Software Engineering

CST IA/IIG/Dip
Alan Blackwell

OUTLINE OF COURSE

- The ‘Software Crisis’
- The Software Life Cycle
- Critical Software
- Quality Assurance
- Tools and Methods
- Large Systems
- Final lecture given by Dr Robert Brady of Brady plc. on developing packaged software.

RESOURCES

- The newsgroup comp.risks
- R S Pressman ***Software Engineering***
- N Leveson ***Safeware***
- Additional reading:
  - F P Brooks ***The Mythical Man Month***
  - P Newman ***Computer-Related Risks***
  - L R Wiener ***Digital Woes***
  - Finkelstein inquiry reports: ***London Ambulance Service*** & ***CAPSA and it's Implementation*** www.cs.ucl.ac.uk/staff/A.Finkelstein/
- Also recommended
  - wide reading in whichever application area(s) interest you (aviation, healthcare, banking,......)

The 'Software Crisis'

- The reality of software development has lagged behind the apparent promise of the hardware
- Most large projects fail - either abandoned, or do not deliver anticipated benefits
  - LSE Taurus £ 400 m
  - Denver Airport $ 200 m
  - CONFIRM $ 160 m
- Some software failures cost lives or cause large material losses
  - Therac 25
  - Ariane
  - Pentium
  - NY Bank - and Y2K in general
- Some combine project failure with loss of life, e.g. London Ambulance Service

London Ambulance Service

- Existing manual operation:
  - 999 calls written on forms
  - map reference looked up
  - conveyor belt to central point
  - controller removes duplicates, passes to NE/NW/S district
  - division controller identifies vehicle and puts note in its 'activation box'
  - form passed to radio dispatcher
- Takes about 3 minutes, and 200 staff (of 2,700 total).
  - some errors (esp. deduplication),
  - some queues (esp. radio),
  - call-backs are laborious to deal with

LAS: Project Background

- Attempt to automate in 1980's failed - the system failed load test
- Industrial relations poor - pressure to cut costs
- Decided to go for fully automated system:
  - controller answering 999 call would have on-screen map
  - could send 'email' directly to ambulance
- Consultancy study to assess feasibility:
  - estimated cost £1.5m, duration 19 months …
  - ***provided*** a packaged solution could be found
  - excluding an automatic vehicle location system (AVLS)

LAS: Award of Tender

- Idea of a £1.5m system stuck, ***but***
  - AVLS added
  - proviso of packaged solution forgotten
  - new IS director hired
  - tender put out 7 February 1991
  - completion deadline January 1992
- 35 firms looked at tender
  - 19 submitted proposals, most said:
    - timescale unrealistic
    - only partial automation possible by January 1992
- Tender awarded to consortium:
  - Systems Options Ltd, Apricot and Datatrak
  - bid of £937,463 … £700K cheaper than next bidder

LAS: Design Phase

- Design work 'done' July
- main contract August
- mobile data subcontract September
- in December told only partial implementation possible in January –
  - front end for call taking
  - gazetteer + docket printing
- by June 91, a progress meeting had minuted:
  - 6 month timescale for 18 month project
  - methodology unclear, no formal meeting program
  - LAS had no full time user on project
- Systems Options Ltd relied on 'cozy assurances' from subcontractors

LAS: Implementation

- Problems apparent with ‘phase 1’ system
  - client & server lockup
- ‘Phase 2’ introduced radio messaging, further problems
  - blackspots, channel overload at shift change,
  - inability to cope with ‘established working practices’ such as taking the ‘wrong’ ambulance
- System never stable in 1992
- Management pressure for full system to go live
  - including automatic allocation
  - CE said: ‘no evidence to suggest that the full system software, when commissioned, will not prove reliable’

LAS: Live Operation

- Independent review had noted need for:
  - volume testing
  - written implementation strategy
  - change control
  - training
  - … it was ignored.
- 26 October
  - control room reconfigured to use terminals not paper
  - resource allocators separated from radio operators and exception rectifiers
  - No backup system.
  - No network managers.

LAS: 26 & 27 October - Disaster

- Vicious cycle of failures
  - system progressively lost track of vehicles
  - exception messages built up, scrolled off screen, were lost
  - incidents held as allocators searched for vehicles
  - callbacks from patients increased workload
  - data delays - voice congestion - crew frustration - pressing wrong buttons and taking wrong vehicles
  - many vehicles sent, or none
  - slowdown and congestion proceeded to collapse
- Switch back to semi-manual operation on 27 Oct
- Irretrievable crash 02:00 4 Nov due to memory leak:
  - 'unlikely that it would have been detected through conventional programmer or user testing'
- Real reason for failure: poor management throughout

The Software Crisis

- Emerged during 1960's
  - large and powerful mainframes (e.g. IBM 360) made far larger and more complex systems possible
  - why did software projects suffer failures & cost overruns so much more than large civil, structural, aerospace engineering projects?
- Term 'software engineering' coined 1968
  - hope that engineering habits could get things under control
  - e.g. project planning, documentation, testing
- These techniques certainly help – we'll discuss
- But first:
  - how does software differ from machinery?
  - what unique problems and opportunities does it bring?

Why is software different?

- Things that make programming ‘fun’ include:

- The joy of making things useful to others
- The fascination of building puzzles from interlocking “moving” parts
- The pleasure of a non-repeating task
  - continuous learning
- The delight of a tractable medium
  - “pure thought stuff”

What makes software hard?

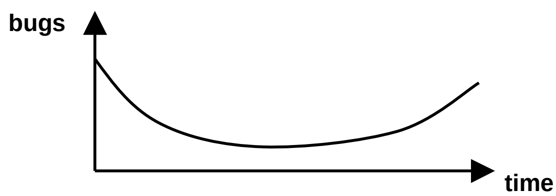
- The need to achieve perfection
- Need to satisfy user objectives, conform with systems, standards, interfaces outside control
- Larger systems qualitatively more complex (unlike ships or bridges) because parts interact in many more than 3 dimensions.
- Tractability of software leads users to demand ‘flexibility’ and frequent changes
- Structure of software can be hard to visualise/model
- Much hard slog of debugging and testing accumulates at project end, when:
  - excitement is gone
  - budget is overspent
  - deadline (or competition) looming

The software life cycle

- Cost of owning a system not just development but whole cost over life cycle:
  - Development, Testing, Operations, Replacement
- In ‘bespoke’ software days
  - 90% of IT department programming effort was maintenance of old systems
- Most research on software costs and methods focuses on this business model.
- Different business models apply
  - to safety critical and related software (lecture 3)
  - to package software (lecture 4)
  - but many lessons apply to them all

Common difficulties

- Code doesn't ‘wear out’ the way that gears in machinery do, but:
  - platform and application requirements change over time,
  - code becomes more complex,
  - it becomes less well documented,
  - it becomes harder to maintain,
  - it becomes more buggy.
- Code failure rates resemble those of machinery
  - (but for different reasons!)



