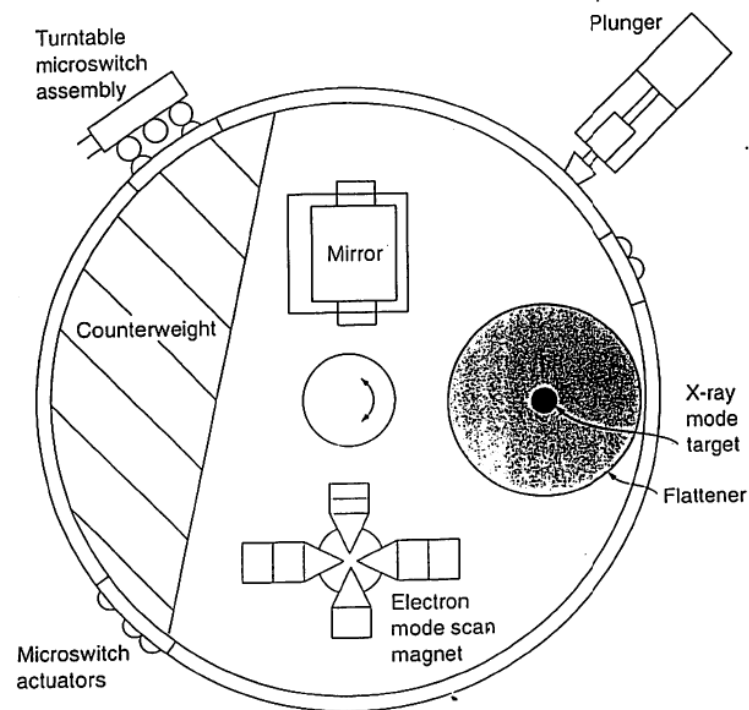


Figure 1. Typical Therac-25 facility.

- The Therac-25 was a radiotherapy machine sold by AECL
- Between 1985 and 1987 three people died in six accidents
- Example of a fatal coding error, compounded with usability problems and poor safety engineering

The Therac Accidents (2)



- 25 MeV 'therapeutic accelerator' with two modes of operation
 - 25MeV focused electron beam on target to generate X-rays
 - 5-25MeV spread electron beam for skin treatment (with 1% of beam current)
- Safety requirement: don't fire 100% beam at human!

The Therac Accidents (3)

- Previous models (Therac 6 and 20) had mechanical interlocks to prevent high-intensity beam use unless X-ray target in place
- The Therac-25 replaced these with software
- Fault tree analysis arbitrarily assigned probability of 10^{-11} to 'computer selects wrong energy'
- Code was poorly written, unstructured and not really documented

The Therac Accidents (4)

- Marietta, GA, June 85: woman's shoulder burnt. Settled out of court. FDA not told
- Ontario, July 85: woman's hip burnt. AECL found microswitch error but could not reproduce fault; changed software anyway
- Yakima, WA, Dec 85: woman's hip burned. 'Could not be a malfunction'

The Therac Accidents (5)

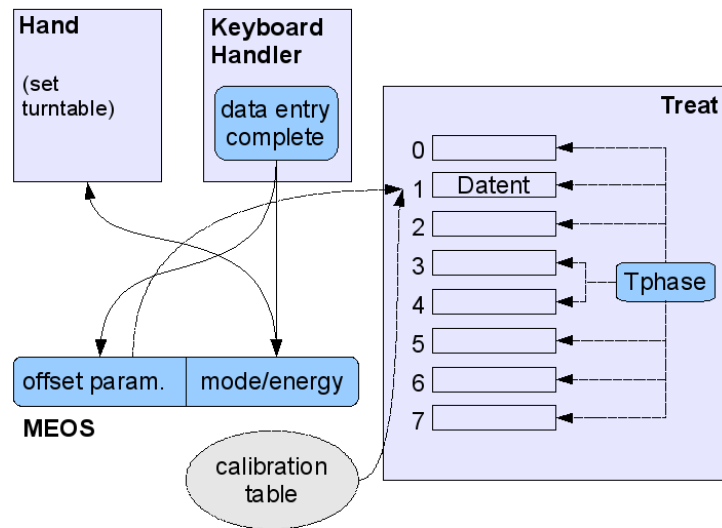
- East Texas Cancer Centre, Mar 86: man burned in neck and died five months later of complications
- Same place, three weeks later: another man burned on face and died three weeks later
- Hospital physicist managed to reproduce flaw: if parameters changed too quickly from x-ray to electron beam, the safety interlock failed
- Yakima, WA, Jan 87: man burned in chest and died – due to different bug now thought to have caused Ontario accident

The Therac Accidents (6)

PATIENT NAME	: TEST		
TREATMENT MODE	: FIX	BEAM TYPE: X	ENERGY (MeV): 25
		ACTUAL	PRESCRIBED
UNIT RATE/MINUTE		0	200
MONITOR UNITS		50 50	200
TIME (MIN)		0.27	1.00
GANTRY ROTATION (DEG)		0.0	0 VERIFIED
COLLIMATOR ROTATION (DEG)		359.2	359 VERIFIED
COLLIMATOR X (CM)		14.2	14.3 VERIFIED
COLLIMATOR Y (CM)		27.2	27.3 VERIFIED
WEDGE NUMBER		1	1 VERIFIED
ACCESSORY NUMBER		0	0 VERIFIED
DATE	: 84-OCT-26	SYSTEM : BEAM READY	OP. MODE : TREAT AUTO
TIME	: 12:55: 8	TREAT : TREAT PAUSE	X-RAY 173777
OPR ID	: T25V02-R03	REASON : OPERATOR	COMMAND:

- East Texas deaths caused by editing 'beam type' too quickly
- This was due to poor software design

