2018

Software and Security Engineering

UNIVERSITY OF CAMBRIDGE, PART IA

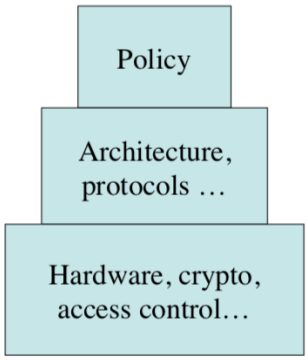
ASHWIN AHUJA

Software and Security Engineering

Security Policy and Safety Case

Security Engineering: About building systems to remain dependable in the face of malice, error and mischance. It is about the whole system not just parts. As a discipline, it focuses on the tools, processes, methods needed to design, implement and test complete systems and to adapt existing systems as their environment evolves.

Design Hierarchy: We have a set design hierarchy which considers the following questions – (1) What are we trying to do? (2) How? (3) With what?



Security vs Dependability:

- Dependability = Reliability + Security
- Reliability – random failure
- Security – malicious failure
- The two are often correlated in practise

Terminology

- **System**
  - Can be:
    - Product or component
    - Some products plus OS, comms and infrastructure
    - Above + applications
    - Above + internal staff (have the potential to create chaos)
    - Above + customers / external users
- **Subject**
  - Physical person
  - Person can also be a legal person (firm)
- **Principal**
  - Personal
  - Equipment
  - Role
  - Complex role (A deputising for B)
- **Secrecy**
  - Mechanisms limiting the number of principals who can access information
- **Privacy**
  - Control of your own secrets

    - Privacy is expressed by confidentiality of a person, which is also shown by secrecy.
- **Anonymity**
    - About restricting access to metadata
    - Has various flavours, from not being able to identify subjects, to not being able to link their actions
- **Integrity**
    - An object's integrity lies in its not having been altered since the **last altered modification**
- **Authenticity**
    - (1) Object has integrity + freshness
    - (2) You're speaking to the right principal – therefore have an authentic copy.
- **Trust**
    - Has several meanings:
        - (1) Warm fuzzy meaning
        - (2) Trusted system or component is one that can break my security policy
        - (3) Trusted system is one I can ensure
        - (4) Trusted system won't get me fired when it breaks
    - Generally, use the second meaning – NSA definition
- **Error**
    - Design flaw or deviation from an intended state
- **Failure**
    - Non-performance of the system within specified environmental conditions
- **Reliability**
    - Probability of failure within a set period of time
- **Accident**
    - Undesired, unplanned even resulting in specified kind or level of loss
- **Hazard**
    - Set of conditions on a system / their environment where failure can lead to an accident
    - '**Hazard Condition**'
- **Critical System**
    - System whose failure will lead to an accident
- **Risk**
    - Probability of an accident
    - Risk is hazard level combined with danger (probability hazard -> accident) and latency (hazard exposure + duration)
        - Micromort – 1 in a million chance of dying
- **Uncertainty**
    - Where the risk is not quantifiable
- **Safety**
    - Freedom from accidents
- **Security Policy**
    - Succinct statement of protection goals – less than a page of normal writing
- **Protection Profile**
    - Detailed statement of protection goals – dozens of pages of semi-formal languages
- **Security Target**

- Detailed statement of protection goals applied to a particular system – hundreds of pages of specification for both functionality and testing

Methodology

- **Top-Down Development:** Need to get the safety / security policy right in the early stages of the project.
- **Iterative:** Then the safety / security requirements can get ignored or confused
- In safety-critical systems world, methodologies for maintaining the safety case
- In both security and safety, the big problem is often maintaining dependability as the system and the environment evolve.

Policy

Example of a bad 'policy':

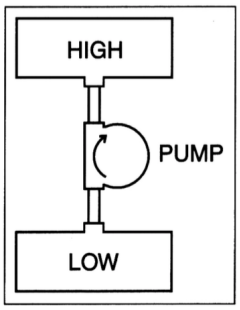
1. This policy is approved by Management.
2. All staff shall obey this security policy.
3. Data shall be available only to those with a 'need-to-know'.
4. All breaches of this policy shall be reported at once to Security.

Issues

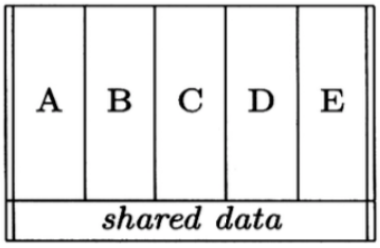
- We generally need better goals
- Who are those with need-to-know
- Breaches are undefined as a concept
- Whose duty is it to report breaches
- What if breach from the Security dpmt.

Traditional Government Approach

- Threat Model
  - Insider who is disloyal
  - Careless insiders
- Therefore, limit number of people you have to trust and make it harder for them to be trustworthy
- This leads to **Multilevel Secure Systems (MLS)**
  - Enforce standard handling rules for materials at 'Confidential', 'Secret', 'Top Secret', etc
  - Every resource has a classification, and principles have clearances.
  - We enforce using Mandatory Access Control
  - **Bell-LaPadula (1973)**
    - (1) Simple Security Policy: no read up
    - (2) *-policy: no write down
    - Therefore, information only flows up
    - With these policies, we can prove that a system that starts in a secure state will remain in one.
  - **Distributed System**
    - Sort of breaks with a distributed system
    - In order to read down, we need to send a message to the other machine with the message to send the data
    - Therefore, we are effectively writing down by sending this command
  - **Typical MLS System**
    - We use architecture to get a high assurance
    - Therefore, we change a complex emergent property of the whole system into a simple property of a testable component



  - Safety via Multilevel Integrity
    - Biba model – data may flow only down from high-integrity to low-integrity
    - It is the dual of BLP
    - The problem is still about insiders, who could break the system.
- Sometimes we want to stop lateral flows of data:



Architecture Matters

- Lots of legacy protocols trust all network nodes
  - DNP3 in Control Systems
  - CAN bus in cars
- This led to a Chrysler recall
- IP addresses and any bad node leads to trouble
- Therefore, we can have separate subnets and have capable firewalls

Safety Policies

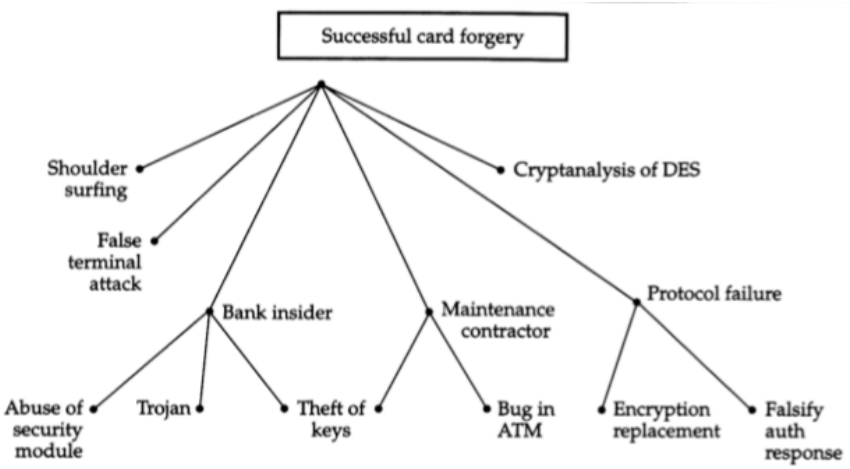
Industries have their own standards, cultures, often with architectural assumptions embedded in component design – no one in one industry works with another. Generally, safety standards tend to evolve. There are two ways to design them:

Bottom Up – Failure Modes and Effects Analysis (FMEA)

- Look at every component and list the failure modes
- Work out what to do about each of them (can also leave alone)
  - Cut probability with overdesign
  - Redundancy
- Then, use secondary mechanisms to deal with interactions

Top Down – Fault Tree Analysis

Work back from the bad outcome that we must avoid, in order to identify critical components.



Bookkeeping

Bulla (c. 3300 BC): Used to keep stock as pieces of clay – different shapes and sizes for different commodities. It eventually led to writing.