More common difficulties

- When software developed (or redeveloped)
  - unrealistic price/performance expectations
  - as hardware gets cheaper, software seems dear
- Two main causes of project failure
  - incomplete/changing/misunderstood requirements
  - insufficient time
- These and other factors lead to the 'tar pit'
  - any individual problem can be solved
  - but number and complexity get out of control

Life cycle costs

- Development costs (Boehm, 75)

	Reqmts/Spec	Implement	Test
Cm'd & Control	48%	20%	34%
Space	34%	20%	46%
O/S	33%	17%	50%
Scientific	44%	26%	30%
Business	44%	28%	28%

- Maintenance costs: typically ten times as much again

Reducing life cycle costs

- By the late 60’s the industry was realising:

- Well built software cost less to maintain
- Effort spent getting the specification right more than pays for itself by:
  - reducing the time spent implementing and testing
  - reducing the cost of subsequent maintenance.

What does code cost?

- Even if you know how much was spent on a project,
  - how do you measure what has been produced?
  - Does software cost per mile / per gallon / per pound?
- Common measure is KLOC (thousand lines of code)
- First IBM measures (60’s):
  - 1.5 KLOC / man year (operating system)
  - 5 KLOC / man year (compiler)
  - 10 KLOC / man year (app)
- AT&T measures:
  - 0.6 KLOC / man year (compiler)
  - 2.2 KLOC / man year (switch)

More code metrics

- More sophisticated measures:
  - Halstead (entropy of operators, operands)
  - McCabe (graph complexity of control structures)
  - "Function Point Analysis"
- Lessons learned from applying empirical measures:
  - main productivity gains come from using appropriate high level language
    - each KLOC does more
  - wide variation between individuals
    - more than 10 times

Brooks' Law

- Brooks' *The Mythical Man-Month* attacked idea that "men" and months interchangeable, because:
  - more people → more communications complexity
  - adding people → productivity drop as they are trained
- e.g consider project estimated at 3 men x 4 months
  - but 1 month design phase actually takes 2 months!
  - so 2 months left to do work estimated at 9 man-months
  - add 6 men, but training takes 1 month
  - so all 9 man-months work must be done in the last month.
- 3 months work for 3 can't be done in 1 month by 9 (complexity, interdependencies, testing, ...)
- Hence Brooks' Law:

"Adding manpower to a late software project makes it later"

Boehm's empirical study

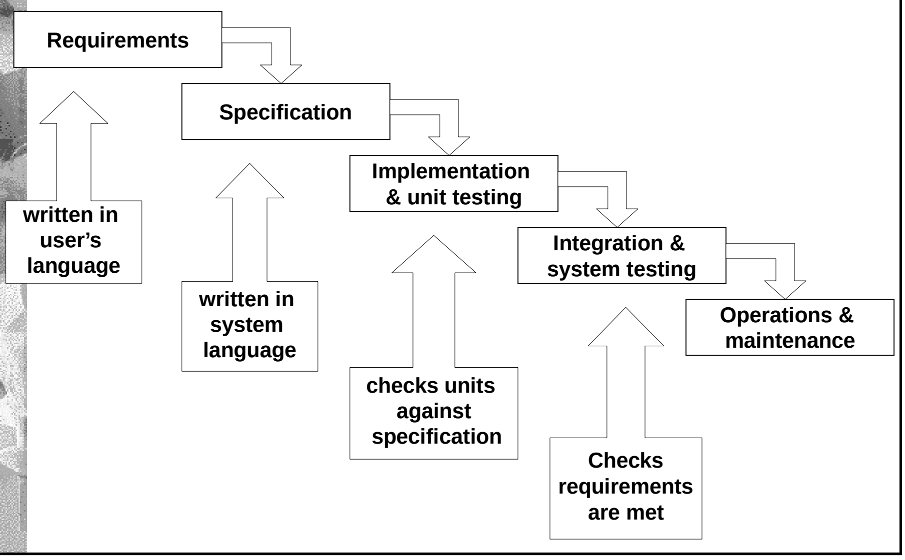
- Brooks' Law (described 1975) led to empirical studies
- Boehm *Software Engineering Economics,* 1981:
  - cost-optimum schedule time to first shipment, T
    - = 2.5 x cube root of total number of man months
  - with more time, cost rises slowly
    - 'people with more time take more time'
  - with less time, the cost rises sharply
  - Hardly any projects succeed in < 0.75T, regardless of number of people employed!
- Other studies show if more people are to be added, should be added early rather than late
- Some projects have more and more resources thrown at them yet are never finished at all (e.g. CONFIRM); others are years late.

Managing with structured design

- Only practical way to build large programs is to divide into modules.
- This enables system architect to control complexity:
  - high level components/subsystems under project teams
    - e.g. general ledger, loans, ATMs
  - divided into modules for individual programmers & testers
    - e.g. calculate interest, update file, ...
- Often subdivision of tasks is straightforward
  - sometimes it isn't
  - sometimes - worst case - it just seems to be!
- Various design *methodologies*
  - e.g. SSADM, Jackson, Yourdon, ...
  - some data driven, others oriented to functionality
  - methodologies & tools discussed in more detail later

The Waterfall Model

- (Royce, 1970; now US DoD standard)



Requirements are developed by at least two groups of people who speak different languages and who come from different disciplines.

Specification, Design and *Implementation* are done by a group of single-discipline professionals who usually can communicate with one another.

Installation is usually done by people who don't really understand the issues or the problem or the solution.

After a start-up period, ***Operation*** is almost always left to people who don't understand the issues, ethics, problem or solution (and often little else).

Maintenance is usually performed by inexperienced people who have forgotten much of what they once knew about the problem or the solution.

New York security consultant Robert Courtney examined 1000s of security breaches - 68% due to careless or incompetent operations.