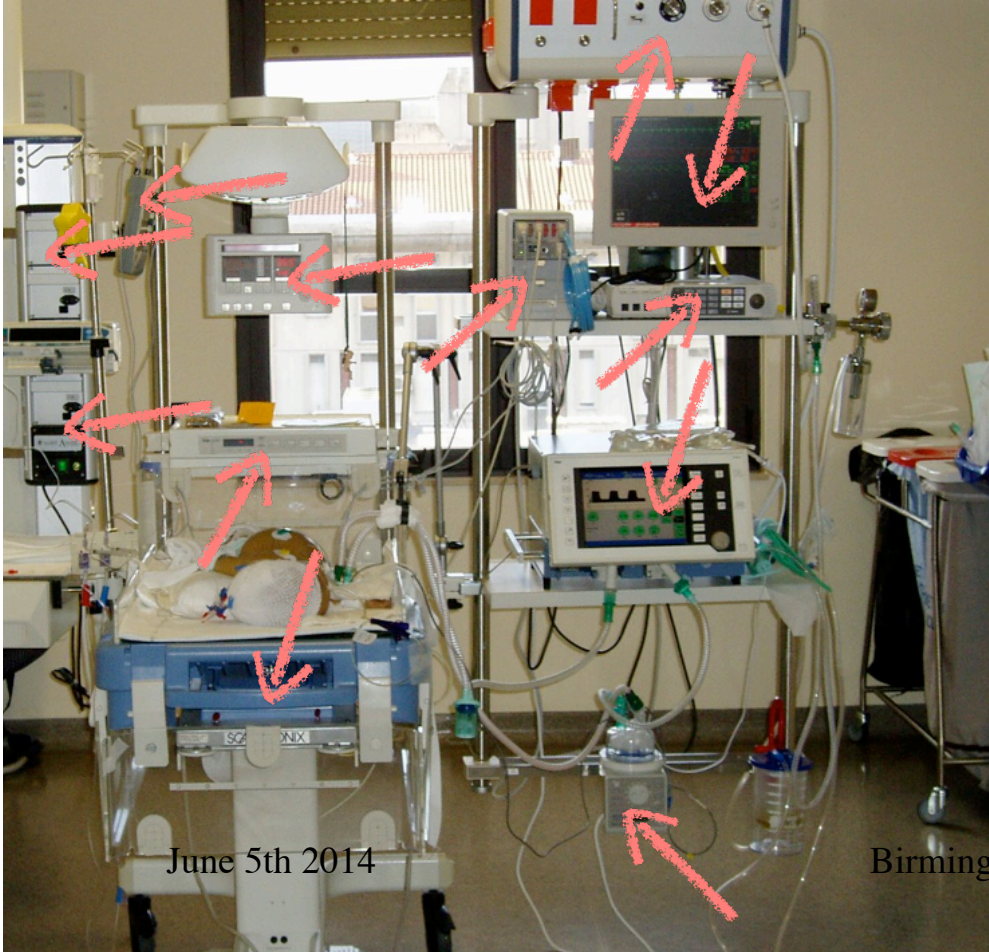
The Therac Accidents (7)



- Datent sets turntable and 'MEOS', which sets mode and energy level
- 'Data entry complete' can be set by datent, or keyboard handler
- If MEOS set (& datent exited), then MEOS could be edited again

The Therac Accidents (8)

- AECL had ignored safety aspects of software
- Confused reliability with safety
- Lack of defensive design
- Inadequate reporting, followup and regulation – didn't explain Ontario accident at the time
- Unrealistic risk assessments
- Inadequate software engineering practices – spec an afterthought, complex architecture, dangerous coding, little testing, careless HCI design...
- AECL got out of the medical equipment business. But similar accidents are still happening! (NY Times article)
- Poor medical device safety usability still costs many lives