Double-Entry Bookkeeping (c. 1100 AD): Each entry in one ledger is matched by opposite entry in another – sales ledger and accounts ledger. Therefore, the bookkeepers must collude to commit fraud.

Separation of Duties in Practise:

- Serial – orders point from person to person through a chain.
- Parallel
    - Multiple simultaneous (each required), paths through the system

Decoupling Policy, Mechanism

- Role-Based Access Control (RBAC) adds an extra indirection layer – each person is associated with a role and each role has a certain number of actions the role has permissions to be able to do.
- However, we still need to device a security policy – SELinux offers MLS with RBAC
    - iPhones have something similar
    - Red Hat uses it to separate services: a web server compromise doesn't automatically get DNS.

Defence in Depth

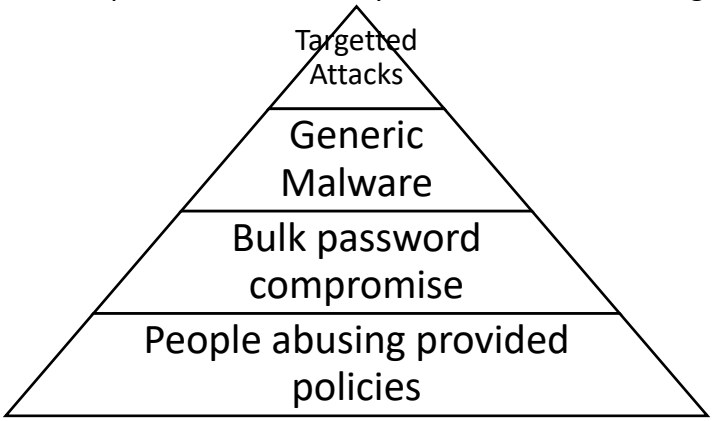
- Swiss cheese model
    - Things fail when holes in the defence layers line up
- Therefore, we ensure that human factors, software and procedures complement each other

Psychology

It is generally important to consider user error – still important to consider what would happen if there is a user error. For example, most car crashes are user error, however, we still build cars with crumple zones.

Hierarchy of Harms

Each step down the hierarchy the number of victims goes up an order or magnitude.



- Example of targeted attacks are the Gmail Spearfishes in the 2016 election
- Car crashes can be considered as an example of abuse of mechanisms (policies)

- Privacy Law
  - We need the consent from the people whose data it is – what does consenting mean
  - Or anonymise the data
  - Both of these are getting harder as the systems get more complex
  - **Example**
  - Facebook privacy policy is found illegal by German court even before recent scandal.
    - No real choice
- Medical device safety
  - Each device has a different interface which has the ability to create a lot of problems
  - Usability problems with medical devices kill about as many people as cars do
  - INFUSION PUMPS
  - Nurses are generally blamed, not vendors
- Abuse of standard mechanisms
  - Generally, by far the most used issue – effects the vast majority of those attacked
  - E.g. crook runs website offering a flat to let so you send off some money.
- **Bulk Password Compromise**
  - The main method is SQL injection – Linkedin June 2012 passwords were stored unsalted.
    - 6.5m passwords posted on a Russian Forum
  - The passwords were reused and exploited there
    - Need to deal with reuse of passwords
- **Phishing and Social Engineering**

- Effectively about getting people to fill in / give details by calling them or emailing them with an online form to be filled in etc.
- Generic phishing has been around since 2005
- 30% yield on well-crafted lures
- 50% yield on personalised attacks
  - Using personal data

Cognitive Factors

Many errors arise from our highly adaptive mental processes:

1. Deal with novel problems in a conscious way
2. Encountered problems dealt with using evolving rules
3. Rules give way to skill over time

These abilities to automate routine actions leads to absent-minded slip or following a wrong rule. There are also systematic limits to rationality in problem solving – biases and heuristics

Risk Misperception

People offered £10 or a 50% chance of £20 usually preferred the former; if offered a loss of £10 or a 50% chance of loss of £20, they prefer the former. Kahneman and Tversky's prospect theory explains such risk perceptions systematically. This risk misperception is exploited by cybercriminals (to remain inconspicuous) and terrorists (to be particularly obvious).

Also, the risk perception is based on the framing of the problem – **Asian Disease Problem:**

- Option 1: A: "200 lives saved", B: "600 lives saved, p=1/3, with p=2/3 0 saved"
  - 72% choose A over B
- Option 2: C: 400 die, D: "p:1/3 no-one die, p=2/3 600 die"
  - A = C
  - B = D
  - Now, 78% choose D over C

Therefore, marketers talk about discount or saving – people facing losses take more risks.

Authority

- Stanley Milgram showed that over 60% of all subjects would torture a student if told to do so by a powerful figure.
  - We investigate the extent to which ordinary people do bad things – Nazi Model
- **Herd Matters**
  - Asch showed most people could deny obvious facts to please others
  - Which is the shorter line task
- Reciprocation is built-in
  - Therefore, we can give a gift, or appear to be in the mark's in-group

Fraud Psychology

Therefore, a hacker can do the following:

1. Appeal to mark's kindness
2. Appeal to the mark's dishonesty
   a. Nigerian Princes
3. Distract them so they act automatically
   a. Ballpoint pen – let the client catch the pen when it is falling
4. Arouse them so they act viscerally

Users' Mental Models: The kind of security advice a user is likely to follow depends on their main mental model. We explore how a user sees the problem (folk beliefs):

- Threats seen as viruses which could be mischievous or crime tools
- Hackers may be seen as graffiti artists or burglars or targeting big fish
- Simply as bad neighbourhoods online

Affordances: Idea that the actions made natural really matter – most people go with the flow (therefore you can nudge them). There is a paper called 'Why Johnny couldn't encrypt' which describes how even an engineering student couldn't understand what encryption was and how to use it.

Compliance Budget (UCL – Adam Beautement): People will only spend a certain amount of time obeying rules, so choose the rules that matter. The violations also matter – they're often an easier way of working and are sometimes necessary. The right way of working should be easiest.

Differences between People

The ability to perform certain tasks can vary widely across subgroups of the population. Therefore, need to be careful about everyone in every group can use things.

Error: What psychology underlies errors?

- Slips and lapses
    - Forgetting plans, intentions, strong habit intrusions
    - Misidentifying objects, signals (Bayesian)
    - Retrieval failures – tip-of-tongue
    - Premature exits from action sequences
- Rule-based mistakes – applying wrong procedure
- Knowledge-based mistakes – heuristics and biases
- How to fix?
    - Training and practise - skill is more reliable than knowledge
    - Error rates (motor industry 10^x)
        - Inexplicable error – (-5)
        - Regularly performed simple tasks – (-4)
        - Complex tasks, little time – (-3)
        - Unfamiliar task dependent on situation – (-2)
        - Highly complex task, lots of stress – (-1)
        - Creative thinking, unfamiliar complex operations, time short and stress is high - ~1

Passwords

Passwords are generally the cheapest and most used way of authenticating someone, but there are a few issues – three in particular:

1. Will users enter passwords correctly?
2. Will they remember them – or choose to weak ones – or write them down?
3. Can they be tricked into revealing them?

The advice is often like 'choose something you can't remember and don't write it down'

Can you train users?

- First-year NatScis experiment
  - Green group: use a memorable phrase
  - Yellow group: choose 8 chars at random
  - Control group
- Expected strengths Y > G > C; got Y=G > C
- Expected resets Y > G > C; got Y=G=C

Password Issues

- Getting people to change their passwords one per month is entirely useless
- You should limit password guessing - 3 guesses for PINs
  - But if the typical person has 5 cards with the same pin, you need 10 wallets on average before you get lucky.
- Salting
  - Bad people sometimes get the password file anyway
  - Don't store $\{0\}_P$, but $[Np, \{Np\}_p]$
- We also slow attacks further by multiple encryption
- We then add breach reporting laws – so must find out if our password is leaked
- We can also externalise the problem using the Oauth protocol
- **Externalities**
  - One firm's action has side-effects for others
  - Password sharing is a conspicuous example – we have to enter credentials everywhere
  - Everyone wants recovery questions
  - Many firms train customers in unsafe behaviour from clicking on external links to entering payment data in frames
  - **Matt Honan Hack**
    - Gmail password reset sends message to backup email and prints part of it (Apple @me.com)
    - Call amazon with email and billing address, they let you add a credit card
    - Call amazon again, can reset account with email, billing address, credit card number
    - Get the last 4 numbers of all credit cards
    - Use this to reset apple account
    - Then reset google account
    - Reset twitter etc
- **Incremental Guessing**
  - Of Alexa top 500 websites, 26 use PAN (Primary Account Number) + expiration date
  - 37 use PAN + postcode
  - 291 use PAN + exp date + CVV
  - So: iterated guessing with a botnet works – some paper receipts have the PAN and the expiry date.

