Critical Software:

Martyn Thomas CBE FREng

www.thomas-associates.co.uk

Please INTERRUPT with questions …

Software is CRITICAL

- Part of most critical systems
- Usually on the critical path

and, as we shall see,

- the software profession (in which I have worked for the past 40 years) is in a critical state

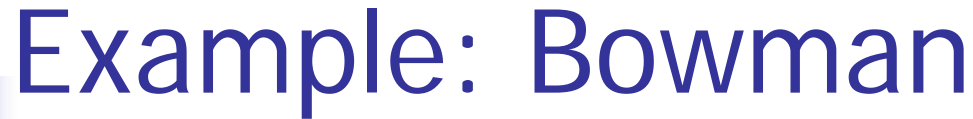
Software projects often overrun

- **Standish "Chaos Chronicles" (2004 edition):**
  - 18% of projects "failed"; (cancelled before completion)
  - 53% of projects "challenged" (operational, but over budget and/or over time with fewer features or functions than initially specified...)
- **Typical Standish figures:**
  - Cost overruns on 43% of projects; and
  - Time overruns on 82% of projects.
- **Oxford University/Computer Weekly 2003 study:**
  - 10% of UK projects "failed"; and
  - 75% of UK projects "challenged".

Why Projects overrun: MANAGEMENT ISSUES

- The requirements were not properly understood, recorded,and analysed - so there were many unnecessarily late changes
- Related hardware or business changes and risks were not planned, budgeted and managed competently
- Requirement changes were not kept under control and budgets and timescales were not adjusted to reflect essential changes
- Stakeholder conflicts were not resolved before the computing project started

Example: Bowman



When I looked at Bowman, ten years ago, it was:

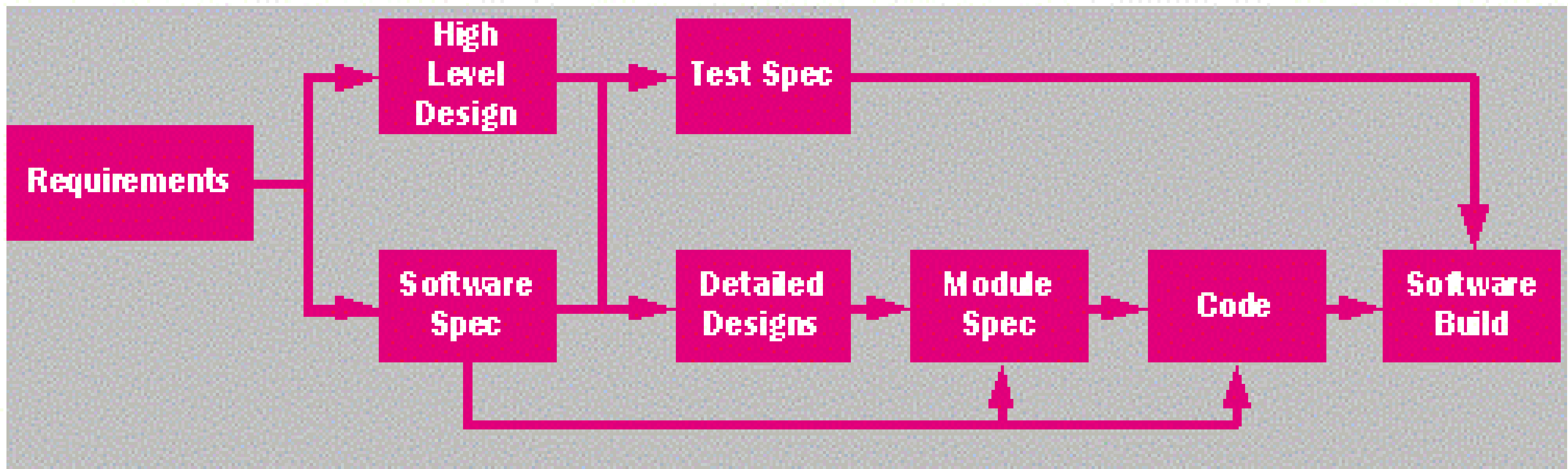
- "A systems integration of COTS components"
  - but with a million lines of custom software
- Required to be the infrastructure for time-critical and safety-critical communications
  - but not designed to *guarantee* message delivery

These were management, not technical issues

From Needs to Systems [1]

- Need: A digital automobile odometer for recording trips and total mileage
- Requirements: The system *shall* record and display total mileage travelled. The user *shall not* be able to reset the total. The system *shall* record and display trip mileages. The user *shall* be able to set the trip counter to zero …