June 5th 2014

Birmingham



June 5th 2014

Birmingham

1	2	3
4	5	6
7	8	9
▲	0	▽

Abbott Gemstar

7	8	9
4	5	6
1	2	3
▲	0	▽

Abbott AimPlus

1	2	3
4	5	6
7	8	9
●	▽	

CME BodyGuard 545

1	2	3	4	5	●
6	7	8	9	▽	

CME BodyGuard 545

▲	2	▽
4	5	6
7	8	9
	0	●

Graseby 500

1	2	3	4
5	6	7	8
9	0	●	C

Graseby Omnifuse

1	2	3	
4	5	6	●
7	8	9	0

SK Medical SK-500III

7	8	9
4	5	6
1	2	3
0	●	

SK Medical SK-600III

1	2	3	4
5	6	7	8
C	9	0	△
	●	▽	

Upreal UPR-900

1	2	3	●
4	5	6	0
7	8	9	

Upreal CTN-TCI-V

7	8	9	0
4	5	6	C
1	2	3	●

BBraun Vista Basic

7	8	9	0
4	5	6	●
1	2	3	C

DRE SP1500 Plus

1	2	3	4	
5	6	7	8	C
	●	0	9	

DRE Avanti Plus

0	1	2	3
●	4	5	6
	7	8	9

Sigma Spectrum

7	8	9
4	5	6
1	2	3
0	●	

Birmingham
Sigma 6000 Plus

1	2	3
4	5	6
7	8	9
	0	●

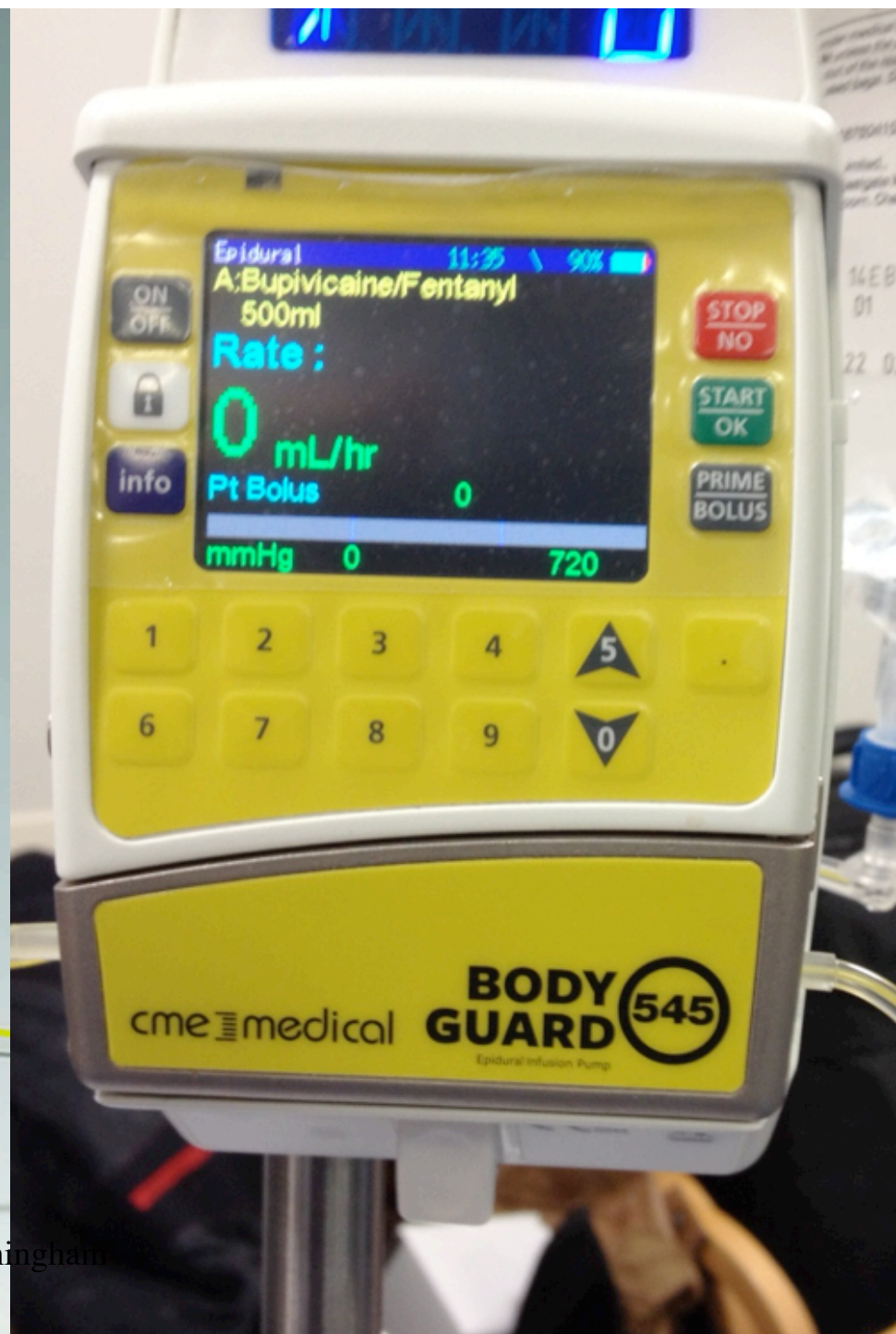
Sigma 8000 Plus

June 5th 2014



June 5th 2014

Birmingham

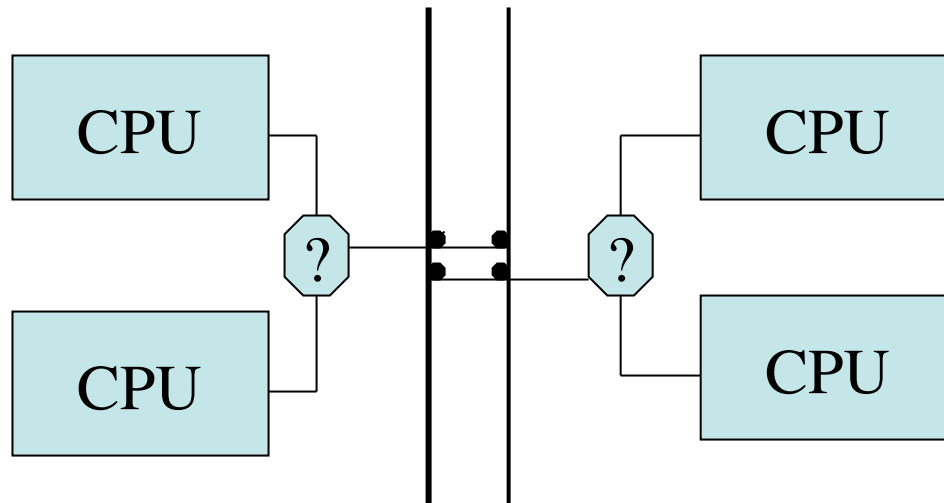


Medical device safety

- Usability problems with medical devices kill about the same number of people as cars
- Biggest killer nowadays: infusion pumps
- Regulators are incompetent / captured
- Nurses get blamed for fatalities
- Avionics are safer, as incentives are better (airlines and pilots don't want crashes)
- Read Harold Thimbleby's paper!

Redundancy

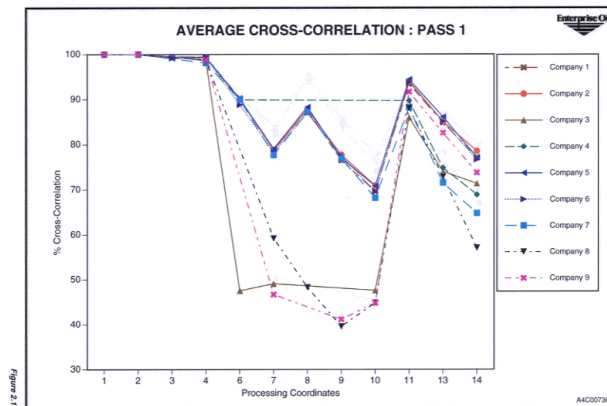
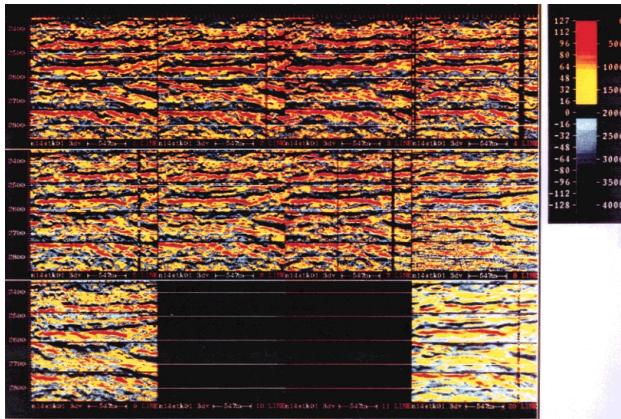
- Some vendors, like Stratus, developed redundant hardware for 'non-stop processing'



Redundancy (2)

- Stratus users found that the software is then where things broke
- The 'backup' IN set in Ariane failed first!
- Next idea: multi-version programming
- But: errors significantly correlated, and failure to understand requirements comes to dominate (Knight/Leveson 86/90)
- Professional implementations often give different answers...

Example: seismic analysis



- Nine separate companies sell software to do standard processing on seismic data
- Given the same inputs, their outputs differ significantly (Hatton/Roberts 94; Hatton, Graham-Cummin, Ince 2010)
- How do you manage this?

Redundancy Management – 737



Panama crash, June 6 1992



- Need to know which way up!
- New EFIS (each side), old artificial horizon in middle
- EFIS failed – loose wire
- Both EFIS fed off same IN set
- Pilots watched EFIS, not AH
- 47 fatalities
- And again: Korean Air cargo 747, Stansted Dec 22 1999

Kegworth crash, Jan 8 1989



- BMI London-Belfast, fan blade broke in port engine
- Crew shut down starboard engine and did emergency descent to East Midlands
- Opened throttle on final approach: no power
- 47 dead, 74 injured
- Initially blamed wiring technician! Later: cockpit design

Complex Socio-technical Systems

- Aviation is actually an easy case as it's a mature evolved system!
- Stable components: aircraft design, avionics design, pilot training, air traffic control ...
- Interfaces are stable too
- The capabilities of crew are known to engineers
- The capabilities of aircraft are known to crew, trainers, examiners
- The whole system has good incentives for learning

Cognitive Factors

- Many errors derive from highly adaptive mental processes
 - E.g., we deal with novel problems using knowledge, in a conscious way
 - Then, trained-for problems are dealt with using rules we evolve, and are partly automatic
 - Over time, routine tasks are dealt with automatically – the rules have give way to skill
- But this ability to automatise routine actions leads to absent-minded slips, aka ‘capture errors’

Cognitive Factors (2)

- Read up the psychology that underlies errors!
- Slips and lapses
 - Forgetting plans, intentions; strong habit intrusion
 - Misidentifying objects, signals (often Bayesian)
 - Retrieval failures; tip-of-tongue, interference
 - Premature exits from action sequences, e.g. ATMs
- Rule-based mistakes; applying wrong procedure
- Knowledge-based mistakes; heuristics and biases

Cognitive Factors (3)

- Training and practice help – skill is more reliable than knowledge! Error rates (motor industry):
 - Inexplicable errors, stress free, right cues – 10^{-5}
 - Regularly performed simple tasks, low stress – 10^{-4}
 - Complex tasks, little time, some cues needed – 10^{-3}
 - Unfamiliar task dependent on situation, memory – 10^{-2}
 - Highly complex task, much stress – 10^{-1}
 - Creative thinking, unfamiliar complex operations, time short & stress high – ~ 1

Where should the path be?