Protocols

The idea of a protocol is ensuring that there is trust movement from where it exists to wherever it is needed. They are a second intellectual core of security engineering – where cryptography and system mechanisms (access control) meet.

Real World Protocol

Example: Ordering wine in a restaurant:

- Sommelier presents the wine list to the host
- Host chooses the wine; sommelier fetches it
- Host samples win; then served to guests.

This offers:

1. Confidentiality
    a. Guests don't know price
    b. But, guests can clearly just come back the next day and find this out.
2. Integrity
    a. Can't substitute cheaper wine
3. Non-repudiation
    a. Host can't falsely complain

Car unlocking protocols

The principals are the Engine Controller € and the Car Key Transponder (T), with the key being KT.

1. Static: (T → E: KT)
    a. Send key in plaintext
    b. This is solvable using a replay attack
    c. Also, man in the middle
2. Non-Interactive: (T→E: T, $\{T, N\}_{KT}$)
    a. N being a nonce
    b. This is breakable using some kind of brute force attack based on the nonce
    c. Observe enough and you can work it out
    d. Or working backwards knowing the way the nonce was developed using some kind of rainbow table?
3. Interactive
    a. E → T: N
    b. T → E: $\{T, N\}_{KT}$
    c. This is vulnerable to a man-in-the-middle attack
    d. Someone in the middle sending an N to a transponder then replay that to the engine controller
    e. If it is a sequential nonce then replay attack very possible as well

Identify Friend or Foe (IFF) (Some sort of reflection attack)

Basic idea is that A challenges B to see if they are on the same side:

- A → B: N
- B → A: $\{N\}_K$

Easily broken by reflecting the challenge:

- A → B: N
- B → A: N
- A → B: $\{N\}_K$
- B → A: $\{N\}_K$

Key Management Protocols

- Method used by Xerox Alto
- Assume Alice and Bob each share a key with Sam, and want to communicate
    - Alice calls Sam and asks for a key for Bob
    - Sam sends Alice a key encrypted in a blob only she can read, and the same key also encrypted in another blob only Bob can read.
    - Alice calls Bob and sends him the second blob
    - This can be kept up to date – timestamp and time-to-live.

Kerberos

- Uses tickets based on encryption with timestamps to manage authentication in distributed systems
    - A → S: A, B
    - S → A: {Ts (timestamp), L (ttl, KAB (encryption key), B, $\{Ts, L, KAB, A\}_{KBS}$ }$_{KAS}$
    - A → B: $\{Ts, L, KAB, A\}_{KBS}$, $\{A, Ta\}_{KAB}$
    - B → A: $\{Ta + 1\}_{KAB}$
    - Where S is the ticket-granting server giving access to the resource B

Europay-MasterCard-Visa (EMV)

- C → M: $sig_B\{C, card\_data\}$
- M → C: N, date, Amt, PIN
- C → M: $\{N, date, amt, trans\_data\}_{KCB}$
- M → B: $\{\{N, date, amt, trans\_data\}_{KCB}, trans\_data\}_{KCB}$
- B → M → C : $\{OK\}_{KCB}$
- Attacks
    - False terminal
    - Replace terminal insides with own electronics
    - Use the customers to do a man-in-the-middle attack in real time on a remote terminal in a merchant selling expensive goods.
    - **RELAY ATTACK**
- Attacks in real world
    - Relay attack is hard to scale
    - Initially – magstripe fallback fraud
        - Chip is broken
        - So swiping works
    - PEDs tampered at Shell garages by service engineers, BP garages as well
    - **No-PIN attack**
        - C → M: $sig_B\{C, card\_data\}$
        - M → C': N, date, Amt, PIN
        - C' → C: N, date, Amt

- C → M: $\{N, date, Amt, trans\_data\}_{KCB}$
- M → B: $\{\{N, date, Amt, trans\_data\{_{KCB}, trans\_data\}_{KMB}$
- B → M: $\{OK\}_{KCB}$
- Fix in theory
  - Compare card data with terminal data at terminal, acquirer, or issuer
- Practise
  - Has to be issuer – otherwise incentives non-existent.
  - Fix issued by Barclays in 2016
- Real problem
  - EMV spec now far too complex

Preplay Attack
In EMV, the terminal sends a random number N to the card along with the date d and the amount amt. The card authenticates N, d, X using the key it shares with the bank. But, it you can predict N for date d, then can precompute authenticator for Amt, d.

Public Key Cryptography

Assumption that you can encrypt with a public key and impossible to decrypt unless you have the private key.

- **Naïve Electronic Implementation**
  - A → B: M^{rA}
  - B → A: M^{rArB}
  - A → B: M^{rB}
    - But, encoding messages as group elements can be tiresome
- **Diffie Helman**
  - A → B: g^{rA}
  - B → A: g^{rB}
  - A → B: $\{M\}g^{rArB}$
  - This has strong cryptography but there is no assurance of who you are talking to at the other side.
- **Needham-Schroeder**
  - A → B: $\{NA, A\}_{KB}$
  - B → A: $\{NA, NB\}_{KA}$
  - A → B: $\{NB\}_{KB}$
  - Then effectively use NA $\oplus$ NB as a key
  - Attack: Alice thinks she is talking to Charlie and Bob thinks he is talking to Alice

$$
\begin{aligned}
&A \rightarrow C: \{NA, A\}_{KC} \\
&C \rightarrow B: \quad \{NA, A\}_{KB} \\
&B \rightarrow C: \quad \{NA, NB\}_{KA} \\
&C \rightarrow A: \quad \{NA, NB\}_{KA} \\
&A \rightarrow C: \{NB\}_{KC} \\
&C \rightarrow B: \quad \{NB\}_{KB}
\end{aligned}
$$

- This is fixed by explicitness – putting all names in all messages

- **Public Key Certification**
  - One way of linking public keys to principals is to physically install them on machines
  - Or trust on first use – set up keys and verify manually that you're speaking to the right principals.
  - Another is certificate use. Signing authorities sign public keys
    - CA = sig_S {T_S, L (ttl is normally approximately 2 years), A, K_A, V_A }
    - This is the basis of SSL / TLS
- **TLS (Transport Layer Security)**
  - Customer C calls server S

    $C \rightarrow S: C, C\#, NC$

    $S \rightarrow C: S, S\#, NS, CS$

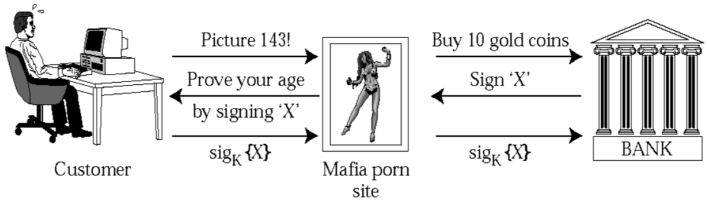
    $C \rightarrow S: \{K0\}_S$

    $C \rightarrow S:$ crypto hash of $K0, NC, NS,$ etc

    $S \rightarrow C:$ crypto hash of $K0, NS, NC,$ etc
    - CS is the certificate
    - K0 premaster served
  - This has been proved to be secure – Larry Paulson 1999
  - **What goes wrong?**
    - Real implementations break about annually
    - Attacks: (1) send bad packets and observe error messages, or (2) measure the time it takes to encrypt, or (3) scavenge memory.
    - Writing crypto code is hard ((4) compiler removes defensive code)
    - (5) Governments demand weak ciphers or attack or coerce certification authority
      - Turkish government certificate
      - Iran and the Netherlands Certificate Authority

Signing Hashes of Payment Messages

- C → M: order
- M → C: X [= hash(order, amount, date, …)]
- C → M: sig_K {X}
- This can be broken using a chosen protocol attack:



Entomology

Types of Bug:

1. **Bugs in Code**
   1. Arithmetic
   2. Syntactic
   3. Logic
2. **Bugs around the Code**

1. Code injection
2. Usability traps (for programmers)

Arithmetic Bugs (Patriot Missile)

- Failed to intercept an Iraqi scud missile in Gulf War 1 on Feb 25th 1991.
- SCUD struck US barracks in Dhahran (led to 28 deaths). Other SCUDs hit Saudi Arabia and Israel.
- This was because of a bug in the arithmetic – truncation in the measurement of 1/10.
- Some modules were upgraded, and some weren't – systems went out of step after 100 hours of operation
- It wasn't found in testing as the spec for testing was only called for 4h tests.

