Software Engineering

Computer Science Tripos Part IA, Part II (General) and Diploma
Lent Term, 1997

Lawrence C Paulson
Computer Laboratory
University of Cambridge

Contents

1 The Software Crisis: 1968 – 2

2 Critical Software 11

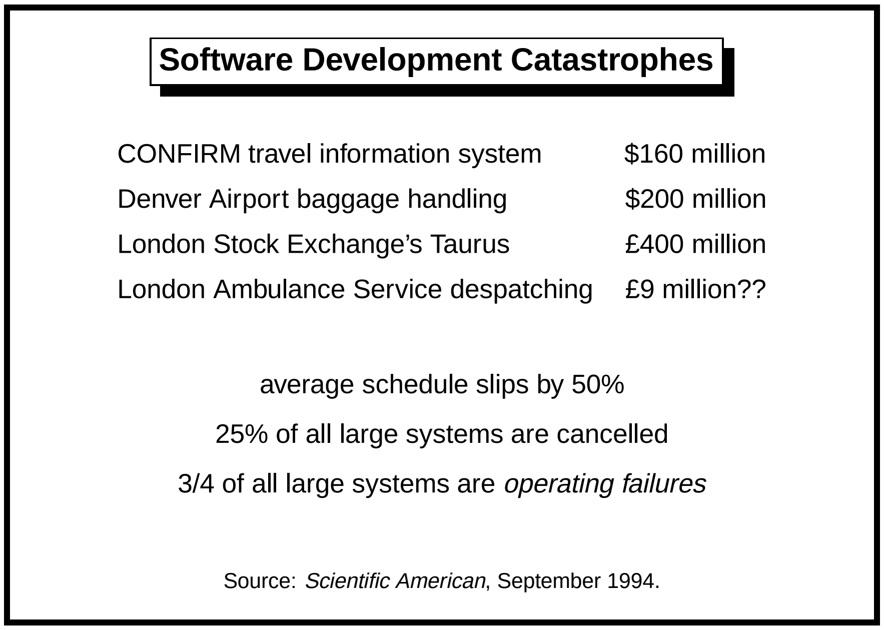
3 The Software Life Cycle 20

4 Formal Methods in Software Development 28

5 Introduction to the Z Specification Language 36

6 Large Software Systems: A Study 44

Slide 101



Software Development Catastrophes

CONFIRM travel information system	$160 million
Denver Airport baggage handling	$200 million
London Stock Exchange's Taurus	£400 million
London Ambulance Service despatching	£9 million??

average schedule slips by 50%

25% of all large systems are cancelled

3/4 of all large systems are *operating failures*

Source: *Scientific American*, September 1994.

These software development projects went badly wrong. Costs are difficult to quantify. Apart from the direct cost of an abandoned project, there could be litigation, loss of reputation, etc. In the case of the Denver airport, each day's delay in opening cost roughly $1.1 million. The system finally went into operation in October 1995, after a manual backup had been built at a cost of $50 million.

The same article describes $144 million spent on a failed air traffic control system. Another part of this system is $1 billion over budget!

Taurus was abandoned after it became clear that it would probably never work [23].

The figure for the London Ambulance Service includes £1.5 million for the recent project and £7.5 million for a previous failed attempt. It is impossible to say whether the ensuing disruption cost any lives.

Anecdotes prove little, but statistics show that far too many large developments fail. We can learn from individual disasters that come to light. Many others are hushed up. By comparison: one could talk of many gruesome plane crashes, but they are a tiny minority of flights; flying is one of the safest modes of travel.

The software crisis is made worse by the dramatic increase in processor power over the last several decades. This increase is unprecedented in other technologies. The term 'software crisis' was coined in 1968, and we have learned a lot since then, but computers are now many orders of magnitude larger. Software projects are growing too, and the larger the project, the worse the problems are. Schedules slip by 100% and the probability of cancellation approaches 50%.

Anatomy of a Disaster: LAS

Slide 102

- developers inexperienced in safety-critical systems
- ‘fundamentally flawed’ system design
- users excluded from the design process
- extreme time pressure, with no realistic testing
- macho management determined to push through

Source: Steve Flowers in the *Guardian* 28/4/1994, page 21

This project arose in response to the failure of a previous attempt to introduce computers, which cost around £7.5 million over three years. It was to be completed within *six months*. The time pressure meant that changes were made without following proper procedures.