From Needs to Systems [2]



From Needs to Systems [3]

Informal methods

- Needs: English
- Req: English
- Design: diagrams, English, pseudocode
- Code: (e.g. C)
- Test: based on Req
- System: >10 faults/KLoC

Formal methods

- Needs: English
- Req: English AND rigorous logic
- Design: diagrams, English AND rigorous logic
- Code: (e.g. Ada)
- Test: based on Req AND Proof
- System: <1 fault/KLoC

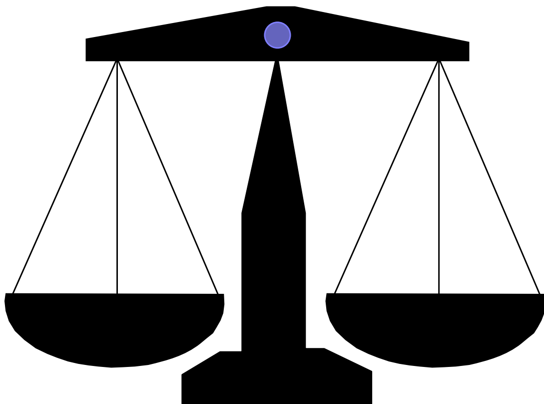
Why Projects overrun: SOFTWARE ISSUES [1]

- No Formal Specification, so:
  - no rigorous analysis for contradictions and omissions in the requirements
    - so requirements errors are found late
  - a weak basis for verifying the design
    - so design errors are found late
  - a weak basis for designing tests
    - acceptance testing will be controversial
  - likelihood of ambiguity
    - misunderstandings will cause rework, especially around interfaces.

Why Projects overrun: SOFTWARE ISSUES [2]

- Development methods are error-prone, and allow errors to propagate
  - design languages with weak or no analysis tools to support them
  - programming languages with weak type-systems and weak analysis tools
- Reliance on the conventional development philosophy: *“Test and Fix”*

Testing software tells you that the tests work – not that the software works



Continuous behaviour means you can interpolate between test results

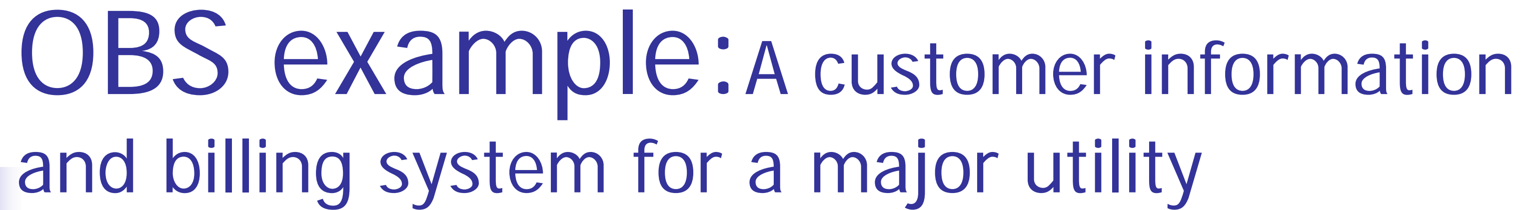
Discrete behaviour means that you can't!

Beware “agile methods”

- Excellent for prototyping or where the required product is not complex and can be allowed to fail in service.
- Dangerous where
  - they are an excuse for delaying agreement on the requirements
  - the system is safety-critical or security-critical or where in-service failures would be very damaging
  - the system architecture is likely to be complex and expensive to change
  - the system will have a long in-service lifetime

Beware "output-based specifications"

- A good idea: say *what* you need to happen not *how* to achieve it.
- BUT often an excuse to leave most of the requirements analysis until after the budget and timescales have been agreed and the contract is in place
  - every change will now increase cost, delay and risk



OBS example: A customer information and billing system for a major utility

- Package and supplier chosen on the basis of an Output Based Specification. Target duration, 15 months
- Detailed requirements analysis took a year
  - detailed interfaces to other systems
  - statutory report formats
  - statutory constraints of handling of delinquent accounts
  - special charging tariffs with hundreds of allowed combinations
  - statutory constraints on which users had access to which customer data
  - etc
- Timescales slipped by 18 months and nearly bankrupted the company