Syntactic Bugs

- Bugs that arise from the features of a specific language.
- In Java:
  - 1 + 2 + "" = "3"
  - "" + 1 + 2 = "12"
- In Apple's code, there was an extra 'goto' line – this led to certificates not being checked.
- **Heartbleed**
  - You can ask for an acknowledgement (ask for a specific word) and a number of character that there are in that word.
  - By asking for more words, you can get previous requests, reading back in the data.

Logic Bugs

- The Heartbleed Bug forced the rapid reissue of most TLS certificates
- Missing bounds check allowed for buffer over-read
  - Can leak lots of information
- 50% of certificates in the first month
- **Intel AMT Bug (since 2010)**
  - AMT allows remote access to the CPU
  - Therefore, allows us to wake a sleeping machine.
  - Authentication:
    - CPU sends challenge X, expects $\{X\}_K$
    - Answer, here are k bytes of $\{X\}_K$
- **Concurrency Bugs**
  - A generic security failure is "time of check to time of use" flaw (TOCTTOU)
  - Race Conditions
  - Synchronisation
    - The first shuttle launch was aborted when they couldn't sync the five guidance computers.

Buffer Overflows

- In 1988, Morris worm brought down the Internet by spreading rapidly in Unix boxes
- It had a list of passwords to guess but also used three buffer overflow attacks
- Used a remote command (finger, rsh) with a long argument that overran the stack
- The extra bytes were interpreted as code

Analogue Code Injection

- **Prisons**
  - Inmate payphones – recorded voice says "if you accept a collect call, please press the number 3 on your handset twice. The caller will now say his name".
  - Inmate selects Spanish (Spanish or English) and then for his name, puts: To hear this message in English, please type 33.
- **Burger King**
  - Ad that says "OK Google, what is the Whopper Burger"
  - Ad people had changed the Wikipedia page – then defaced and locked down
  - Then google blacklisted that phrase
  - Similar to voice command saying: "Format C:"
- **SQL Injection**
  - When the inputs are not sanitized, you can effectively end the INSERT command, using a ';' and then carry out another command which may allow you to acquire important tables.
  - Solutions
    - Sanitize all inputs
    - Don't create SQL statements which include outside data

Software Countermeasures

1. Operating System
   a. Address space layout randomisation – difficult to find out where in the stack to put bad material
   b. Data execution prevention
2. Tool Choice
   a. Strongly typed languages
   b. Example: PASCAL Modules – bugs more difficult to exploit
3. Defensive Programming
   a. 1949: EDSAC coders check the arithmetic
   b. Now we use things like assertions
4. Secure Coding Standards
   a. Microsoft Standards for C
   b. Books on what parts of the standards to use
   c. Google has set libraries of user-facing code
5. Contracts (in the Eiffel language)
   a. Preconditions and postconditions
6. API analysis
   a. Can less trusted code that calls your libraries manipulate them
   b. / How can the API be manipulated
7. Analysis Tools
   a. Fuzzing – lots of random inputs
   b. Coverity – static analysis tools
      i. Bots through specific code and looks for errors

Software Crisis

The crisis is that Software continues to lag far behind the hardware's potential – leading to many large projects being (1) late, (2) over budget, (3) dysfunctional, (4) abandoned.

Examples are LAS (London Ambulance Service), CAPSA (University Accounting Service), NPfIT (NHS program from IT), DWP (Department for Work and Pensions), Addenbrookes. Some failures cost lives (Therac 25), or billions (Ariane 5, NPfIT). Some expensive scares (Y2K, Pentium) and some combine the above (LAS).

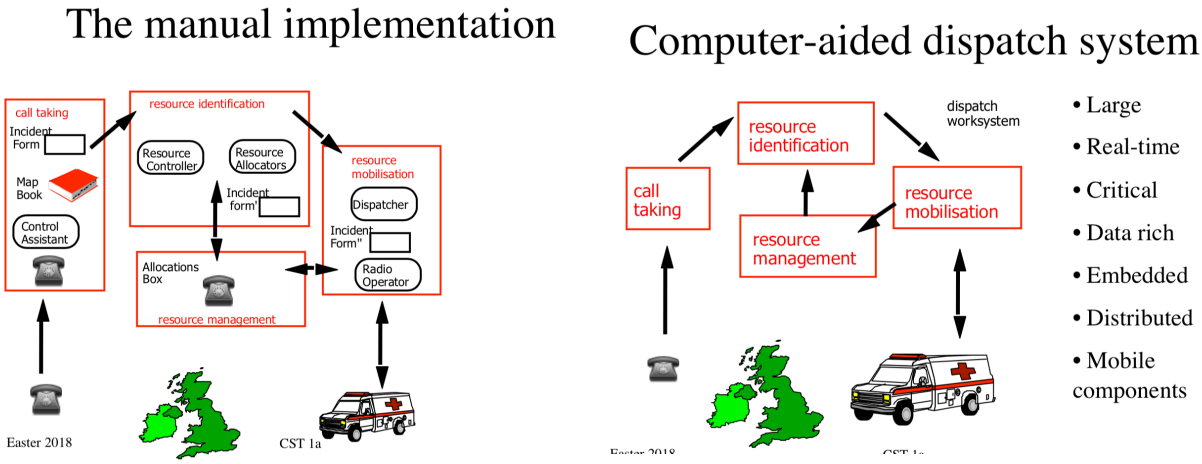
London Ambulance Service

- It is widely cited as an example of project failure as it has been thoroughly documented (and pattern frequently repeated).
- The attempt to automate ambulance dispatch in 1992 failed conspicuously with London being left without service for a day
    - Led to 20 deaths
    - CEO being sacked
    - Public outrage

Original Dispatch System: 999 calls written on paper ticket and map reference looked up. They put it on a conveyor to the central point. A controller then deduplicates the tickets (multiple people call for the same things) and passes to three divisions – NW / NE / S. Division Controllers identify the vehicle and puts a note in its activation box. The ticket is passed to the radio controller. This takes 3 minutes, 200 staff of 2700. Some errors and queues (especially radio controller), and call-backs ("Where is my ambulance?") are tiresome.

Project Context:

- Attempt to automate in 1980s failed – the system failed the load test
- Industrial relations were poor (strikes) – pressure to cut the costs
- There was also lots of public concern over service quality
- SW Thames RHA decided on fully automated system – responder would email the ambulance. A consultancy study said this might cost £1.9mn and take 19 months – **provided packaged solution found**. AVLS (automated vehicle location system) would be extra.



Tender Process:

- £1.5mn system stuck – idea of AVLS added and proviso of packaged solution was forgotten

- 35 firms looked at tender – 19 proposed and most said the timescale was unrealistic, with only partial automation possible by Feb 1992 (they had asked for Jan 1992 – though tender was only in Feb 1991)
- Tender awarded to consortium of Systems Options Ltd, Apricot and Datatrak for £937,463 - £700K less than next lowest bidder.

First Phase:

- Design work 'done' in 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 had already minuted a 6-month timescale for 18-month project, lack of methodology, no full-time user, and software company's reliance on cosy assurance on subcontractors.