Cognitive Factors (4)

- Violations of rules also matter: they're often an easier way of working, and sometimes necessary
- 'Blame and train' as an approach to systematic violation is suboptimal
- The fundamental attribution error
- The 'right' way of working should be easiest: look where people walk, and lay the path there
- Need right balance between 'person' and 'system' models of safety failure

Cognitive Factors (5)


- Ability to perform certain tasks can vary widely across subgroups of the population
- Age, sex, education, ... can all be factors
- Risk thermostat – function of age, sex
- Example: banks tell people ‘parse URLs’
- Baron-Cohen: people can be sorted by SQ (systematizing) and EQ (empathising)
- Is this correlated with ability to detect phishing websites by understanding URLs?

Online Safety & Personality Survey :: FF1 - Mozilla Firefox


File Edit View History Bookmarks Tools Help

http://www.tylerwmoore.com/limesurvey/index.php

Getting Started Latest Headlines

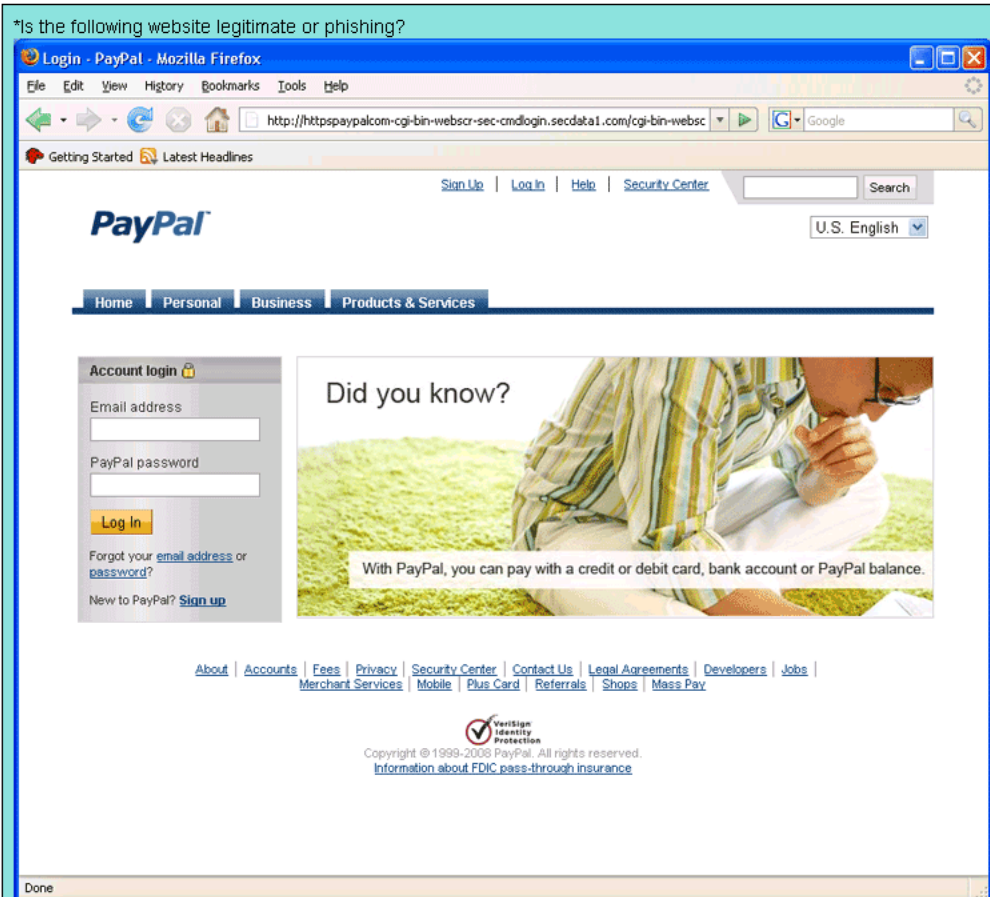
 **UNIVERSITY OF CAMBRIDGE**

Online Safety & Personality Survey

0%  100%

FF1

*Is the following website legitimate or phishing?



Done

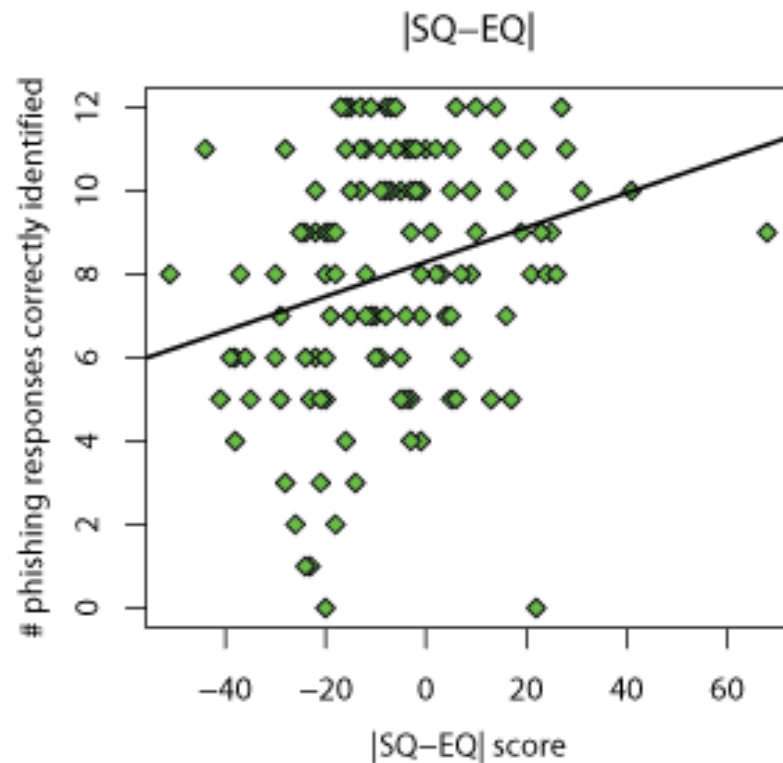
Choose one of the following answers

☐ Legitimate website

☐ Illegitimate phishing website

☐ I'm not sure

Results



- Ability to detect phishing is correlated with SQ-EQ
- It is (independently) correlated with gender
- The 'gender HCI' issue applies to security too

Cognitive Factors (6)

- People's behaviour is also strongly influenced by the teams they work in
- Social psychology is a huge subject!
- Also selection effects – e.g. risk aversion
- Some organisations focus on inappropriate targets (King's Cross fire)
- Add in risk dumping, blame games
- It can be hard to state the goal honestly!