Feedback in the waterfall model

- *Validation* operations provide feedback
  - from Specification to Requirements
  - from Implementation/unit testing to Specification
- *Verification* operations provide feedback
  - from Integration/ system testing to Implementation/unit testing
  - from operations/maintenance back to Integration/system testing
- What's the difference between 'validation' and 'verification"?
  - Validation: 'are we building the right system?'
  - Verification: 'are we building it right?'
- What about validation from operations back to requirements?
  - this would change the model (and erode much of its value)

Advantages of waterfall model

- Project manager's task easier with clear milestones
- Can charge for requirement changes
  - each stage can even be a separate contract
- System goals, architecture & interfaces clarified together
  - conducive to good design practices
- Compatible with many tools and design methods
- *Where applicable*, waterfall is an ideal approach
  - critical factor: whether requirements can be defined in detail, in advance of any development or prototyping work.
  - sometimes they can (e.g. a compiler);
  - sometimes they can't (e.g. a human-computer interface)

Objections to waterfall model

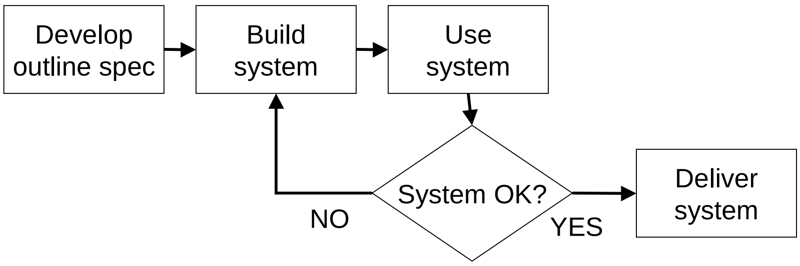
- “Reality isn't like that”
- Iteration is important in the software development process, especially where:
  - requirements not yet understood by development team
  - requirements not yet understood by the customer
  - in some application types e.g. user interfaces
  - technology is changing
  - legal environment is changing
  - customer environment changing, eg one customer to many
- Quality improvement from top-down approach may be unimportant over system lifecycle
- Specific objections from safety-critical and package software developers

A Cautionary Tale

- In 1985, a large bank decided to replace a mixture of old systems with a centralised IBM mainframe
  - Decided to buy in a retail banking package and customise it as they had ‘no experience at specifying a next generation banking system’
  - A proprietary variant of waterfall was adopted.
  - A user team prepared a list of requirements changes needed to adapt the package from its original US environment
- When the system was fielded in the first branches
  - people realised that these changes had made it functionally almost identical to the bank’s old system
  - The many changes also meant that the code was incompatible with the next release of the package
- ’Instant legacy system’ at a nine-figure cost

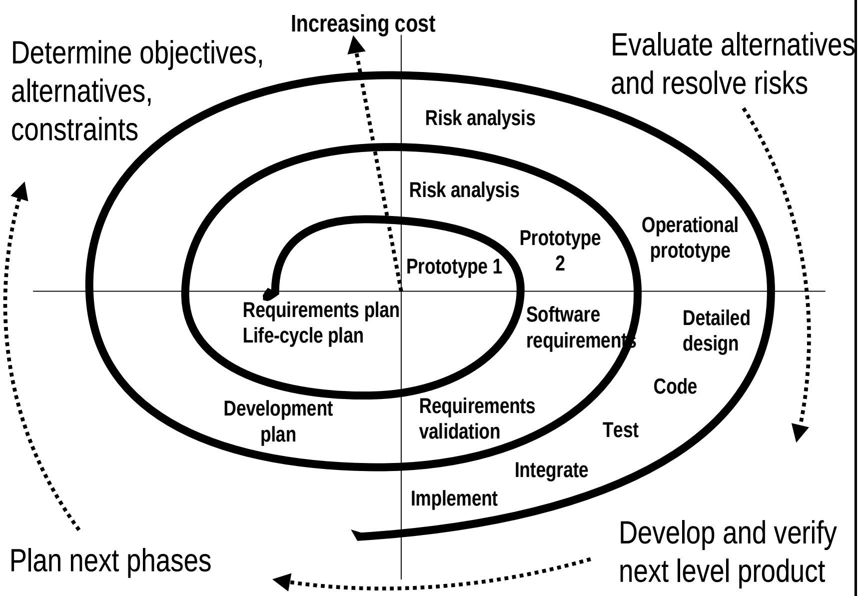
Iterative development

- Some systems need iteration to clarify requirements
- Others make operations fail-safe as possible
- Naive approach:



- This algorithm needn't terminate (satisfactorily)
- Can we combine management benefits of waterfall, with flexibility of iterative development?

Spiral model (Boehm, 88)



Features of spiral model

- fixed number of iterations, each of form:
  - identify alternatives, then
  - assess and choose, then
  - build and evaluate
- conventionally presented as
  - outward spiral from the starting point
  - successive iterations of steps on the same radial
- driven by risk management
- iterative prototyping applied to relevant parts of the system
  - e.g. human computer interface

Critical software

- Many systems have the property that a certain class of failures is to be avoided if at all possible
  - safety critical systems
    - failure could cause death, injury or property damage
  - security critical systems
    - failure could result in leakage of classified data, confidential business data, personal information
  - business critical systems
    - failure could affect essential operations
- Critical computer systems have a lot in common with critical mechanical or electrical systems
  - bridges, flight controls, brakes, locks, ...
- Start out by studying how systems fail

Example - Patriot Missile

- Failed to intercept an Iraqi SCUD missile on 25/2/91; SCUD struck a US barracks in Dhahran
- Other SCUDs got through to Saudi Arabia, Israel
- Reason for failure:
  - measured time in 1/10 sec, truncated from binary representation .0001100110011....
  - as system upgraded from anti-aircraft to anti-missile, greater accuracy introduced - but not everywhere in the code
  - two modules got out of step by 1/3 sec after 100 hours operation. Target not acquired
  - defect not found in testing as the spec called for 14 hour continuous operation only
- Many critical systems failures are multifactorial:
  - 'a reliable system can't fail in a simple way!