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 practises'
- But management decided to go live 26/10/1992
    - CEO said 'No evidence to suggest that the full system software … will not prove reliable'
    - Independent review called for volume testing, implementation strategy, change control – ignored
- No backup on changeover day
- **Changeover Day**
    - Cascade Failure – Vicious Failure
        - (1) system progressively lost track of vehicles
        - (2) exception messages went off screen
        - (3) incidents held as allocators searched for vehicles
        - (4) call-backs from patients increased - causing congestion
        - (5) date delays -> voice congestion -> crew frustration -> pressing wrong buttons and taking wrong vehicles -> many or no vehicles sent any incident.
        - (6) slowdown and congestion leading to collapse
    - Switch back to semi-manual on 26th and fully manual on Nov 2nd

What went wrong?

1. LAS ignored consultancy advice
2. Procurers insufficiently qualified and experienced
3. No systems view
4. Specification was inflexible but incomplete – the technology was imposed on the staff / drawn up without adequate consultation with staff.
5. Attempt to change organisation through technical system
6. Ignored established work practises and staff skills
7. **Project**

   a. Management confusion
   b. Poor change control – no independent QA, suppliers misled
   c. Inadequate software development tools
   d. Poor interface for ambulance crews
      i. Dark spots
   e. Poor control room interface
8. Go-Live
   a. System went live with known faults
      i. Slow response times
      ii. Workstation lockup
      iii. Loss of voice comms
   b. Software not properly tested with real loads
   c. Inadequate staff training
   d. No back up system

NHS National Programme for IT

Like the LAS, this was an attempt to centralise power and change the working practise. There was an earlier failed attempt in the 1990s. There was a meeting in Feb 2002 with Blair - £5bn was promised. Contracts given as follows: five LSPs plus national contracts: £12bn. However, most systems were years late / didn't work and the NPfIT was abolished by coalition government.

Universal Credit

Unify hundreds of welfare benefits and mitigate poverty trap by tapered withdrawal as claimants start to earn - this was meant to go live in October 2013. However, there were a large number of problems – big systems tend to take 7 years not 3 yet somehow, they hoped that 'agile' development would fix this. Also, this depended on data from the HMRC which depended on firms.

Smart Meters

The idea of smart meters is to market prices and get peak demand shaving. The EU Electricity Directive in 2009 states that 80% by 2020 for smart meters. In 2009, Labour created a £10bn centralised project to save the planet and help fix supply crunch in 2017. In March 2010, experts said couldn't change 47mn meters in 6 years – therefore excluded it in the spec. The coalition government tried again and failed.

Managing Complexity

Software engineering is about managing complexity at a number of levels: at a micro level, bugs arise in protocols – interactions grow at $O(n^2)$ or even $O(2^n)$. Especially with complex socio-technical systems, we can't predict reactions to new functionality.

Mostly failures through wrong, changing or contested requirements.

Historical Information:

- C1500 BC Project Failures
  - Collapse of a building (the main large projects) was considered the failure of the project

- 19th Century View
  - Ensure all the designs are correct – otherwise the thing won't work
- Complexity – 1870
  - Bank of England
- Complexity – 1876
  - Dun, Barlow & Co
    - Effectively one of the first bank – they offered lots of credit to people
- Complexity – 1906
  - Sears, Roebuck
    - Continental-scale mail order meant specialisation
    - Big departments for single bookkeeping functions
    - This was the beginning of automation
- Complexity – 1940
  - First National Bank of Chicago
    - Everything was done on paper – this makes things very complicated
    - Easy to lose track of things
- 1960s – The Software Crisis
  - In 1960s, large powerful mainframes made even more complex systems were possible
  - People started asking why the project overruns and failures were so much more common for software than for mechanical engineering, shipbuilding.

What makes Software different from traditional Engineering projects

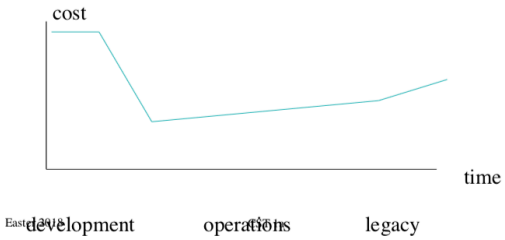
- It is very complex and error-prone
  - Lots of interlocking and moving parts
- Large systems become qualitatively more complex
- Tractability of software means that customers demand flexibility and frequent changes
- Systems become more complex to use over time – features accumulate, and interactions have odd effects.
- The structure can be hard to visualise or model
- Debugging and testing piles up at the end.

Software Lifecycle

1950s Case of company that develops and maintains software for itself:

- Initial development cost – 10%
- Continuing maintenance cost – 90%

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



Cost of code

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

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

- **Boehm, 1981 (empirical studies after Brooks)**
  - Cost-optimum schedule time to first shipment = 2.5 dev-months
  - With more time, the cost rises slowly
  - With less, it rises shapely
  - Very few projects success with less than ¾ of this 2.5 dev-months.
  - This has been supported by lots of other studies

First-Generation Lessons

- 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.
- Also, worth putting effort into getting specification – more than pays for itself in terms of time saved.

Mythical Man Month: 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 months' work.
- If training someone takes a month, we must add 6 developers
- But the work done by 3 developers in 3 months, can't be done by 9 developers in one month; interaction costs $O(n^2)$
- **Brooks' Law:** Adding manpower to a late project makes it later
  - **Brooks also debunked interchangeability**

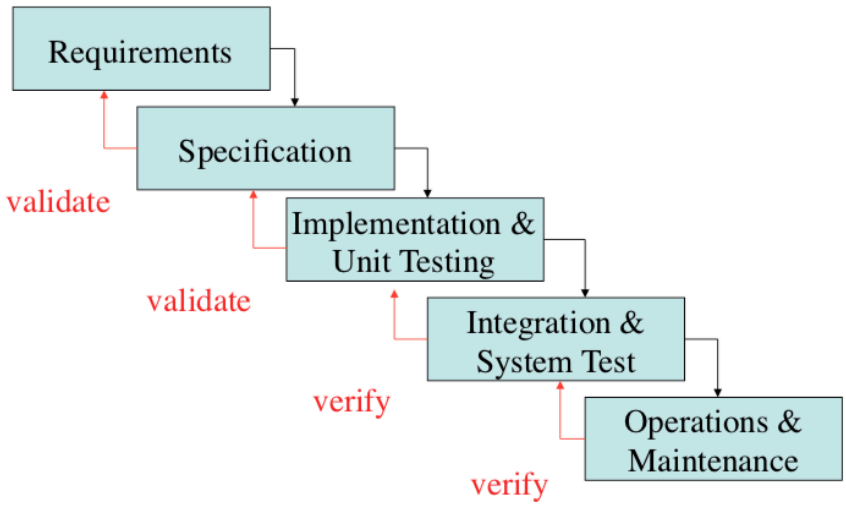
Tar Pits

You can pull any one of your legs out of the tar, but getting the entire body out is challenging. Individual software problems are all soluble, but all of them being solved together is much more challenging.

Structured Design

Only way to build large complex programs is to divide and conquer. A number of technologies have been developed to solve the solving of problems in parts and then recombining the solution – SSDM, Jackson, Yourdon, UML.

Waterfall Model: Requirements are written in the user's language, specifications in system language. The number of steps is a minimum – can be more – eg. System spec, functional spec, programming spec. The philosophy is a progressive refinement of what the user wants. It was created by the US Air Force and became promoted by governments around the world as a method of maximising how much progress is made. Feedback can only occur with the next step up – no more. Otherwise, this renders the top-down structure as pointless.



People often consider there being a useful feedback loop from the end 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. **Waterfall model is only used for specific development phases – adding a feature. However, can sometimes occur for whole system.**

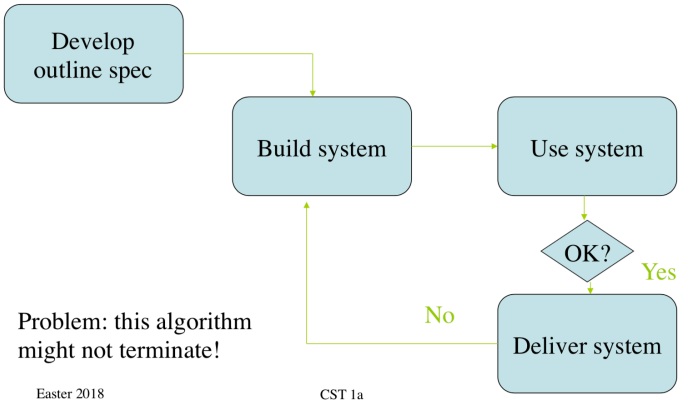
+:

1. Gets user to better define their goals – conducive to good design practise
    a. Static not a dynamic target
2. It allows the developer to be able to charge for changes to the requirements – especially in overcharging the public sector
3. Works well with management and technical tools
4. Whenever viable, generally best approach
    a. Really critical aspect is whether you can define the requirements in detail in advance.