Software Systems are usually not dependable

- Security vulnerabilities
  - e.g. *Code Red* and *Slammer* worms caused $billions of damage and infected ATMs etc
- Safety-critical faults
  - current certification requirements are completely inadequate
- Requirements errors
  - the important requirements lie well outside the software!
- Programming mistakes
  - COTS software contains thousands of faults

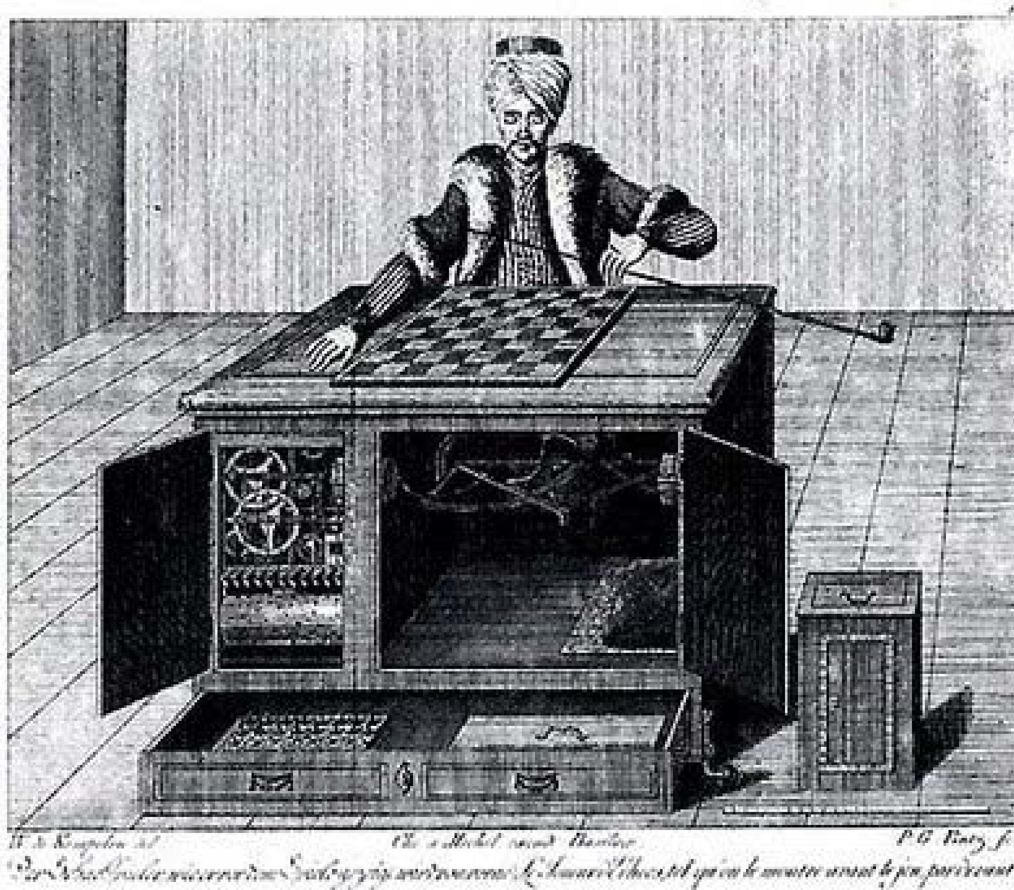
Requirements Problems

- **what happened**
- Airbus A320, Warsaw 1993
- aircraft landed on wet runway
- aquaplaned, so brakes didn't work
- pilot applied reverse thrust, but disabled
- **why**
- REQ: airborne ⇔ disabled
- ASSUME not WheelPulse⇔ airborne ⇔ disabled
- **simplified;** for full analysis, see [Ladkin 96]

Coding Errors

```
type Alert is (Warning, Caution, Advisory);
function RingBell(Event : Alert) return Boolean
-- return True  for Event = Warning or Event = Caution,
-- return False for Event = Advisory
is
  Result : Boolean;
begin
  if Event = Warning then
    Result := True;
  elsif Event = Advisory then
    Result := False;
  end if;
  return Result;
end RingBell;
-- C130J code: Caution returns uninitialised (usually
   TRUE, as required).
```

Don't trust demonstrations …



Wolfgang von Kempelen's Mechanical Turk

Almost all software contains very many faults

- Typical industrial / commercial software development:
  - 6-30 faults delivered / 1000 lines of software
    - 1M lines: 6,000-30,000 faults after acceptance testing

source: Pfleeger& Hatton, IEEE Computer, pp33-42, February 1997.