The consortium consisted of Apricot, Systems Options and Datatrak; they made the cheapest bid. ‘The design ignored the limitations of radio-base systems in urban areas.’ It is essential to liaise with the intended users of the system, in this case ambulance drivers and despatchers. They know the most about operational conditions — and they have to use the final system.

There were 81 known errors when the system was put into operation. It ran for a day and a half before being shut down. After a further 10-day trial it was abandoned. The LAS has reverted to manual operation. The recent death of an 11-year-old girl, who had to wait nearly an hour for an ambulance, suggests there is a need for a working computerised system.

An inquiry into this fiasco has led to new guidelines (Poise: Procurement Of Information Systems Effectively), which are now being applied within the NHS. The report of the inquiry [26] is a catalogue of errors.

Safety-critical systems must be built to a particularly high standard; see next lecture.

Slide 103

Anatomy of a Disaster: CONFIRM

- inability to integrate two information systems
- managers hiding serious problems from superiors
- staff sacked for refusing to fudge timetables
- too little liaison between clients & developers
- clients changing requirements late into project

Source: Effy Oz in *CACM* 10/1994

This system aimed to combine airline, hotel and rental car reservations. It was built by the developers of SABRE, 'the world's most successful airline reservation system.' But 'the success of one system does not always guarantee the good fortune of a more advanced system.' F. P. Brooks [4] has written of the 'second system effect,' where a successful system leads to an unrealistically complicated second system.

Bad news was buried to an extraordinary degree. At one point, about half the people assigned to CONFIRM were seeking new positions. A consultant was hired to evaluate the project, but his report displeased his superiors and was buried. The project staggered on for another year [24].

Slide 104

Software vs Other forms of Engineering

- Pure design, weightless & intangible.
- Much greater capacity to contain complexity.
- No manufacturing defects, corrosion or ageing.
- Replication does not bring reliability.
- Discrete, not continuous behaviour.

Source: Motor Industry Software Reliability Association (MISRA)

The first two points explain why manufacturers want to use so much software! MISRA are concerned with cars, but the points particularly apply to aircraft.

We speak of ‘software defects’ but all such defects are inherent in the design. The closest thing to ageing is obsolescence, when software can no longer be run due to changes in hardware, libraries, etc.

One cannot make software more reliable by having duplicate copies of it. The Space Shuttle has four processors running identical software in order to cope with *hardware* failure. There is a fifth processor running independently written software, to cope with software failure. It has been observed, though, that independent programming teams make similar mistakes — especially in considering the most difficult situations.

Discrete behaviour means we cannot interpolate test results. We can test a bridge at extreme loads and be assured that it will not fail for small loads. With software, every input is a distinct situation. We have mathematical models for bridges, but not for software.

Slide 105

Intrinsic Difficulties with Software

- *complexity* of the required functions
- *conformity* with existing artifacts & standards
- *changeability*: demands for additional/revised functions
- *invisibility* of structure & operation: no useful drawings or models

Source: F. P. Brooks, ‘No Silver Bullet,’ 1987

Complexity. Software is required to take on a great many primitive functions. A big building does not have to be more complex than a small one — it just has more of the same components. Scaling up software does not mean repeating components, but increasing complexity. *Essential* complexity may be contrasted with *accidental* complexity, say of using assembly language.

Conformity. Software must conform to complex interfaces imposed from outside. A compiler must translate a programming language to machine code, precisely obeying the definitions of the source and target languages. Electronic mail software must conform to complex network transport protocols. Systems that interact with the outside world must conform with complex physical artifacts — a chemical plant, railway network, etc.

Changeability. Software is seen as easy to change. Customers demand new functions, and change their minds even near the delivery date (if they are allowed to). Old software may require extensive updating to work with newly available hardware.

Invisibility. Software is invisible and impossible to visualize. Many diagrams can be produced — control flow, data flow, static structure — but none is obviously fundamental. This makes software hard to comprehend, and hard to maintain. And the development process is hard to monitor.

Slide 106

More Intrinsic Difficulties

- As hardware gets cheaper, software seems dearer.
- A long-term project may be chasing a moving target.
- Hardware is built from components, but software is mainly built from scratch.
- Software is easy to steal, so developers are forced to cut costs.

The first point explains why businesses now buy accounting software off the shelf. It is cheaper to buy an existing package than to build something in-house. That did not matter earlier, hardware was the dominant cost.

The second point is that even if the system's functions are frozen, its environment changes over time. Hardware is changing rapidly; it can be hard to predict what sort of platform will be available to run the software in a few years' time. Porting software to a new environment can be expensive and introduce errors.