Definitions

- **Error**:
  - design flaw or deviation from intended state
- **Failure**:
  - non-performance of the system within some subset of the specified environmental conditions
- **Fault**:
  - Computer science: error $\rightarrow$ fault $\rightarrow$ failure
    - but note electrical engineering terminology: (error $\rightarrow$) failure $\rightarrow$ fault
- **Reliability**:
  - probability of failure within a set period of time
  - Sometimes expressed as 'mean time to (or between) failures' - mttf (or mtbf)

More definitions

- **Accident**
  - undesired, unplanned event that results in a specified kind (and level) of loss
- **Hazard**
  - set of conditions of a system, which together with conditions in the environment, will lead to an accident
  - thus, failure + hazard $\rightarrow$ accident
- **Risk**: hazard level, combined with:
  - **Danger:** probability that hazard $\rightarrow$ accident
  - **Latency:** hazard exposure or duration
- **Safety**: freedom from accidents

System Safety Process

- Obtain support of top management, involve users, and develop a system safety program plan:
  - identify hazards and assess risks
  - decide strategy for each hazard (avoidance, constraint,....)
  - trace hazards to hardware/software interface: which will manage what?
  - trace constraints to code, and identify critical components and variables to developers
  - develop safety-related test plans, descriptions, procedures, code, data, test rigs ...
  - perform special analyses such as iteration of human-computer interface prototype and test
  - develop documentation system to support certification, training ,..
- Safety needs to be designed in from the start. It cannot be retrofitted

Real-time systems

- Many safety critical systems are also *real time*
  - typically used in monitoring or control
- These have particular problems
  - Extensive application knowledge often needed for design
  - Critical timing makes verification techniques inadequate
  - Exception handling particularly problematic.
- eg Ariane 5 (4 June 1996):
  - Ariane 5 accelerated faster than Ariane 4
  - alignment code had an ‘operand error’ on float-to-integer conversion
  - core dumped, core file interpreted as flight data
  - full nozzle deflection → 20 degrees angle of attack → booster separation → self destruct

Hazard Analysis

- Often several hazard categories e.g. Motor Industry Software Reliability Association uses:
  - **Uncontrollable**: failure outcomes not controllable by humans and likely to be extremely severe
  - **Difficult to control**: effects might possibly be controlled, but still likely to lead to very severe outcomes
  - **Debilitating**: effects usually controllable, reduction in safety margin, outcome at worst severe
  - **Distracting**: operational limitations, but a normal human response limits outcome to minor
  - **Nuisance**: affects customer satisfaction, but not normally safety
- Different hazard categories require different failure rates and different levels of investment in varying software engineering techniques

More hazard analysis

- In complex or high-risk systems, may want much more structured hazard analysis
- e.g. US Navy nuclear-capable missile programme:
  - preliminary hazard analysis, leading to
  - system hazard analysis: interfaces between components
  - operating hazard analysis: human machine interfaces
  - maintenance hazard analysis
  - computer program safety analysis
  - subsystem hazard analysis
  - radiation hazard analysis
  - nuclear safety analysis
  - inadvertent launch analysis
  - weapon control interface analysis
- Overlapping and interlocking studies drive the safety programme

THERAC-25

- 25 MEV ‘Therapeutic accelerator’ with two modes of operation:
  - 25 MEV focused electron beam on a target that generates X-rays for treating deep tumours
  - 0.25 MEV spread electron beam for direct treatment of surface tumours
- Patient in shielded room, operator console outside
  - operator confirms dosage settings from console
- Turntable between patient and beam contains:
  - scan magnet for steering low power beam
  - X-ray target to be placed at focus of high power beam
  - plunger to stop turntable in one or other position
  - microswitches on the rim to detect turntable position

THERAC hazard

- Focused beam for X-ray therapy
  - 100x the beam current of electron therapy
  - highly dangerous to living tissue
- Previous models (Therac 6 and 20)
  - fuses and mechanical interlocks prevented high intensity beam selection unless X-ray target in place
- Therac 25 safety mechanisms replaced by software.
  - fault tree analysis arbitrarily assigned probability 10^{-11} to fault ‘computer selects wrong energy’.
- But from 1985-87, at least six accidents
  - patients directly irradiated with the high energy beam
  - three died as consequence
- Major factors: poor human computer interface, poorly written, unstructured code.

The THERAC accidents

- Marietta, Georgia, June 1985:
  - Woman’s shoulder burnt. Sued & settled out of court. Not reported to FDA, or explained
- Ontario, July 1985:
  - Woman’s hip burnt. Died of cancer. 1-bit switch error possible cause, but couldn’t reproduce the fault.
- Yakima, Washington, December 85:
  - Woman's hip burnt. Survived. ‘Could not be a malfunction’
- Tyler, Texas, March 86:
  - Man burned in neck and died. AECL denied knowledge of any hazard
- Tyler, Texas, April 86:
  - 2nd man burnt on face and died. Hospital physicist recreated fault: if parameters edited too quickly, interlock overwritten
- Yakima, Washington, January 87:
  - Man burned in chest and died. Due to different bug thought now to have also caused the Ontario accident

THERAC lessons learned

- AECL ignored safety aspects of software
  - assumed when doing risk analysis (and investigating Ontario) that hardware must be at fault
- Confused reliability with safety
  - software worked & accidents rare …
  - … so assumed it was ok
- Lack of defensive design
  - machine couldn't verify that it was working correctly
- Failure to tackle root causes
  - Ontario accident not properly explained at the time (nor was first Yakima incident ever!)

More THERAC lessons

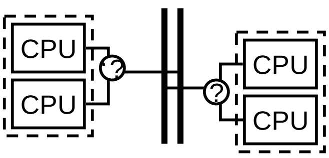
- Complacency
  - medical accelerators previously had good safety record
- Unrealistic risk assessments
  - "think of a number and double it"
- Inadequate reporting, follow-up and government oversight.
- Inadequate software engineering
  - specification an afterthought
  - complicated design
  - dangerous coding practices
  - little testing
  - careless human interface
  - careless documentation design

Failure modes & effects analysis

- FMEA is heart of NASA safety methodology
  - software not included in NASA FMEA
  - but other organisations use FMEA for software
- Look at each component's functional modes and list the potential failures in each mode.
  - Describe worst-case effect on the system
    - 1 = loss of life
    - 2 = loss of mission
    - 3 = other
  - Secondary mechanisms deal with interactions
- Alternative: Fault Tree Analysis
  - work back systematically from each identified hazard
  - identify where redundancy is, which events are critical

Redundancy

- Some systems, like Stratus & Tandem, have highly redundant hardware for 'non-stop processing'



- But then software is where things break
- 'Hot spare' inertial navigation on Ariane 5 failed first!
- Idea: multi-version programming
  - But: significantly correlated errors, and failure to understand requirements comes to dominate (Knight, Leveson 86/90)
- Also, many problems with redundancy management. For example, 737 crashes Panama/Kegworth