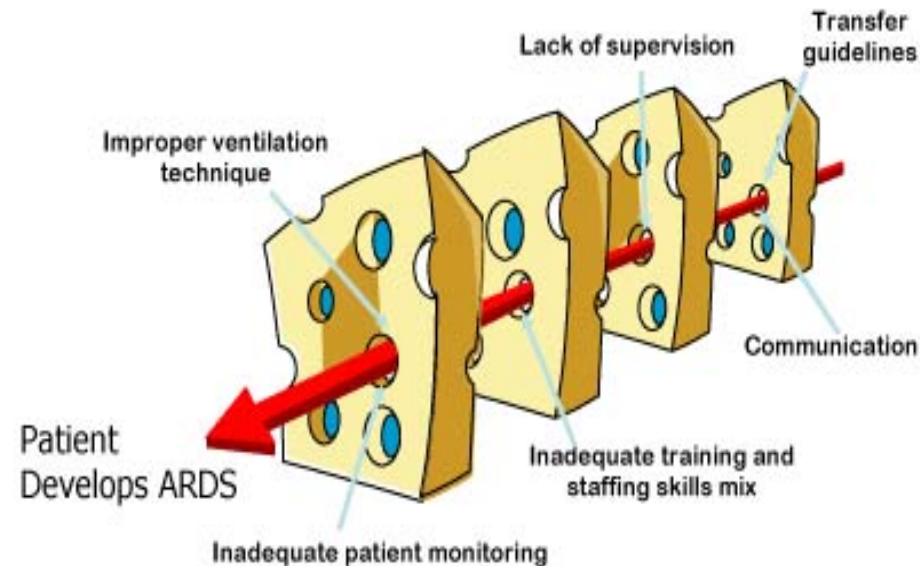
Software Safety Myths (1)

- ‘Computers are cheaper than analogue devices’
 - Shuttle software cost \$10⁸ pa to maintain
- ‘Software is easy to change’
 - Exactly! But it’s hard to change safely
- ‘Computers are more reliable’
 - Shuttle software had 16 potentially fatal bugs found since 1980 – and half of them had flown
- ‘Increasing reliability increases safety’
 - They’re correlated but not completely

Software Safety Myths (2)

- ‘Formal verification can remove all errors’
 - Not even for 100-line programs
- ‘Testing can make software arbitrarily reliable’
 - For MTBF of 10^9 hours you must test $>10^9$ hours
- ‘Reuse increases safety’
 - Not in Ariane, Patriot and Therac, it didn’t
- ‘Automation can reduce risk’
 - Sure, if you do it right – which often takes an extended period of socio-technical evolution

Defence in Depth



- Reason's 'Swiss cheese' model
- Stuff fails when holes in defence layers line up
- Thus: ensure human factors, software, procedures complement each other

Pulling it Together

- First, understand and prioritise hazards. E.g. the motor industry uses:
 1. Uncontrollable: outcomes can be extremely severe and not influenced by human actions
 2. Difficult to control: very severe outcomes, influenced only under favourable circumstances
 3. Debilitating: usually controllable, outcome at worst severe
 4. Distracting; normal response limits outcome to minor
 5. Nuisance: affects customer satisfaction but not normally safety

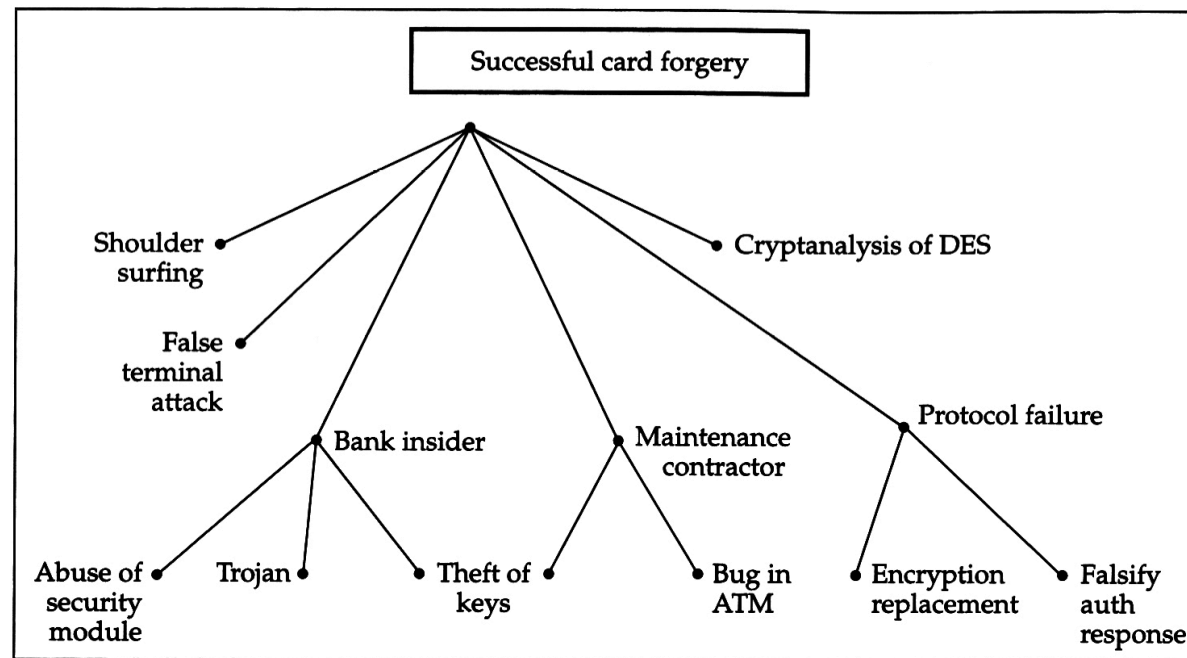
Pulling it Together (2)

- Develop safety case: hazards, risks, and strategy per hazard (avoidance, constraint)
- Who will manage what? Trace hazards to hardware, software, procedures
- Trace constraints to code, and identify critical components / variables to developers
- Develop safety test plans, procedures, certification, training, etc
- Figure out how all this fits with your development methodology (waterfall, spiral, evolutionary ...)