Software components are scarce. Some exceptions are numerical libraries and parser generators (yacc). Software re-use is a key goal of researchers. Object-oriented programming (OOP) is often touted as a solution.

How many of you have *never* illegally copied software? Losses to the software industry are enormous, and this limits what they can spend on development.

Slide 107

Management Difficulties

- incomplete, inconsistent, changing requirements
- senior managers who believe in fairies
- staff who believe in fairies
- unrealistic deadlines
- ‘Adding staff to a late project makes it later.’ (Brooks)
- staff turnover

There is evidence to suggest that managers do not want to hear about problems. So staff are afraid to report problems, e.g. in the CONFIRM project. Richard P. Feynman, while investigating the Space Shuttle disaster, found that NASA’s senior management had completely unrealistic beliefs about the Shuttle’s reliability. They did not know about faults that had been reported repeatedly.

Extreme time pressure is common on many projects, e.g. the London Ambulance Service despatching system. Shortcuts are taken, tests omitted, and work done at 4am is likely to be done badly.

Adding staff causes further delays because jobs have to be re-allocated and existing staff have to spend time training the newcomers. It is not like hiring more bricklayers.

Because software is invisible, all knowledge about a program may reside in one programmer’s head. If he or she departs, other staff are forced to reverse engineer the program.

Lecture 6 will cover these issues in more detail.

Slide 108

Course Overview

1. software crisis
2. critical software
3. the software life cycle
4. formal methods
5. the Z specification language
6. very large systems

Read this overview as a partial list of approaches to Software Engineering. Other aspects of the subject mainly concern management.

Critical software is software that simply must not fail. What is reliability and to what extend can we achieve it?

The software life cycle consists of several key stages: requirements, design, implementation, testing, maintenance. The idea that projects go through these stages in strict succession is called the *waterfall* model of software development. This model is known to be simplistic.

Formal methods based on mathematics are being promoted as a means of attaining high reliability. This lecture is a survey.

The Z specification language is one of the most popular formal methods in the UK. This lecture is an introduction to it.

Very large systems present particular problems. This lecture discusses the outcome of a famous study [8].

Slide 109

Related Books

⋆ Carlo Ghezzi et al. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.

Roger S. Pressman. *Software Engineering*. McGraw-Hill, 1994.

Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

Lauren Ruth Wiener. *Digital Woes*. Addison-Wesley, 1993.

Ghezzi is the course text. It concisely covers the main issues. Sommerville and Pressman are also useful. They are broadly similar: massive volumes covering all aspects of software engineering.

Leveson's book is a serious treatment of system and software safety, aimed at developers and managers.

Wiener lists some major software failures and discusses the underlying social issues. She is a journalist (it occasionally shows) and aims for a general audience. Her book is well referenced.

Neumann's *Computer-Related Risks* [23] contains extensive listings of computer-related incidents concerning safety, security, privacy, etc. A final chapter tries to draw some conclusions on the control of risks.

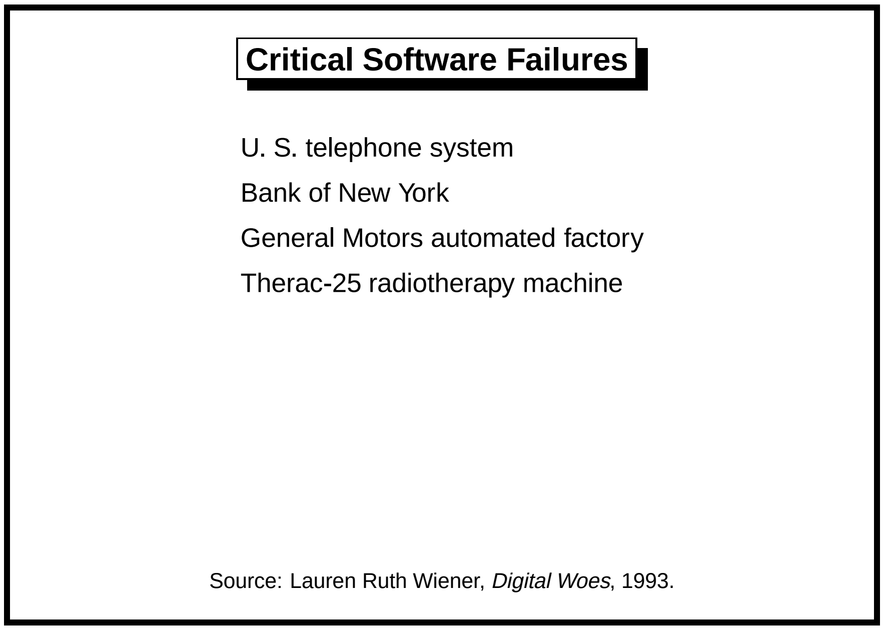
Good for laughs, and some insights, is

John Gall. *Systemantics: How Systems Work and Especially How They Fail*. Fontana, 1979.