Customer *beta-testing* has become accepted practice

© UFS, Inc.

Example: Safety Related Faults

- Erroneous signal de-activation.
- Data not sent or lost
- Inadequate defensive programming with respected to untrusted input data
- Warnings not sent
- Display of misleading data
- Stale values inconsistently treated
- Undefined array, local data and output parameters

More safety related faults

-Incorrect data message formats
-Ambiguous variable process update
-Incorrect initialisation of variables
-Inadequate RAM test
-Indefinite timeouts after test failure
-RAM corruption
-Timing issues - system runs backwards
-Process does not disengage when required
-Switches not operated when required
-System does not close down after failure
-Safety check not conducted within a suitable time frame
-Use of exception handling and continuous resets
-Invalid aircraft transition states used
-Incorrect aircraft direction data
-Incorrect Magic numbers used
-Reliance on a single bit to prevent erroneous operation

Errors found in C130J software *after* certification.

Source: Andy German, Qinetiq. Personal communication.

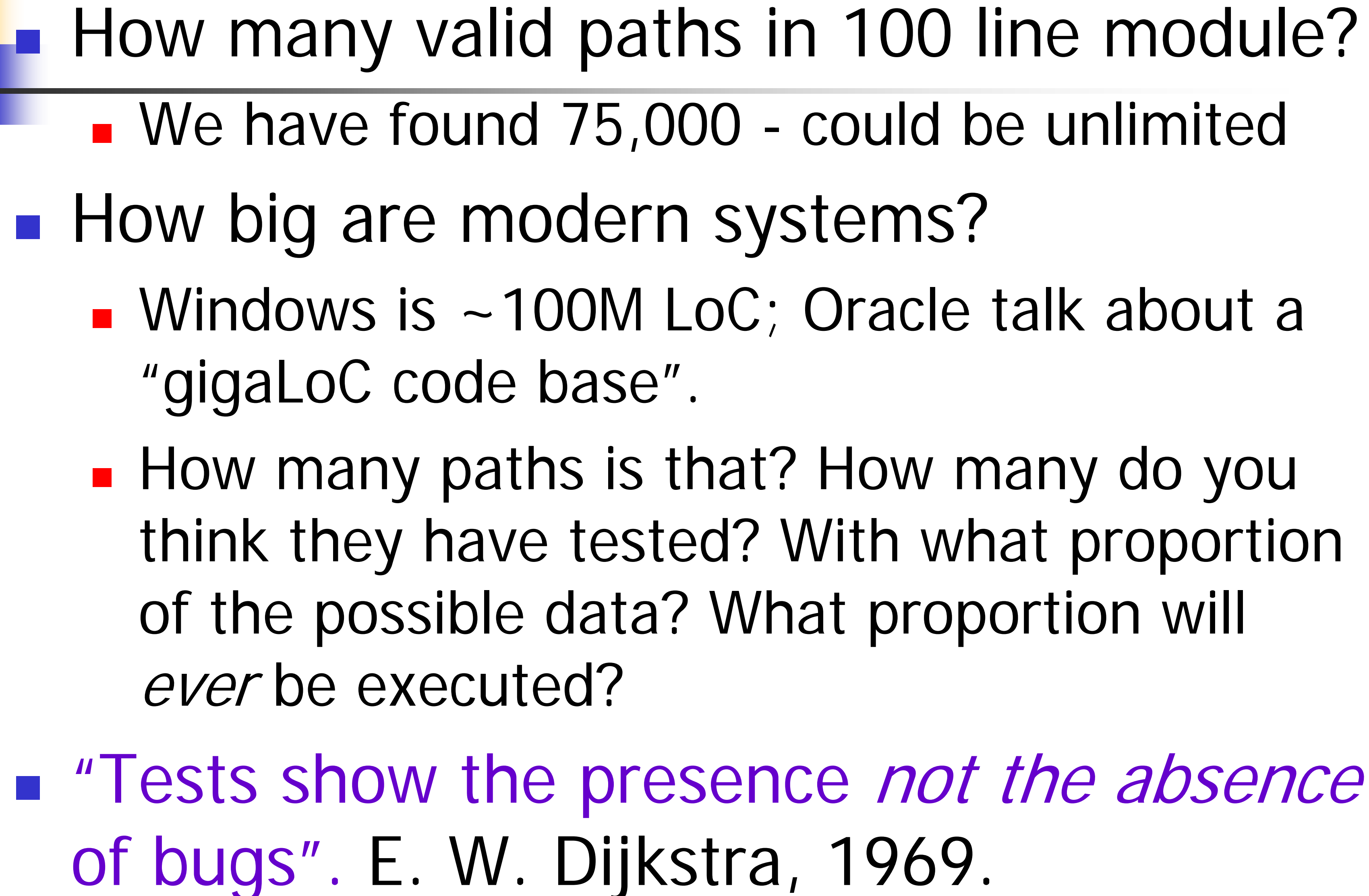
Why don't companies adopt methods that avoid these faults?