Pulling it Together (3)

- Managing relationships between component failures and outcomes can be bottom-up or top-down
- Bottom-up: ‘failure modes and effects analysis’ (FMEA) – developed by NASA
- Look at each component and list failure modes
- Then use secondary mechanisms to deal with interactions
- Software not within original NASA system – but other organisations apply FMEA to software

Pulling it Together (4)



- Top-down – ‘fault tree analysis’ (in security, a threat tree)
- Work back from identified hazards to identify critical components

Pulling it Together (5)

- Managing a critical property – safety, security, real-time performance – is hard
- Although some failures happen during the ‘techie’ phases of design and implementation, most happen before or after
- The soft spots are requirements engineering, and operations / maintenance later
- These are the interdisciplinary phases, involving systems people, domain experts and users, cognitive factors, and institutional factors like politics, marketing and certification

Keeping it Together

- The emerging problem with the “Internet of Things” is that safety now includes security
- Put software everywhere, and attacks scale!
- Things like cars, medical devices and grid equipment have 10-year certification cycles
- The Panix lesson
- Expect everything to go to monthly updates
- This will really stress a lot of regulators!

Tools

- Homo sapiens uses tools when some parameter of a task exceeds our native capacity
 - Heavy object: raise with lever
 - Tough object: cut with axe
 - ...
- Software engineering tools are designed to deal with complexity

Tools (2)

- There are two types of complexity:
 - **Incidental complexity** dominated programming in the early days, e.g. keeping track of stuff in machine-code programs. Solution: high-level languages
 - **Intrinsic complexity** is the main problem today, e.g. complex system (such as a bank) with a big team.
‘Solution’: structured development, project management tools, ...
- We can aim to eliminate the incidental complexity, but the intrinsic complexity must be managed

Incidental Complexity (1)

- The greatest single improvement was the invention of high-level languages like FORTRAN
 - 2000 loc/year goes much farther than assembler
 - Code easier to understand and maintain
 - Appropriate abstraction: data structures, functions, objects rather than bits, registers, branches
 - Structure lets many errors be found at compile time
 - Code may be portable; at least, the machine-specific details can be contained
- Performance gain: 5–10 times. As coding = 1/6 cost, better languages give diminishing returns

Incidental Complexity (2)

- Thus most advances since early HLLs focus on helping programmers structure and maintain code
- Don't use 'goto' (Dijkstra 68), structured programming, pascal (Wirth 71); info hiding plus proper control structures
- OO: Simula (Nygaard, Dahl, 60s), Smalltalk (Xerox 70s), C++, Java ... covered elsewhere (but do see 'Objects have failed' on the course page)
- Don't forget the object of all this is to manage complexity!

Incidental Complexity (3)

- Early batch systems were very tedious for developer ... e.g. GSCS
- Time-sharing systems allowed online test – debug – fix – recompile – test – ...
- This still needed plenty scaffolding and carefully thought out debugging plan
- Integrated programming environments such as TSS, Turbo Pascal,...
- Some of these started to support tools to deal with managing large projects – ‘CASE’

Formal Methods

- Pioneers such as Turing talked of proving programs correct
- Floyd (67), Hoare (71), ... now a wide range:
 - Z for specifications
 - HOL for hardware
 - BAN for crypto protocols
- These are not infallible (a kind of multiversion programming) but can find a lot of bugs, especially in small, difficult tasks
- Not much use for big systems

Programming Philosophies

- ‘Chief programmer teams’ (IBM, 70–72): capitalise on wide productivity variance
- Team of chief programmer, apprentice, toolsmith, librarian, admin assistant etc, to get maximum productivity from your staff
- Can be effective during implementation
- But each team can only do so much
- Why not just fire the less productive programmers?

Programming Philosophies (2)

- ‘Egoless programming’ (Weinberg, 71) – code should be owned by the team, not by any individual. In direct opposition to chief programmer team
 - But: groupthink entrenches bad stuff more deeply
- ‘Literate programming’ (Knuth et al) – code should be a work of art, aimed not just at machine but also future developers
 - But: creeping elegance is often a symptom of a project slipping out of control

Programming Philosophies (3)

- ‘Extreme Programming’ (Beck, 99): aimed at small teams working on iterative development with automated tests and short build cycle
- ‘Solve your worst problem. Repeat’
- Focus on development episode: write tests first, then the code. ‘The tests are the documentation’
- Programmers work in pairs, at one keyboard and screen
- That didn’t survive, but episodes did, and people added the ‘scrum’

Capability Maturity Model

- Humphrey, 1989: it's important to keep teams together, as productivity grows over time
- Nurture the capability for repeatable, manageable performance, not outcomes that depend on individual heroics
- CMM developed at CMU with DoD money
- It identifies five levels of increasing maturity in a team or organisation, and a guide for moving up

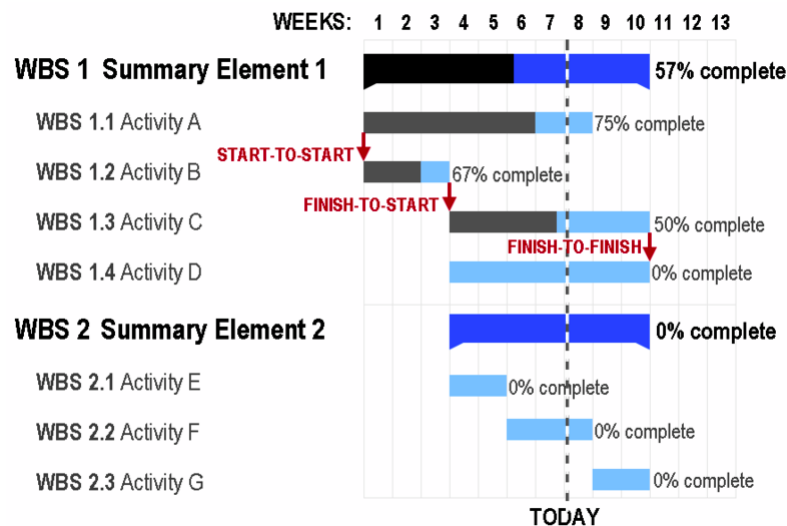
Capability Maturity Model (2)

1. Initial (chaotic, ad hoc) – the starting point for use of a new process
2. Repeatable – the process is able to be used repeatedly, with roughly repeatable outcomes
3. Defined – the process is defined/confirmed as a standard business process
4. Managed – the process is managed according to the metrics described in the Defined stage
5. Optimized – process management includes deliberate process optimization/improvement