Example - Kegworth Crash

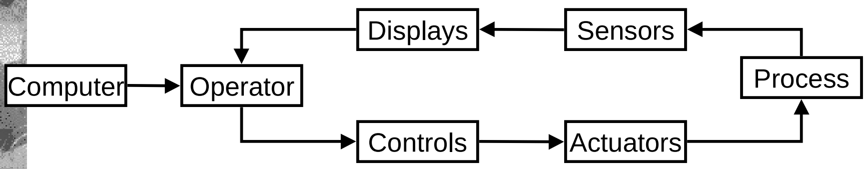
- British Midland 737-400 flight 8 January 1989
  - left Heathrow for Belfast with 8 crew + 118 passengers
  - climbing at 28,300', fan blade fractured in #1 (left) engine. Vibration, shuddering, smoke, fire
  - Crew mistakenly shut down #2 engine, cut throttle to #1 to descend to East Midlands Airport.
  - Vibration reduced, until throttle reopened on final approach
  - Crashed by M1 at Kegworth. 39 died in crash and 8 later in hospital; 74 of 79 survivors seriously injured.
- Initial assessment
  - engine vibration sensors cross-wired by accident
- Mature assessment
  - crew failed to read information from new digital instruments
- Recommendations:
  - human factors evaluations of flight systems, clear 'attention getting facility', video cameras on aircraft exterior

'Human Error' probabilities

- Extraordinary errors 10^{-5}
  - difficult to conceive how they would occur
  - stress free environment, powerful success cues
- Errors in common simple tasks 10^{-4}
  - regularly performed, minimum stress involved
- Press wrong button, read wrong display 10^{-3}
  - complex tasks, little time, some cues necessary
- Dependence on situation and memory 10^{-2}
  - unfamiliar task with little feedback and some distraction:
- Highly complex task 10^{-1}
  - considerable stress, little time to perform
- Unfamiliar and complex operations O(10^0)
  - involving creative thinking, time short, stress high

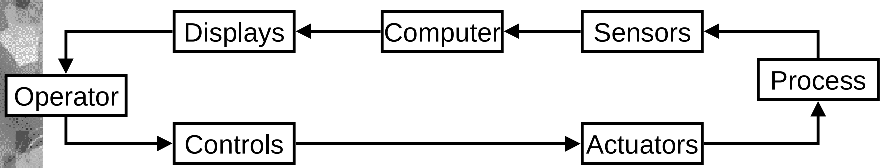
"Skill is more reliable than knowledge"

Modes of Automation



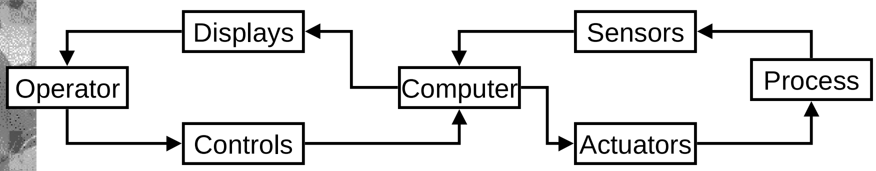
(a) Computer provides information and advice to controller, perhaps by reading sensors directly

Modes of Automation



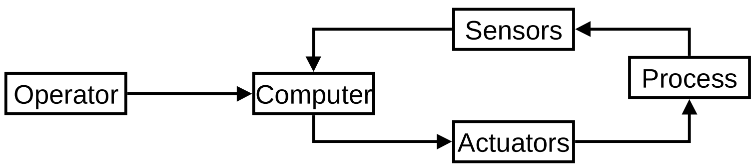
(b) Computer reads and interprets sensor data for operator

Modes of Automation



(c) Computer interprets and displays data for operator and issues commands; operator makes varying levels of decisions)

Modes of Automation



(d) Computer assumes complete control of process with operator providing advice or high-level direction

Myths of software safety

- *Computers are cheaper than analogue or electromechanical devices*
  - shuttle software costs $100,000,000 p.a. to maintain
- *Software is easy to change*
  - but hard (and expensive) to change safely
- *Computers are more reliable*
  - shuttle had 16 potentially fatal bugs since 1980 – half of them had actually flown
- *Increasing software reliability increases safety*
  - perfectly functioning software still causes accidents

More myths

- *Testing or formal verification can remove all errors*
  - exhaustive testing is usually impossible
  - proofs can have errors too
- *Software reuse increases safety*
  - using the same software in a new environment is likely to uncover more errors
- *Automation can reduce risk*
  - potential not always realised, humans still need to intervene

Tools

- We use tools when some parameter of a task exceeds our native ability
  - heavy object: raise with lever
  - tough object: cut with axe
- Software engineering tools deal with *complexity*. There are two kinds of complexity:
  - ***Incidental*** complexity dominated programming in the early days. eg. writing machine code is tedious and error prone. Solution: high level language
  - ***Intrinsic*** complexity of applications is the main problem nowadays. eg. complex system with large team. "Solution": waterfall/spiral model to structure development, project management tools, etc.
- We can aim to *eliminate* incidental complexity but must *manage* intrinsic complexity

Incidental complexity

- Greatest programmer productivity improvement: high level languages (since FORTRAN)
  - 2000 LOC/year goes much further in Java than assembler
  - code easier to understand and maintain
  - more appropriate level of abstraction
    - data structures/functions/objects not bits/registers/branches
  - structure enables typos etc to be found at compile time
  - code may be portable; at least hide machine specific detail
    - drivers etc written once, not embedded in each application
- Objections:
  - compilers have errors (but programmers make more!)
  - performance (so optimise, but only where needed)
- Performance gain (of programmers) 5-10 times

More incidental complexity fixes

- Now coding about 1/6 of total project effort
  - no similar performance gain anywhere else
- Most advances since early high level languages have focused on helping programmers to structure and manage code
  - don't use ‘goto’ (Dijkstra, 68)
  - structured programming; pascal (Wirth, 71)
- Basic idea: combining information hiding with control structures
  - facilitates stepwise refinement
  - facilitates correct abstraction

More incidental complexity fixes

- Object-oriented programming
  - Simula (Nygaard, Dahl, 60s)
  - Smalltalk (Xerox, 70s)
  - C++, Java, ...
- Basic idea: bundle code & data into *object*
  - really design philosophy, not language family
  - but successful as result of language success
- Well covered in the rest of the course.
- Don’t forget main purpose is to manage complexity! (Y2K, Ariane, Patriot, ...)