-:

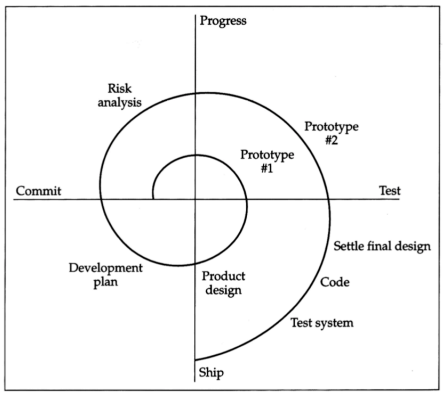
1. Iteration can be critical in the development process
2. Requirement not yet understood by the developers
3. Or not even by the customer
4. Changes in technology
    a. Moore's Law
5. Changes in environment
6. Attainable quality improvement imperceptible over system lifecycle.

Iterative Development



Spiral Model

- Set number of iterations; eg. Engineering prototype, pre-production prototype, then product.
- Each iteration is done top-down
- It is driven by risk management
    - Put energy into prototyping parts which aren't yet understood



Evolutionary Model

Products today are very complex – MS tried to rewrite Word from scratch twice and failed. The big change that made code evolution possible was the arrival of automatic regression testing which allowed (in addition to unit tests) the ability to test the entire product overnight. The development cycle is (1) add changes, (2) check them in, (3) test them.

Components:

1. Version Control – git
2. Code review – gerrit

3. Automated build – make
4. Continuous integration - Jenkins

Critical Systems

Many systems must avoid a certain class of failures with very high assurance:

1. Safety Critical Systems
    a. Failure could cause death, injury or property damage
2. Security Critical Systems
    a. Failure could allow leakage of confidential data, fraud.
3. Real-time Systems
    a. Timing is important – safety or security issues

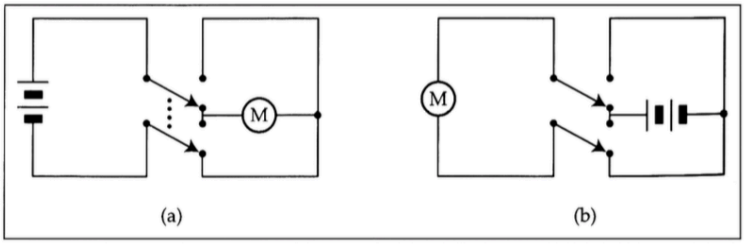
Critical computer systems have lots in common with mechanical systems – therefore useful to consider issues with mechanical and electrical failures:

Multiple Systems Failure: Many safety-critical systems are also real-time systems used in monitoring or control. Therefore, exception handling is often tricky (and it would be great to have no core dumps ever). But criticality of timing makes many simple verification techniques inadequate – testing can be very hard.

Emergent Properties:

- In general, safety, security and real-time performance are system property – deal with them holistically.
- A very common error is not getting the scope right
    - For example, the interface of medical devices – blood infusion systems
    - Generally, not considering human factors such as usability and training.

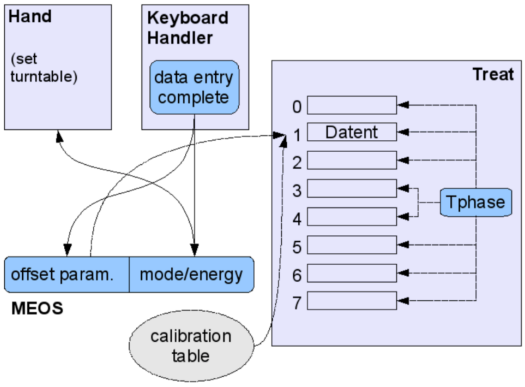
1. **Tacoma Narrows**
    a. Collapse of a bridge
    b. High speed wind produced aeroelastic flutter which matched the bridge's natural frequency – led to resonance.
2. **Hazard Elimination**



    a. Better to use a second one – if there is a short we don't get a blown-up battery.
    b. Some architecture and tool choices can eliminate some types of software hazards
        1. Strongly-typed language limits syntax errors and memory leaks
    c. But, hazards are generally more than software.
3. **Ariane 5, June 4th 1996 – European Space Agency**
    a. Ariane 5 accelerated faster than Ariane 4
    b. Caused operand error in float-to-integer conversion
    c. Backup inertial navigation set dumped core
    d. Core interpreted by the live set as flight data

e. Full nozzle deflection → 20-degree angle of attack → booster separation → protective detonation

4. **Therac Accidents (Therapeutic Accelerator)**
   a. Radiotherapy Machine sold by AECL (Atomic Energy Canada)
   b. Between 1985 and 1987, three people died in six accidents
   c. Caused by fatal coding error, compounded with usability problems and poor safety engineering.
   d. What happened?
      1. Safety requirement: don't fire 100% current at human
      2. 25 MeV with two modes of operation
         1. 25 MeV focused electron beam on target to generate X-rays
         2. 5-25 MeV spread electron beam for skin treatment (with 1% beam current)
      3. Previous version had mechanical interlocks to prevent high-intensity beam use unless X-ray target in place
         1. Therac-25 replaced these with software
      4. Fault-tree analysis assigned probability of 10^{-11} to 'computer selects wrong energy'
      5. Code was poorly written, unstructured and really documented.



      6. Datent sets turntable and 'MEOS' which sets mode and energy level
         1. Data entry complete can be set by datent or keyboard handler
         2. If MEOS set (& datent exited), then MEOS could be edited again – using the keyboard handler.
   e. Accidents
      1. Marrietta, GA, June 85: woman's shoulder burnt – settled out of court (FDA not told)
      2. Ontario, July 85: woman's hip burnt. AECL found microswitch error but couldn't reproduce fault – they changed the software anyway.
      3. Yakima, WA, December 95: another woman's hip burnt – couldn't be a malfunction.
      4. East Texas Cancer Centre, March 1986: man burnt in the neck and died five months later of complications
      5. Same thing happened – man burned on face and died three weeks later
      6. Then physicist managed to reproduce the flaw: if parameters changed too quickly from x-ray to electron beam, the safety interlock failed.
         1. Beam-type change interface with "x" not particularly linked to x-ray.

2. This was not very usable due to poor software design.

7. Yakima, WA, January 87: man burned in chest and died – different bug thought to have caused Ontario accident.

f. Why?

1. AECL had ignored safety aspects of software
2. Confused reliability with safety
3. Lack of defensive design
4. Inadequate reporting, follow-up and regulation
5. Unrealistic risk assessments
6. Inadequate software engineering practises – specification was an afterthought, complex architecture, dangerous coding, little testing, careless HCI design

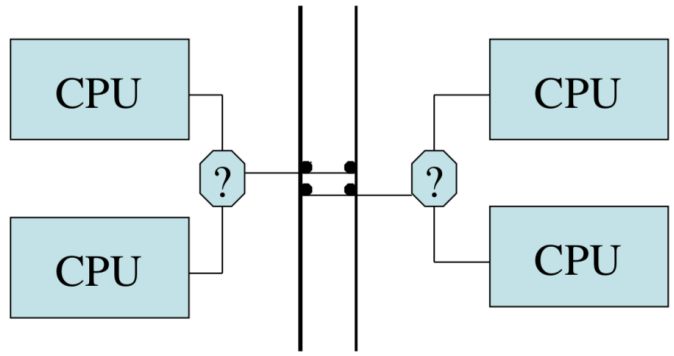
5. Panama Crash, June 6th 1992

a. Not knowing which way is up!
b. There was an EFIS (for each pilot), WW2 artificial horizon was the backup.
c. EFIS failed – there was a wire loose. This failed for both pilots as they both fed off the same IN set.
d. The pilots watched EFIS, not the artificial horizon, therefore led to 47 casualties.
e. The same things happened in 1999.
f. Cockpit design issue – pilots did not know they had other options.