Project Management

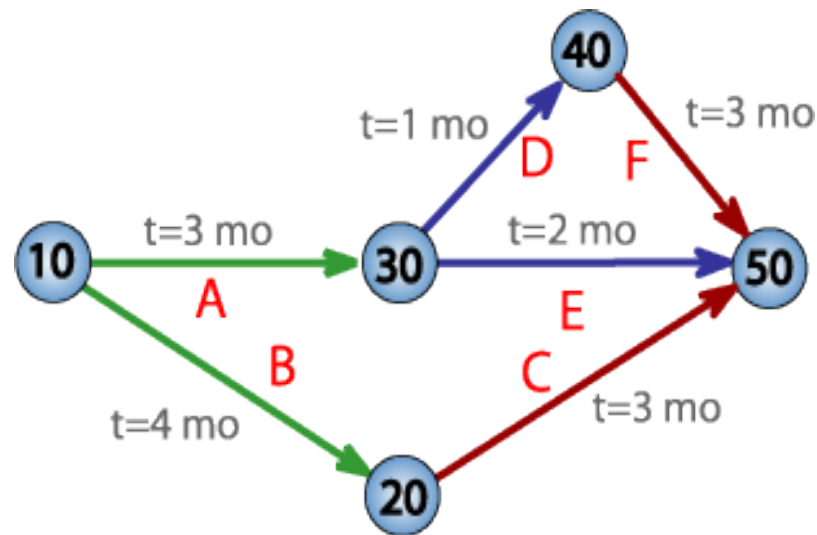
- A manager's job is to
 - Plan
 - Motivate
 - Control
- The skills involved are interpersonal, not techie; but managers must retain respect of techie staff
- Growing software managers a perpetual problem!
'Managing programmers is like herding cats'
- Nonetheless there are some tools that can help

Activity Charts



- 'Gantt' chart (after inventor) shows tasks and milestones
- Problem: can be hard to visualise dependencies

Critical Path Analysis



- Project Evaluation and Review Technique (PERT): draw activity chart as graph with dependencies
- Give critical path (here, two) and shows slack
- Can help maintain 'hustle' in a project
- Also helps warn of approaching trouble

Keeping People Motivated

- People can work less hard in groups than on their own projects – ‘free rider’ or ‘social loafing’ effect
- Competition doesn’t invariably fix it: people who don’t think they’ll win stop trying
- Dan Rothwell’s ‘three C’s of motivation’:
 - Collaboration – everyone has a specific task
 - Content – everyone’s task clearly matters
 - Choice – everyone has a say in what they do
- Many other factors: acknowledgement, attribution, equity, leadership, and ‘team building’ (shared food / drink / exercise; scrumming)

Testing

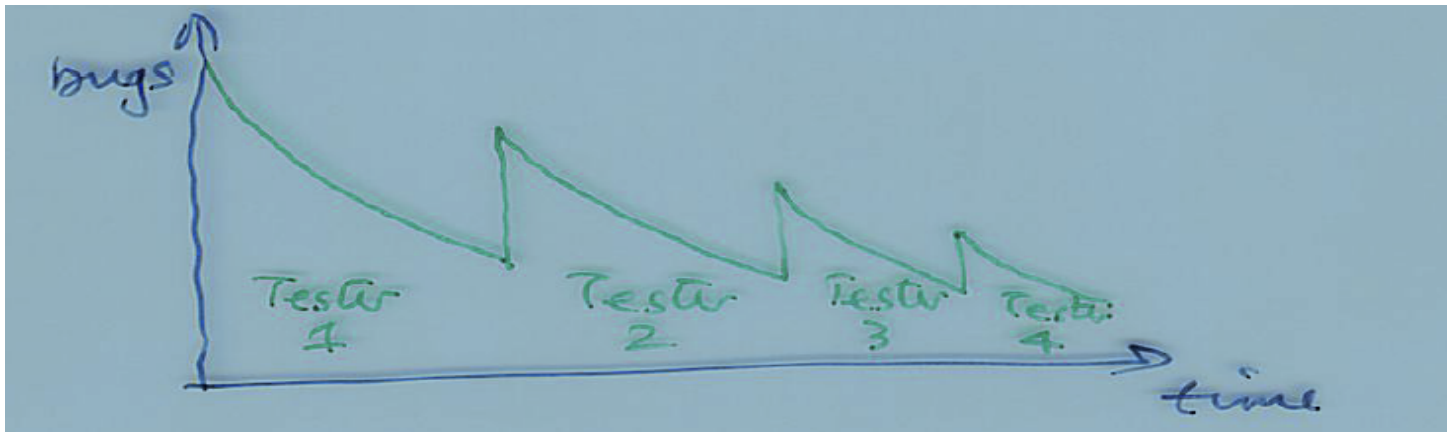
- Testing is often neglected in academia, but is the focus of industrial interest ... it's half the cost
- Bill G: “are we in the business of writing software, or test harnesses?”
- Happens at many levels
 - Design validation
 - Module test after coding
 - System test after integration
 - Beta test / field trial
 - Subsequent litigation
- Cost per bug rises dramatically down this list!

Testing (2)

- Main advance since 2000 is design for testability, continuous integration, and automated regression tests
- Tests check that new versions of the software give same answers as old version
 - Customers more upset by failure of a familiar feature than at a new feature which doesn't work right
 - Without regression testing, 20% of bug fixes reintroduce failures in already tested behaviour
 - Test the inputs that your users actually generate!

Testing (3)

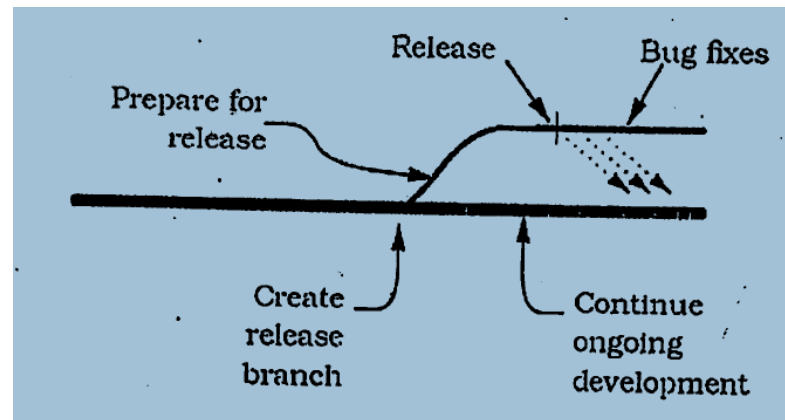
- Reliability growth models help us assess mtbf, number of bugs remaining, economics of further testing...
- Failure rate due to one bug is $e^{-k/T}$; with many bugs these sum to k/T
- So for 10^9 hours mtbf, must test $>10^9$ hours
- But: changing testers brings new bugs to light



Testing (4)