More incidental complexity fixes

- Early batch systems tedious for developers
  - coding forms, submit jobs, puzzle over output
- Time sharing: test-debug-fix-compile-test
  - test ‘scaffolding’, careful debugging plan
- Tools to support cycle:
  - snapshots, dump analysis, source debuggers
- Integrated programming environments
  - TSS, Unix, Smalltalk, Turbo Pascal …
- CASE Tools
  - Computer Aided Software Engineering
  - manage intrinsic complexity of large projects

Formal methods

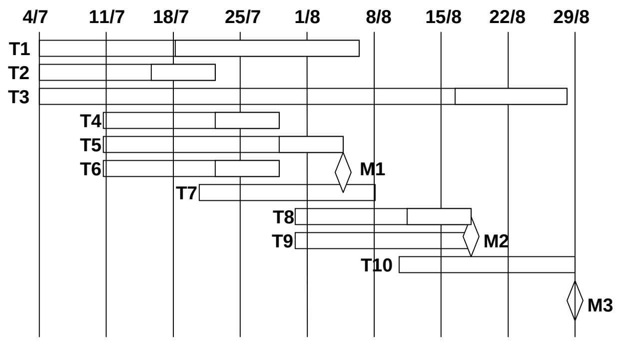
- Pioneers (e.g. Turing) talked of proving programs using mathematics
  - program verification started with Floyd (67)
  - followed up by Hoare (71) and others
- Now a wide range of techniques and tools for both software and hardware, ranging from the general to highly specialised.
  - Z, based on set theory, for specifications
  - LOTOS for checking communication protocols
  - HOL for hardware
  - BAN for cryptographic protocols
- Not infallible – but many bugs found
  - force us to be explicit and check designs in great detail
  - but proofs have mistakes too
- Considerable debate on value for money

Project management

- A manager's job is to deal with human consequences of intrinsic complexity by
  - planning, motivating, controlling
- Skills primarily interpersonal rather than technical …
  - but managers need respect of technical staff
- Growing capable managers a perpetual problem of the 'software crisis'.
  - 'managing programmers is like herding cats'
- However tools can help
  - at least with planning and controlling aspects
  - especially managing time allocated to subprojects
  - "project management" tools

Activity Charts

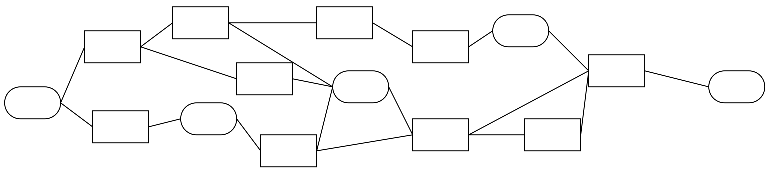
- Show a project's tasks and milestones (with allowable variation)



- Problem: relatively hard to visualise interdependencies and knock-on effects of any milestone being late.

Critical Path Analysis

- Drawing activity chart as graph with dependencies makes critical path easier to find and monitor



- PERT charts include bad / expected / good durations
- warn of trouble in time to take actions
- mechanical approach not enough
  - overestimates of duration come down steadily
  - underestimates usually covered up until near deadline!
- management heuristic
  - the project manager is never on the critical path

Documentation

- Projects have various management documents:
  - contracts - budgets - activity charts & graphs - staff schedules
- Plus various engineering documents:
  - requirements - hazard analysis - specification - test plan - code
- How do we keep all these in step?
  - Computer science tells us it's hard to keep independent files in synch
- Possible solutions
  - high tech: CASE tool
  - bureaucratic: plans and controls dept
  - convention: self documenting code

An alternative philosophy

- Some programmers are very much more productive than others - by a factor of ten or more
- ‘Chief programmer teams’, developed at IBM (1970-72) seek to capitalise on this
  - team with one chief programmer + apprentice/assistant,
  - plus toolsmith, librarian,admin assistant, etc
  - get the maximum productivity from the available talent
- Can be very effective during the implementation stage of a project
  - However, each team can only do so much
  - Complementary to (rather than opposed to) waterfall/spiral and other project management methodologies

More alternative philosophies

- ‘Egoless programming’
  - code owned by team, not by individual (Weinberg, 1971).
  - in direct opposition to the ‘chief programmer’ idea.
- ‘Xtreme Programming’ (XP)
  - small groups work together for fast development cycle iteration, early exposure to users. (Beck 199x)
- ‘Literate programming’
  - code as a work of art, designed not just for machine but for human readers / maintainers (Knuth et al)
- Objections:
  - can lead to wrong design decisions becoming entrenched, defended, propagated more passionately
  - ‘creeping elegance’ may be symptom of project out of control
- There is no silver bullet!

Configuration Management & Change Control

- One of the most critical, yet often poorly performed, software tasks - from the point of view of reliability, safety, security, ...
  - main idea is to control the process
  - test process may have multiple stages (for home written software) or be a simple compatibility check (for package upgrades)
  - someone must assess residual risk & take responsibility for live running
- Fewer changes are easier to manage
  - e.g. AT&T exchange code updated quarterly
- Need to manage:
  - backup and recovery
  - rollback
  - interim bug fixes

Testing

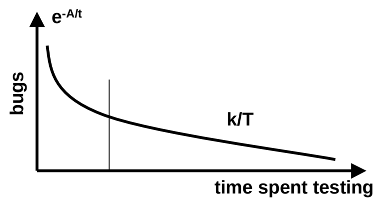
- Testing is neglected in academic studies
  - but great industrial interest - maybe half the cost
- Bill Gates: 'are we in the business of writing software, or test harnesses?'
- It takes place at a number of levels - cost per bug removed rises dramatically at later stages:
  - validation of the initial design
  - module test after coding
  - system test after integration
  - beta test 1 field trial
  - subsequent litigation
  - ...
- Common failing is to test late, because early testing wasn't designed for.
  - This is expensive. We must design for testability

Regression testing

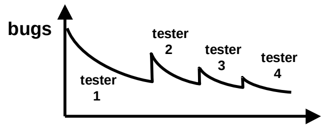
- Main software engineering advance in package software development has been in testing
  - design for testability
  - regression testing - checking that new version of software gives same answers as old version
- Use a large database of test cases, including all bugs ever found. Specific advantages:
  - customers are much more upset by failure of a familiar feature than of a new one
  - otherwise each bug fix will have a ~ 20% probability of reintroducing a problem into set of already tested behaviours
  - reliability of software is relative to a set of inputs. Best test the inputs that users actually generate!