We are like the barber-surgeons of earlier ages, who prided themselves on the sharpness of their knives and the speed with which they dispatched their duty -- either shaving a beard or amputating a limb.

Imagine the dismay with which they greeted some ivory-towered academic who told them that the practice of surgery should be based on a long and detailed study of human anatomy, on familiarity with surgical procedures pioneered by great doctors of the past, and that it should be carried out only in a strictly controlled bug-free environment, far removed from the hair and dust of the normal barber's shop.

Professor Sir Tony Hoare

Testing can never be the answer



Most software costs flow from error detection and correction

- The cost of correcting an error rises steeply with time
  - Around 10-fold with each lifecycle phase
  - So specification errors are 100 times more costly than coding errors if both are found by testing.
- The only way to reduce costs, duration and risks is to greatly reduce errors and to find almost all the rest almost immediately.

Strong Software Engineering

- Objective: Avoid errors and omissions
  - … and detect errors before they grow in cost
- How? The same way other engineers do
  - Explore what you should build. Create *precise* but *high-level* descriptions.
  - Gradually add detail in the design, doing the hardest things first
  - Use powerful software tools *at every stage* to check for errors and omissions
- Result: < 1 error / KLoC at no extra cost!

How do you get the right technical solution to a business requirement?

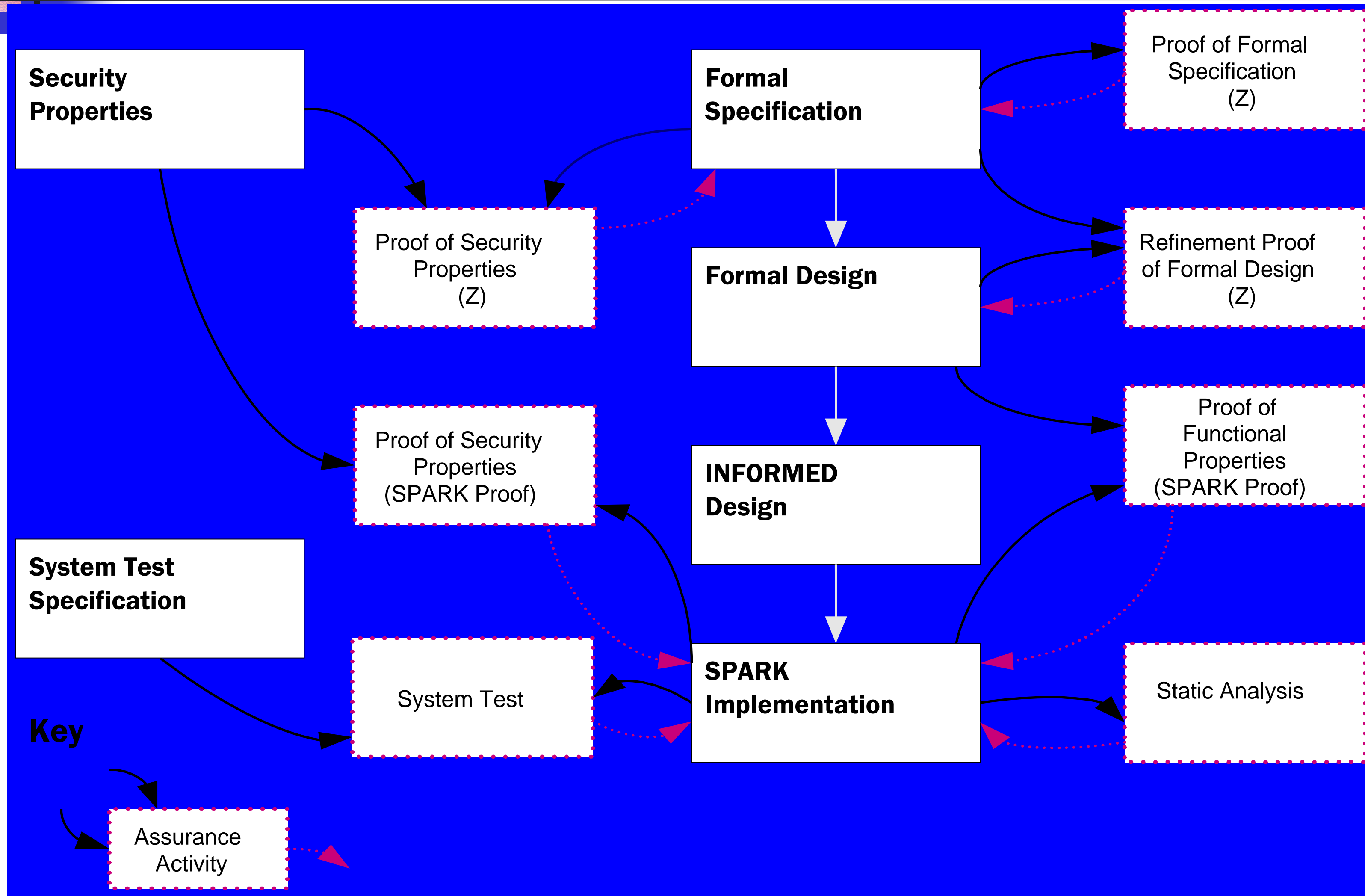
USE AN ARCHITECT!