Software Safety Myths

1. Computers are cheaper than analogue devices
    a. Shuttle software cost $\$10^8$ pa to maintain
2. Software is easy to change
    a. But hard to change safely
3. Computers more reliable
    a. Shuttle software had 16 potentially fatal bugs found since 1980 – and half of them had flown
4. Increasing reliability increases safety
    a. Correlated but not completely
5. Reuse safety myths
    a. Not in Ariane, etc
6. Formal verification can remove all errors
7. Testing can make software arbitrarily reliable
8. Automation can reduce risk
    a. Often takes an extended period of evolution
    b. But things like redundancy

Redundancy



But, software is generally where things broke then (backup IN set in Ariane). This introduces the idea of multi-version programming. However, this leads to problems like (1) error correlation, (2) dominated by failures to understand the requirements. Also different implementations also often give different answers.

Understanding Hazards – motor industry

1. Uncontrollable – outcomes can be severe and not influenced by humans
2. Difficult to control – very severe outcomes, influenced only under favourable circumstances
3. Debilitating
4. Distracting
5. Nuisance

Issues to consider:

1. Develop safety case
2. Who will manage what? Trace hazards to hardware, software, procedures.
3. Trace constraints to code and identify critical components
4. Develop safety test plans, procedures, certification, training
5. Figure out how all this fits with development methodology.

Soft spots are requirements engineering, certification and then operation / maintenance – these are interdisciplinary, involving systems people, domain experts and users, cognitive factors, politics and marketing.

Certification: Things have a 10-year certification cycle – this has problems. Therefore, need to go to monthly updates, stressing regulators.

Development

Tools

Types of Complexity:

1. Incidental Complexity
    a. Dominated programming in the early days
    b. Keeping track of stuff in machine-code programs
    c. **Managing Incidental Complexity**
        i. Invention of high-level languages.
            1. More dense code

2. Code easier to understand and maintain
3. Date abstraction
4. Compile-time error detection
5. Portability of code – machine-specific details may be contained
6. Performance gain: 5-10 times

ii. Helping programmers structure and maintain code
1. Don't use 'goto'
2. Structured programming
3. Information hiding
4. Object-oriented programming

iii. Time-sharing systems
1. Allowed online test – debug – fix – recompile – test
2. Still needed plenty of scaffolding and careful debugging plan

iv. IDEs

v. Formal Methods
1. Z notation for specification
2. HOL functions for hardware – formal verification
3. Burrows-Abadi-Needham Logic – crypto logic
a. Set of rules for defining and analysing information protocols.

2. Intrinsic Complexity
a. Main problem today
b. Complex system (such as a bank) with a big team.
c. Solution is with structured development, project management tools.

Static Analysis Tools

Outcome of formal-methods community is modern static analysis tools – things like Coverity don't expect to find all bugs, juts many of them. However, when you buy the tool, you find many bugs and the ship date slips. This occurs every time you update the analysis tool.

Programming Philosophies: IBM, 1970-72: Idea of having a hierarchical group of people to program. This is effective during implementation – but each team of people (chief programmer, apprentice, toolsmith, librarian, admin assistant, etc) can only do so much.

Egoless Programming: Code should be owned by the team, not by any individual.

N.B. MS System: Developers, not analysts, programmers, testers -> bugs fixed by the same person. Therefore, they slow down the bad people.

Literate Programming (Knuth): Code should be a work of art, aimed not just at a machine but also future developers.

Capability Maturity Model

Keep teams together, as productivity increases over time:

- Humphry, 1989
- Nurtures the capability for repeatable, manageable performance, not outcomes that depend on individual heroics.

- Leads to the development of CMM developed at CMU with DoD money

This identifies five levels of increasing maturity in a team or organisation, and a guide for moving up:

1. Initial – 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 as a standard business process
4. Managed – the process is managed according to the metrics described in the Defined stage
5. Optimised – process management includes deliberate process optimisation

Trends in development style:

- Emphasis shift from requirements to testing to people
- 1990s: lots of effort into spec
- 2000s: major effort in an incremental build system with an automatic regression test environment

Agile Development

Extreme Programming

Aimed at small teams working on iterative development with automated tests and a short build cycle. The ideas are:

- Episodic: Small teams working on iterative development with automated tests and short build cycle.
- Solve worst problem. Repeat
- Write tests then code – tests are the documentation
- Programmers work in pairs – one keyboard and one screen
    - This didn't survive, but episodic idea did – people added the idea of scrum

Agile Today:

- Sound technical foundation: languages with a build environment and automated testing methodology – design with testability in mind
- Agree processes: daily scrum, weekly stand-up, customer interaction, (short) sprints
- Consider other important parts – security policy, safety case, real-time constraints.

Important to note that the specification **still matters**. In a study of failure of 17 demanding systems, Curtis, Krasner and Iscoe and 1998, causes of failure were threefold: But getting it right is hard

1. Low application domain knowledge
    a. Often hard to find people – even when you do, you are likely to get specification mistakes
2. Changing requirements (often conflicting)
    a. There are often good reasons for this happening
        i. Competing products, new standards
        ii. Changing environment
        iii. New customers
3. Breakdown of communication and coordination

But also, spec can be unhelpful:

1. Spec-driven development leads to communications problems
2. Big firms do hierarchy – but if information flows via the least common manager, then the bandwidth will be inadequate.
3. So, need committees
    a. This leads to politicking
4. Management attempts to gain control results in restricting many interfaces – to the consumer for example.

Project Management

Manager's job is to (1) plan, (2) motivate, (3) control. They have a number of tools for planning:

1. Gantt Chart
    a. Shows tasks and milestones
    b. But hard to visualise the dependencies
2. PERT Chart – draw as a graph with dependencies
    a. Allows us to do critical path analysis
    b. Helps warn of impending trouble

Motivating:

- People often work worse in groups than on their own:
    - Free ride / social loafing effect
- 3 Cs of motivation
    - Collaboration – everyone has a specific task
    - Content – everyone's task matters the same
    - Choice – everyone has a say in what they do
- Acknowledgement, attribution, equity, leadership and team building

Documentation

Important for a PM to consider how to deal with management documents (budgets, PERT charts, staff schedules) and engineering documents (requirements, hazard analyses, test plans, code). Partial solutions:

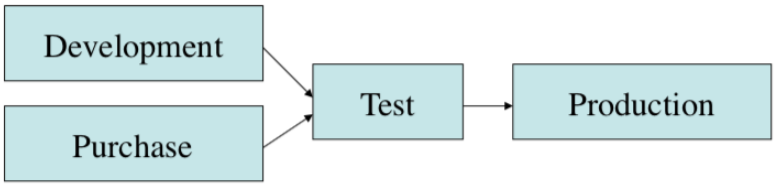
- High tech
    - IDEs and version control software
- Bureaucratic
    - Plans and controls department
- Social consensus
    - Style, comments and formatting

Release Management

More bugs will always be found as you prepare for release – Sod's Law. The main focus with release is stability – but adding things like (1) copy protection, (2) rights management. The critical decision is always whether to patch old versions or to force upgrades.

Change Control

Change Control is important – manage the testing and deployment of the new software. Must assess the risk and take responsibility for live running, and manage the backup, recovery, rollback, etc.



Vulnerabilities

If a bug is found, it could be disclosed responsibly, revealed immediately or exploited. There is a primary exploit window until the patch is shipped. But it is important to not that many old devices aren't patched. This allows attacks like Mirai, where networked devices which have a certain vulnerability (running Linux (especially IoT devices) – logging in using a list of common factory passwords) are used in DDOS attacks.

Responsible Disclosure: Old approach – try and deny existence of bugs for as long as you can. Therefore, new approach is disclosing vulnerabilities to CERT, which tells creator then publishes after a time delay to allow patches to be created.