When to stop testing

- Reliability growth model helps assess
  - mean time to failure
  - number of bugs remaining
  - economics of further testing, ...
- Software failure rate
  - drops exponentially at first
  - then decreases as K/T



- Changing testers brings new bugs to light



- to get a mttf of 10^9 hours, need 10^9 hours testing

Risk reduction vs Due diligence

- Techniques so far were about risk reduction
  - But risk reduction can be fuzzy and open-ended
  - We may know *how much* to test but not *what* to test
- Organisations are highly averse to such uncertainty:
  - prefer to avoid residual risk issues
  - cultural pressure (eg aviation, banking) to do as others do
  - legal pressures everywhere
    - negligence judged 'by the standards of the industry'
- So risk reduction gets replaced with 'due diligence'
  - following a standard checklist
  - hiring a big-name consultant
  - complying with BS xxxx or ISO yyy
- Often more expensive than doing the job properly
  - it can also lead to 'structural' disasters

CAPSA project

- Now Cambridge University Financial System
- Previous systems:
  - In-house COBOL system 1966-1993
    - Didn't support commitment accounting
  - Reimplemented using Oracle + COTS 1993
    - No change to procedures, data, operations
- First attempt to support new accounts:
  - Client-server "local" MS Access system
  - To be "synchronised" with central accounts
  - Loss of confidence after critical review
- May 1998: consultant recommends restart with "industry standard" accounting system

CAPSA project

- Detailed requirements gathering exercise
  - Input to supplier choice between Oracle vs. SAP
- Bids & decision both based on optimism
  - 'vapourware' features in future versions
  - unrecognised inadequacy of research module
  - no user trials conducted, despite promise
- Danger signals
  - High 'rate of burn' of consultancy fees
  - Faulty accounting procedures discovered
  - New management, features & schedule slashed
  - Bugs ignored, testing deferred, system went live
- "Big Bang" summer 2000: CU seizes up

CAPSA mistakes

- No phased or incremental delivery
- No managed resource control
- No analysis of risks
- No library of documentation
- No requirements traceability
- No policing of supplier quality
- No testing programme
- No configuration control

CAPSA lessons

- Classical system failure (Finkelstein)
  - More costly than anticipated
    - £10M or more, with hidden costs
  - Substantial disruption to organisation
  - Placed staff under undue pressure
  - Placed organisation under risk of failing to meet financial and legal obligations
- Danger signs in process profile
  - Long hours, high staff turnover etc
- Systems fail systemically
  - not just software, but interaction with organisational processes

Problems of large systems

- Study of 17 large & demanding systems
  - (Curtis, Krasner, Iscoe, 1988)
  - 97 interviews investigated organisational factors in project failure
- Main findings - large projects fail because
  - (1) thin spread of application domain knowledge
  - (2) fluctuating and conflicting requirements
  - (3) breakdown of communication and coordination
- These were often linked, with typical progression to disaster (1) $\rightarrow$ (2) $\rightarrow$ (3)

More large system problems

- Thin spread of application domain knowledge
  - who understands all aspects of running a telephone service/bank branch network/hospital?
  - many aspects are jealously guarded secrets
  - sometimes there is structured knowledge (eg pilots)
  - otherwise, with luck, you may find a genuine 'guru‘
  - So expect specification mistakes
- Even without mistakes, specification may change:
  - new competitors, new standards, new equipment, fashion
  - change in client: takeover, recession, refocus, …
  - new customers, e.g. overseas, with different requirements
- Success and failure both bring their own changes!

More large system problems

- How to cope with communications overhead?
  - Traditionally via hierarchy
    - information flows via managers, they get overloaded
  - Usual result - proliferation of committees
    - politicking, responsibility avoidance, blame shifting
  - Fights between 'line' and 'staff' departments
    - Management attempts to gain control may result in constriction of some interfaces, e.g. to customer
  - Managers often loath to believe bad news
    - much less pass it on
  - Informal networks vital, but disrupted by 'reorganisation‘

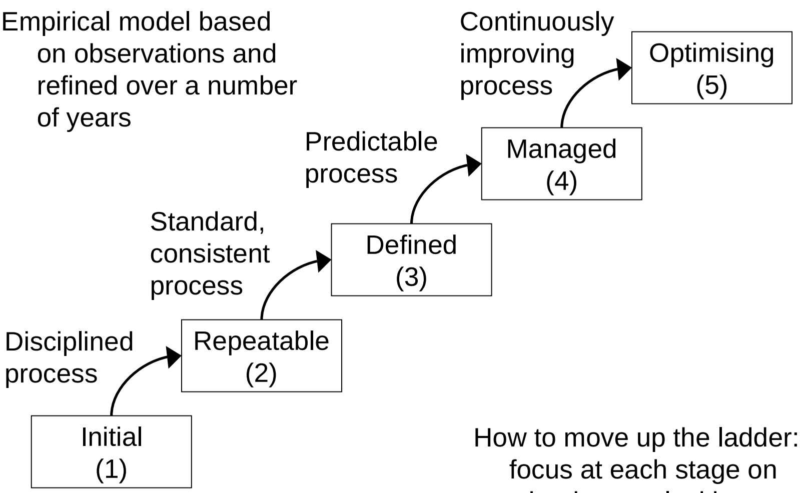
- We trained hard, but it seemed that every time we were beginning to form up into teams, we would be reorganised. I was to learn later in life that we tend to meet any new situation by reorganising, and a wonderful method it can be for creating the illusion of progress while producing confusion, inefficiency and demoralisation.
  - Caius Petronius (AD 66):

Capability Maturity Model

- By mid-80's, people had begun to realise the importance of keeping teams together
  - ability to work as a team productively grows over time
  - emphasis shift from 'product' to 'process'
- A good team itself isn't enough
  - need repeatable, manageable performance
  - not outcome dependent on individual genius or heroics
- Capability Maturity Model (CMM)
  - 'market leading' approach to this problem
  - developed at CMU with DoD funding
  - identifies five levels of increasing maturity in a software team or organisation
  - provides a guide to moving up from one level to the next