See the Royal Academy of Engineering report on complex IT Systems.

Role of the Systems Architect

- Help the customer to understand the requirements and possibilities
- Propose appropriate and technically feasible high-level solutions (architectures)
- Help resolve stakeholder conflicts and agree requirements and architecture
- Complete and FORMALISE the technical specification *This will eliminate most requirements risk.*
- Manage supplier selection
- Manage the supply contract for the customer
- Manage requirement changes
- Manage the user acceptance phase

Then use *Correct by Construction* development



C by C Guiding Principles

- Write right
- Step, don't leap
- Say something once
- Check here before going there
- Argue your corner
- Use the right tools for the job
- Use your brains

Write Right

Capture information, requirements, designs and code unambiguously and accurately.

Use notations with strong semantics

Step, Don’t Leap

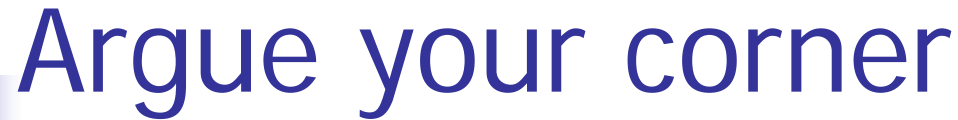
Make each development step semantically small, to reduce errors and improve checking.

Say something once

Make each document have a clear purpose and write information in only one place. This aids clarity and avoids inconsistency.

Check here before going there

Verify each step before moving on. Check for consistency against something else – usually the previous design step.



Argue your corner

Document design decisions and explain why they are justified – that will help you be sure they are right, and help you and others to understand them later.

Use the appropriate verification method for each property, whether tool-supported proof, informal peer review, static analysis or testing.

Use your brains

Think about everything you do.
Don’t put off difficult analyses.
Discuss the requirements carefully
and in depth with your customers.
Search intelligently for conflict.

OVERVIEW- Correct by Construction (C by C) Process

- A software engineering process employing good practices and languages
  - SPARK (Ada 95 subset with annotations)
  - math based formalisms (Z) at early stages for verification of partial correctness.
- A supporting commercial toolset (Z/Eves, Examiner, Simplifier, Proof Checker) for specifying, designing, verifying/analyzing, developing safety or security critical software.

Taken from an NSA presentation

The Development Approach

- Requirements Analysis Step (REVEAL approach)
  - Identify system boundaries
  - Clarify dependencies on environment
- Security Analysis
  - Develop Security Target & Security Policy Model (CC) using Protection Profile
  - Identify key properties to ensure security
  - Validating functional spec with security properties
- Specification
  - Define and document customer requirements in Z and English with customer feedback

Development Aproach (continued)

- Design (w/ (semi)formal documents)
  - Refined functional spec (written in Z)
  - INFORMED design document
    - Details data store and flow, dependencies of modules (SPARK packages), etc.
    - Links design statements to implementation modules- straightforward and tool supported
  - Provides baseline orthogonal documents for developers and testers– functional specs(Z), design docs (Z for behavior and environment dependencies), test specs

Development Approach (continued)