Shared Infrastructure: This is the idea of sharing open source code. This has a number of benefits, but also interaction issues – including the idea of responsible disclosure and different license terms.

Agency Issues: Based on fact that employees optimise their own utility, not the project's – people avoid blame. Tort's law reinforces herding: negligence is judged to the standards of the industry. Therefore, (1) use checklists, (2) use the correct tools, (3) hire consultants to verify people's work.

Knowing when you're done

1. Cathedral – software built by a group of developers based on a central plan
    a. Common Criteria – International standard for computer security certification
        i. Provides assurance that the process of specification, implementation and evaluation has been conducted in a rigorous, standard and repeatable manner at a level commensurate with the target environment.
    b. Protection Profiles
        i. Provides an implementation independent specification of information assurance safety requirements
        ii. Generally, a combination of threats, security objectives, assumptions, security functional requirements, security assurance requirements/
        iii. Used to substantiate vendors' claims of a given family of products. Specifies an Evaluation Assurance Level – indicates the depth and rigor o the security evaluation.
        iv. NIST and NSA cooperate to produce validated US government PPs
    c. CLEF

        i. Similar equivalent for the UK
    d. Security accreditations
2. Bazaar – open-source software
    a. Patch cycle
    b. Responsible disclosure
    c. Breach reporting

Safety: Mostly the Cathedral approach rather than the Bazaar. We have sets of regulators for each of the governments and different products, FAA / CAA for aircraft, UNECE / independent lab testing (Europe) for card.

Focus on Outcomes or the Process

Outcomes	Process
• The metrics are easier for them • Rare catastrophes have a large uncertainty • Accidents are random but not security exploits	• Necessary to adapt • Security development lifecycle is established • Compliance

Incentivisation: The world offers a hostile review – dogfood, alpha, beta, ops. This is sometimes useful to some applications to get higher assurance of CC.

Testing

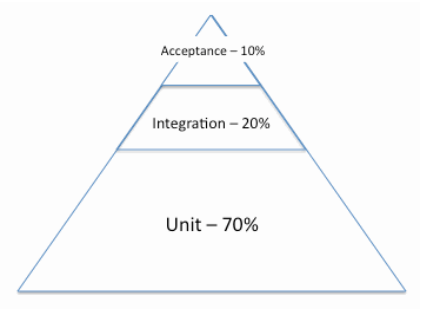
Some problems can be detected statistically – including type issues which can easily be detected. However, a number of problems cannot be detected – and therefore they must be found through testing.

Testing: Checking of how software performs at run-time. We have input values passing through a system under test, which leads to a certain output behaviour. These pass through an 'oracle', which tests if we have passed or failed the test.

Types of Testing

1. **Unit Tests**
    a. Checking isolated pieces of functionality
2. **Integration Tests**
    a. Checks that the parts of a system work together
3. **End to End (E2E) Tests**
    a. Simulates real-user scenarios

Together, these form a 'testing pyramid': 70% of the tests are unit tests, 20% of tests are integration tests and 10% are E2E tests. Unit tests are simple and cheap and as you move up the pyramid, the tests get more complex and expensive.



With them, we cover testing at all the levels:

- Design validation, UX prototyping
- Module testing after coding
- System test after daily build
- Beta test / field trial
    - Cost per bug rises dramatically as you move down the list

Unit Testing Example and Important Points

```
static long calculateAgeInDays (String dateOfBirth)
{
    Instant dob = dateFormat.parse(dateOfBirth).toInstant();
    Instant currentTime = new Date().toInstant();
    Duration age = Duration.between(dob, currentTime);
    return age.InDays();
}
```

This is a non-hermetic test – the test relies on an external parameter. We can fix this with dependency insertion – there exist dependency injection frameworks to allow you to solve this.

1. Design classes for tests – with dependency insertion if necessary
2. Test naming
    a. Name tests very specifically – what's being tested, what's the input, what's the expected output.
3. One property per test
    a. Allows you to very easily see what the problem is with.
4. Arrange, Act, Assert
    a. Arrange all necessary preconditions and inputs
    b. Act on the object or method under test
    c. Assert the expected results have occurred.
5. Writing assertions
    a. The assertions need to be aware of the datatype – including testing for close to, for floating point numbers for example.
6. JUnit Lifestyle
    a. For any test, the following things will occur
    b. (1) Instance of the testing class created
    c. (2) The Before function is run
    d. (3) The actual test function is run

    e. (4) The instance will be destroyed (even if other tests in the same testing class).
7. Using @Before vs constructors
    a. @Before tends to be better

Mocking

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;

LinkedList mockedList = mock(LinkedList.class);
// can specify behaviour that you want
when(mockedList.get(0)).thenReturn("first");
mockedList.add("added");
// assert that things got called
verify(mockedList).add("added");
```

Mocking allows us to probe a class object to see what has happened to it.

Flaky Tests

Flaky tests are tests that must have some reliance on uncontrollable factors – therefore run on the same code, it may sometimes pass and may sometimes fail. An example of the number of flaky tests is the statistical number of tests as below:

	% of tests which are flaky
All tests	1.65%
Java web driver	10.45%
Android emulator	25.46%

Automated test generation

Automated test generation can find unnoticed bugs:

- One approach is random testing
- We generate the inputs at random
- And we use search to refine the inputs to make them more effective
- We can check for bad things like a buffer overflow.
- This has been used for a number of things including finding thousands of security vulnerabilities in open source code (including operating systems).

Code Coverage Testing

We can create tests that ensure that the entirety of the code (and test code) is run – the number of lines. Clearly, running all the code does not necessarily mean that the code does not have a bug.

Test coverage can use a number of properties:

- Statement Coverage
    - All lines executed
- Branch Coverage
    - All decisions explored at each branch

- Path Coverage
    - All paths through the program were taken
- Data Flow Coverage
    - Is every possible definition (data input) tested? – this is generally impossible to really do

Mutation Testing

Mutation testing can tell us how robust our tests are. In order to do this, we generate small changes to the program that we are testing – including:

1. Changing + to –
2. Changing a constant term
3. Negate a condition

Then, we ensure that the test fails – this implies the test works.

Integrating testing into software engineering process

Firstly, it is important to note that defects in software are inevitable – 1-25 errors per 1000 lines for delivered software. Additionally, 80% of errors are in 20% of the project's classes.

Test Driven Development

Uses tests as specification:

1. Write tests which demonstrate the desired behaviour
2. Implement new functionality
3. Check tests now pass
4. Repeat

+:

- Guarantees that you write tests
- That code is testable
- Tests can be written that directly describe the customer's requirements

-:

- Requires an early commitment to how the project will work
- Changes in approach are hard
- Some areas are more important to test than others

Fix when found:

- When we find a problem, we need to know that we've fixed it
- Once we fix a bug, it needs to stay fixed.

Continuous Integration

Run a test suite on every change – can reject changes which break tests or just report. (Solves the problem or not wanting broken code to be committed to the repository). 20% of bug fixes reintroduce failures in already tested behaviour.

Regression Testing:

1. Write tests that exercise existing functionality
2. Develop new code
3. Run tests to check for regressions

Bug Fixing with Regression Testing:

1. Write test that reproduces the bug
2. Check that it fails
3. Fix the bug
4. Check that the test passes

However, we cannot run all the tests on every change as we need to deliver the results to developers every day. Also, we need to manage the execution cost of running tests. There are a number of strategies:

1. Test Suite Minimisation
    a. Choose a subset of tests which achieve coverage on the project
        i. This is an NP-Complete Problem, so we have to use heuristics
        ii. If some test is the only test to satisfy a test requirement, then it is an essential test
        iii. Therefore, we (1) choose all essential tests and (2) choose remaining tests greedily in terms of coverage added.
2. Test Set Selection
    a. Choose a subset of tests which are appropriate for the change submitted
3. Test Set Prioritisation
    a. Choose an ordering such that tests are more likely to find a defect are run earlier

Reliability Growth Models

Help us to assess the mean time between failures, number of bugs remaining and the economics of further testing. Failure rate due to one bug is $e^{-k/t}$, with many bugs, this sums to k/t. Therefore, for 10^9 hours mtbf, we must test > 10^9 hours.