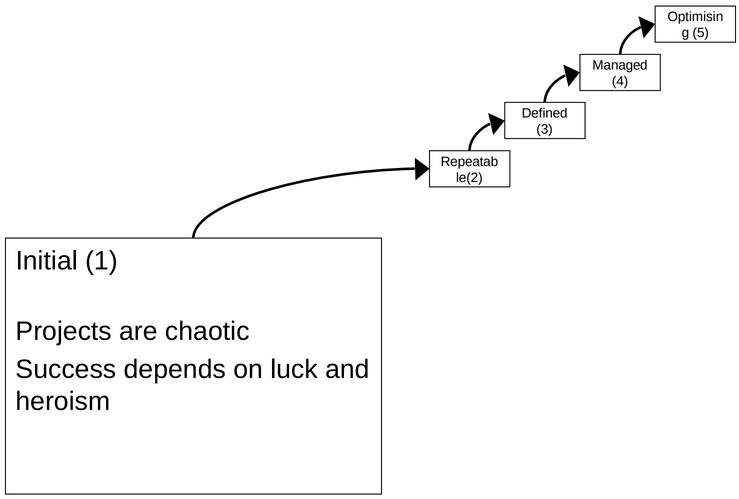
Levels of CMM

Empirical model based on observations and refined over a number of years

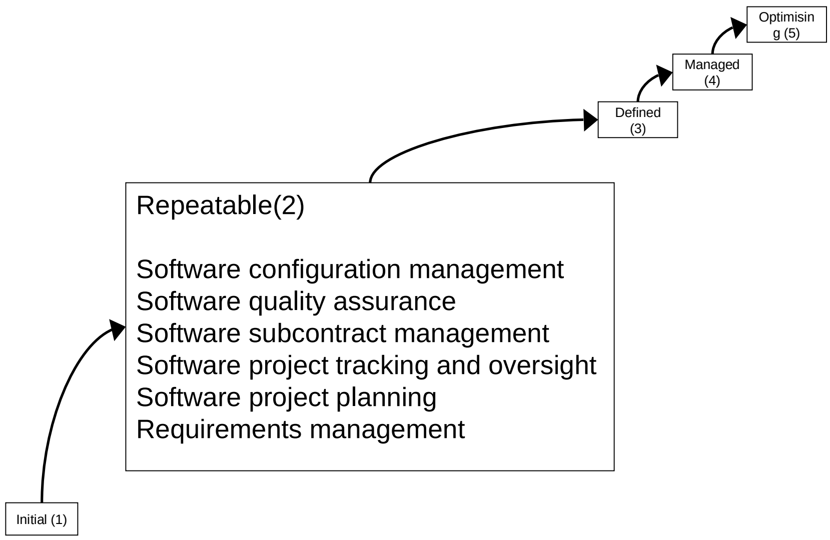


How to move up the ladder: focus at each stage on what is most lacking

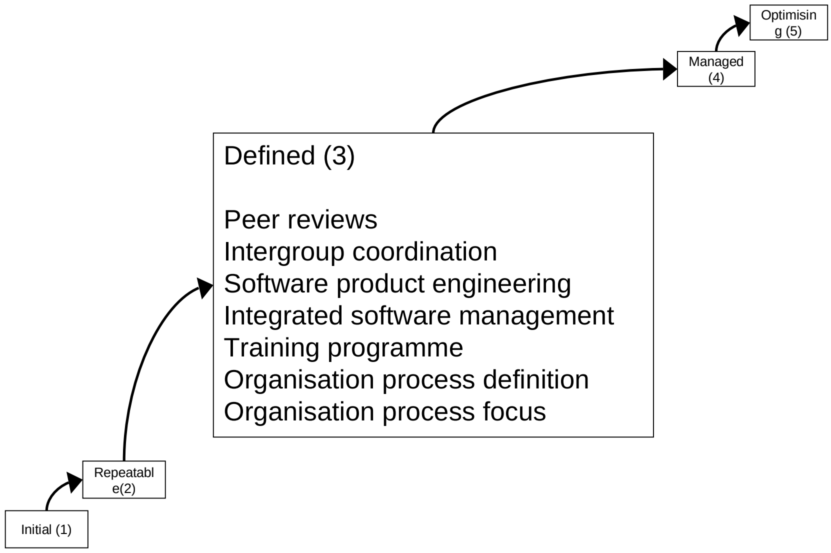
Levels of CMM



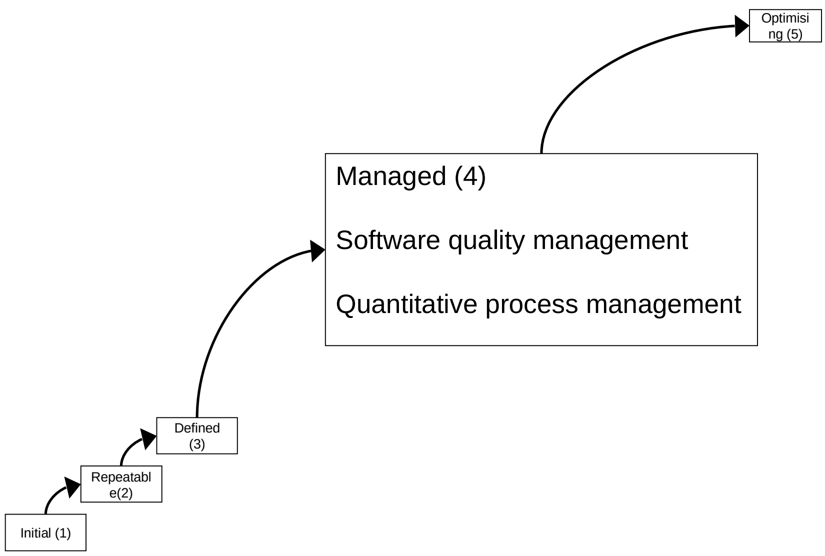
Levels of CMM



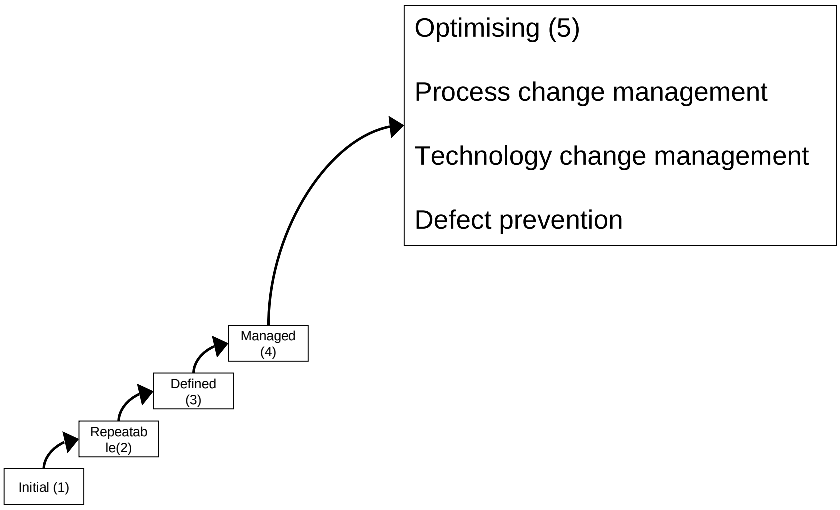
Levels of CMM



Levels of CMM



Levels of CMM



CONCLUSIONS

- Software engineering is hard
  - because it is about managing complexity
- Can reduce incidental complexity using tools
  - high level languages, design environments
  - but intrinsic complexity remains, especially given size of many modern products
- To hold it all together, engineers must:
  - understand the requirements
  - partition problem into manageable subproblems
  - use project management techniques
- Top down approach is necessary but not sufficient
  - may need to iterate the design
- The maturity of the process is important
  - Rome wasn’t build in a day; neither was Microsoft