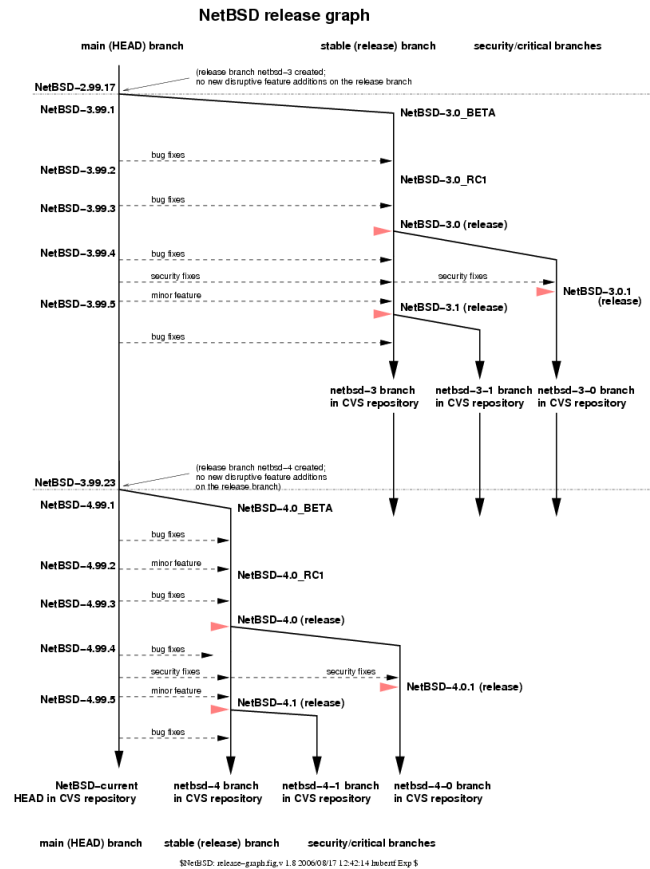
- The critical problem with testing is to exercise the conditions under which the system will actually be used
- Many failures result from unforeseen input / environment conditions (e.g. Patriot)
- Incentives matter hugely: commercial developers often look for friendly certifiers while military arrange hostile review (ditto manned spaceflight, nuclear)

Release Management



- Getting from development code to production release can be nontrivial!
- Main focus is stability – work on recently-evolved code, test with lots of hardware versions, etc
- Add all the extras like copy protection, rights management
- Do you patch old versions, or force upgrades?

Example – NetBSD Release



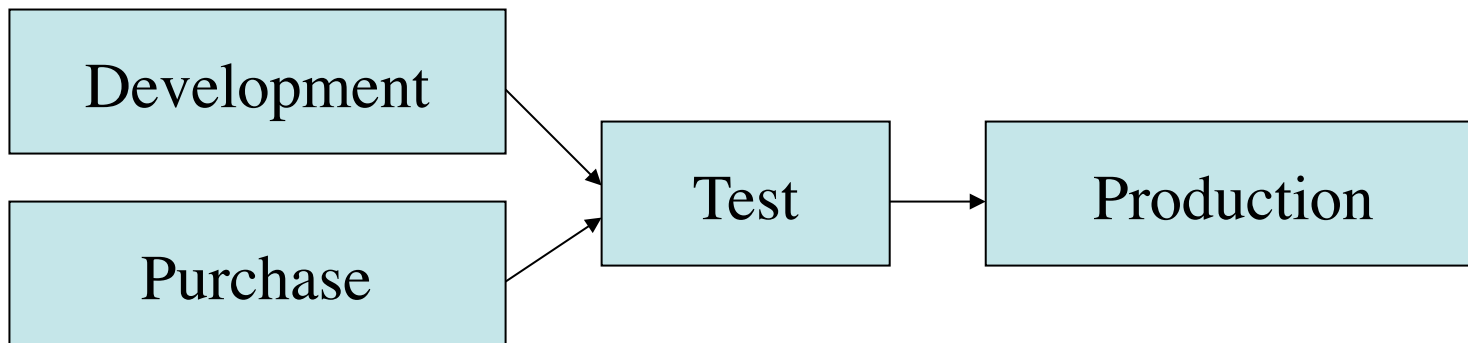
- Beta testing of release
- Then security fixes
- Then minor features
- Then more bug fixes
- ...

Shared Infrastructure

- We share a lot of code through open source operating systems, libraries and tools
- Huge benefits but also interaction issues!
- Can you cope with an emergency bug fix (e.g. a deadly OpenSSL flaw)?
- How do you feed your fixes back to others?
- Do you encourage responsible disclosure?
- Are you aware of different license terms?

Change Control

- Change control and configuration management are critical yet often poor
- The objective is to control the process of testing and deploying software you've written, or bought, or got fixes for
- Someone must assess the risk and take responsibility for live running, and look after backup, recovery, rollback etc



Documentation

- Think: how will you deal with management documents (budgets, PERT charts, staff schedules)
- And engineering documents (requirements, hazard analyses, specifications, test plans, code)?
- CS tells us it's hard to keep stuff in synch!
- Possible partial solutions:
 - High tech: integrated development environment
 - Bureaucratic: plans and controls department
 - Social consensus: style, comments, formatting

Problems of Large Systems

- Study of failure of 17 large demanding systems, Curtis Krasner and Iscoe 1988
- Causes of failure
 1. Thin spread of application domain knowledge
 2. Fluctuating and conflicting requirements
 3. Breakdown of communication, coordination
- They were very often linked, and the typical progression to disaster was $1 \rightarrow 2 \rightarrow 3$

Problems of Large Systems (2)

- Thin spread of application domain knowledge
 - How many people understand everything about running a phone service / bank / hospital?
 - Many aspects are jealously guarded secrets
 - Some fields try hard, e.g. pilot training
 - Or with luck you might find a real ‘guru’
 - But you can expect specification mistakes
- The spec may change in midstream anyway
 - Competing products, new standards, fashion
 - Changing environment (takeover, election, ...)
 - New customers (e.g. overseas) with new needs

Problems of Large Systems (3)

- Comms problems inevitable – N people means $N(N-1)/2$ channels and 2^N subgroups
- Traditional way of coping is hierarchy; but if info flows via 'least common manager', bandwidth will be inadequate
- So you proliferate committees, staff departments
- This causes politicking, blame shifting
- Management attempts to gain control result in restricting many interfaces, e.g. to the customer

Agency Issues

- Employees often optimise their own utility, not the projects; e.g. managers don't pass on bad news
- Prefer to avoid residual risk issues: risk reduction becomes due diligence
- Tort law reinforces herding behaviour: negligence judged 'by the standards of the industry'
- Cultural pressures in e.g. aviation, banking
- So: do the checklists, use the tools that will look good on your CV, hire the big consultants...

Conclusions

- Software engineering is about managing complexity. That's why it's hard. That's our trade
- We can cut incidental complexity using tools
- But the intrinsic complexity remains: you manage it by getting early commitments, partitioning the problem, using project management, ...
- Top-down approaches can sometimes help, but really large systems evolve
- The grand challenge facing engineers over the next 25 years will be learning how to direct the evolution of complex socio-technical systems

Conclusions (2)

- Things are made harder by the fact that complex systems are usually socio-technical
- People come into play as users, and also as members of development and other teams
- About 30% of big commercial projects fail, and about 30% of big government projects succeed. This has been stable for years, despite better tools!
- Better tools let people climb a bit higher up the complexity mountain before they fall off
- But the limiting factors are human too