- Implementation
  - Coding in SPARK Ada with static analyzer (EXAMINER)
    - prevents uninitialized variables, buffer overflows, incorrect info flows
- System test
  - Incremental builds with increasing functionality
  - Specification based testing
- Assurance
  - Proof of key properties (conformance to specification and no run-time exceptions)

```
package Odometer
--# own Trip, Total: Integer;
is
  procedure Zero_Trip;
  --# global out Trip;
  --# derives  Trip from ;
  --# post Trip = 0;

  function Read_Trip return Integer;
  --# global in Trip;

  function Read_Total return Integer;
  --# global in Total;

  procedure Inc;
  --# global in out Trip, Total;
  --# derives Trip from Trip & Total from Total;
  --# post Trip = Trip~ + 1;
End Odometer
```

-- *example taken from* ***High Integrity Software*** *(SPARK book by* ***John Barnes****)*

The Tokeneer Experiment

see http://www.praxis-his.com/pdfs/issse2006tokeneer.pdf

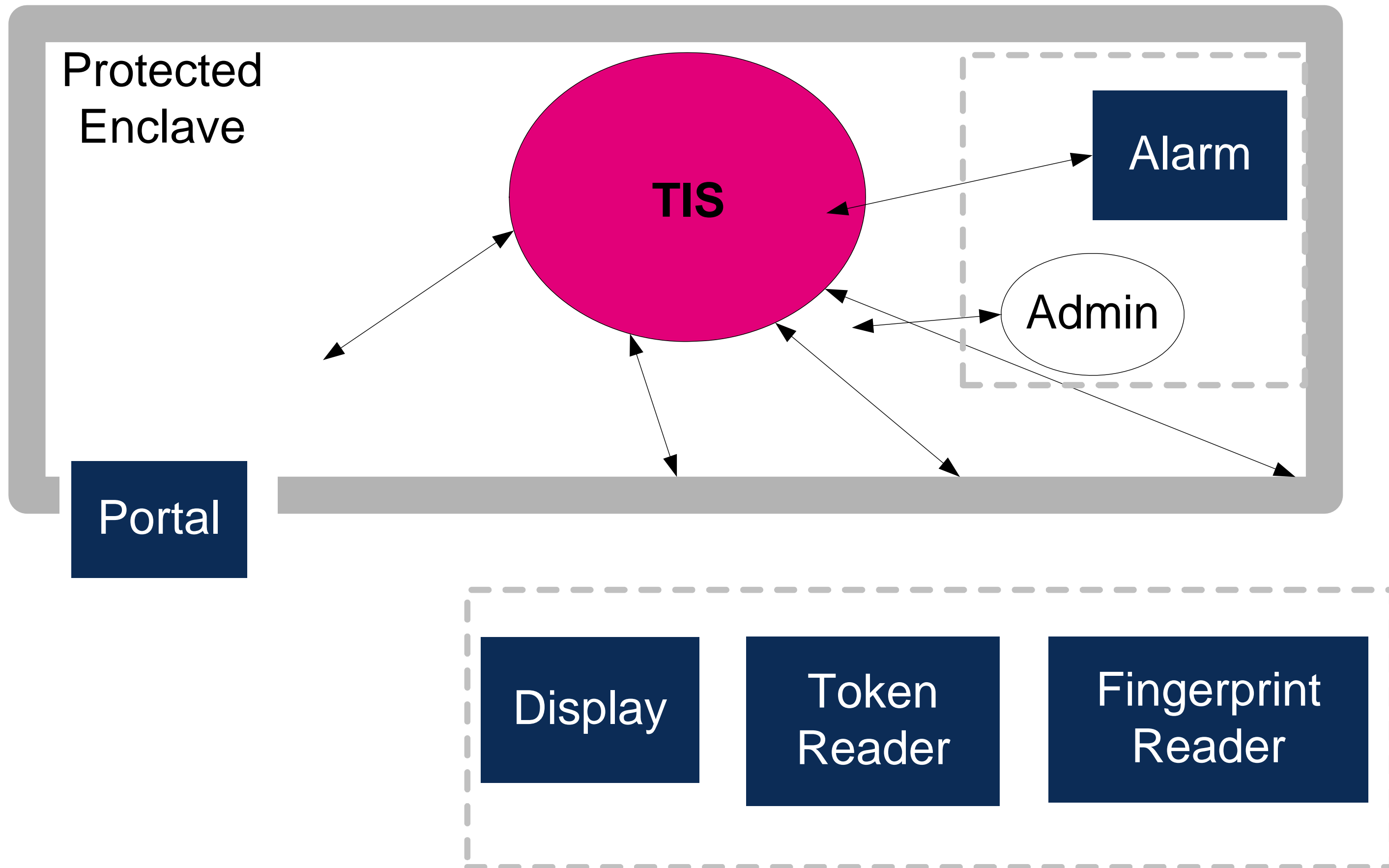
From a presentation by

Randolph Johnson
National Security Agency
drjohns@orion.ncsc.mil

Tokeneer Identification Station background

- Sponsored and evaluated by Research teams token & biometric and HCSS
- Developed by Praxis Critical Systems
- Tested independently by SPRE Inc., N.M.
- Adapted and extended by student interns

TOKENEER ID Station



Statistics of System

	Ada Source Lines	Spark annotations	LOC/day (Ada only)
Core	9,939	16,564	38
Support	3,697	2,240	88

Additional metrics

- Total effort   260 man days
- Total cost – $250k
- Total schedule – 9 months
- Team – 3 people part-time
- Testing criterion – 99.99% reliability with 90% degree of confidence
- Total critical failures – 0 [Yes, zero!]

WHY use Correct by Construction S/W Engineering?

- Meets Common Criteria and ITSEC security requirements for EAL5 +
- Produces code more quickly and reliably and at lower cost than traditional methods
- Is commercially supported (ORA Canada, Praxis HIS, Pyrrhus Software, SPRE Inc.)
- Reasonable learning curve
- C by C is proven and practical

Taken from Randolph Johnson's Tokeneer presentation

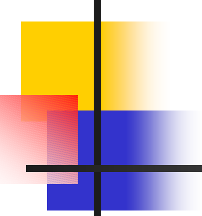
Conclusions

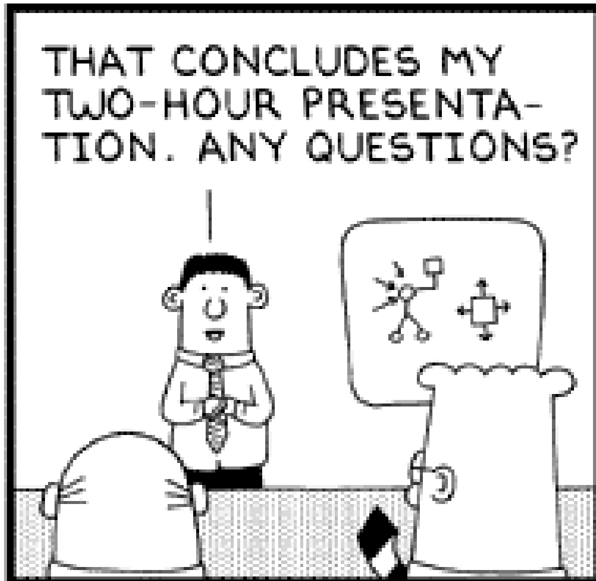
1. We must greatly reduce the risk in complex software projects
2. We must greatly improve the dependability of software-based systems
3. There is a solution

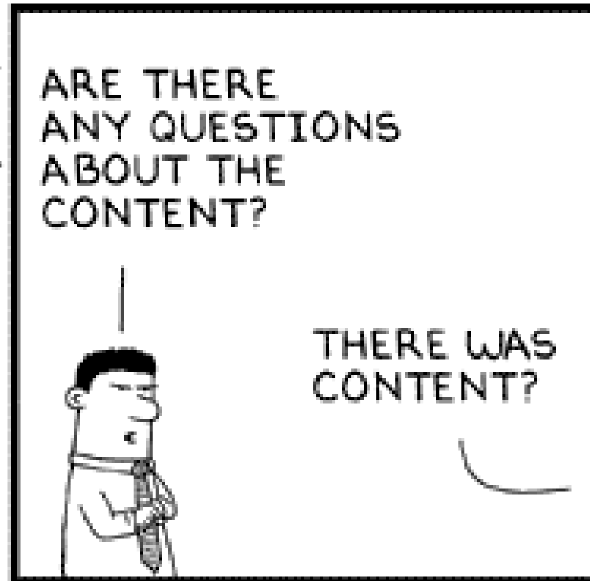
The Solution

- Two-stage procurement:
  1. Use an *independent* architect who can **formalise** the specification
  2. Use Correct by Construction development with strong development methods and powerful analysis tools.
- In other words, science-based software (and systems) engineering

Questions?







© 2003 United Feature Syndicate, Inc.