'Software's Chronic Crisis' in the September 1994 *Scientific American*, gives a general overview of problems and proposed solutions.

Scary is Borning's article on nuclear war [2].

Slide 201



Critical Software Failures

U. S. telephone system

Bank of New York

General Motors automated factory

Therac-25 radiotherapy machine

Source: Lauren Ruth Wiener, *Digital Woes*, 1993.

Wiener [29] lists incidents where software failures have resulted in loss of life, huge amounts of property, reputations, etc. She notes that software products do not carry warranties, but rather warranty disclaimers.

Telephone networks in several major United States cities suffered failures in the summer of 1991. The problems were traced to telephone switching software consisting of several million lines of code. The potential losses are incalculable.

The Bank of New York's securities transaction software had a storage management fault. During ninety minutes on 20 November 1985 it lost information on $32 billion in transactions. The Bank was forced to borrow $23.6 billion from the U. S. Federal Reserve for a day, at a cost in interest of $5 million. Although the information was later recovered, the bank's reputation suffered.

The General Motors Hamtramck factory, in Detroit, had 50 automatic guided vehicles to ferry parts and 260 robots to weld and paint. But a year after its opening in 1985, the plant was still operating at half its capacity due to software problems. GM was trying to keep up with the Japanese, but Wiener suggests that they might have looked at worker training and motivation instead of building robots.

The Therac-25 killed two patients by administering a massive overdose of radiation. A physical interlock, present on previous models to prevent this occurrence, was omitted on the grounds that computers do not make mistakes. There were three other serious incidents.

It's not just software! A bug in the Pentium's floating-point unit is said to have cost Intel $306 million for replacement chips alone.

Slide 202

Critical Software

- *Safety-critical*: charged with protecting human life
  – cars & aircraft
  – chemical, nuclear plants
  – medical equipment
- *Mission-critical*: charged with an essential task
  – telephone routing
  – financial transactions
  – stock control
- *Security-critical*: charged with protecting confidential information

If critical software fails, the loss could be intolerable: people could die or a company could go out of business. The guidelines are not clear: is typesetting software mission critical?– yes for a publisher, maybe not for other enterprises. Is ambulance despatching safety-critical?– not all ambulance trips are for emergencies.

Safety-critical software is becoming more and more common. In cars it may control the brakes and throttle. Software in car engines (e.g. to control fuel injection) is perhaps not safety-critical.

Mission-critical software includes the software that runs a company's main business. It could also be taken to include software in household appliances. A modern television or video can contain megabytes of software. If this fails, the appliance cannot do its job properly. The manufacturer might be forced to make a hugely expensive recall.

The *safety* and *mission* aspects of software may come into conflict. Safety obviously must take precedence over property.

Security-critical software is increasingly important to banks and other financial institutions. There are also applications in the government and military. Note that software might be safe, and carry out its main functions, but (perhaps by using a faulty cryptographic protocol) allow an intruder to obtain confidential information.

Lauren Wiener [29] tells the story of a surgeon using a spreadsheet for vital calculations during an operation. Whether one believes this or not, the story shows how any software could be given a critical role. Most software is not built to the necessary standard of reliability!

Slide 203

‘Hard’ Features of Critical Software

Real-time: safety-critical systems must react promptly

Distributed: financial systems often link several sites

Concurrency: many sensors & controls

Environmental: must cope with hardware errors

Human users: must be easy to use correctly

Critical software, ironically, often has the very characteristics that make failures more likely: real-time constraints, concurrency, and harsh physical environments.

Most safety-critical systems have real-time constraints. The brakes must engage immediately the brake pedal is pressed. Fly-by-wire aircraft must also respond promptly to their controls. With mission-critical software less is at stake, but the task must be performed in a reasonable time.

Distributed systems and concurrency make debugging difficult because failures can seldom be reproduced. Environmental factors cannot affect software directly, but require hardware redundancy (multiple sensors and processors) that complicate the software. Such systems can be hard to test under realistic conditions.

There are many examples (not just in software) of designs that make human error more likely. System designers should resist the temptation to take over the user’s job. For example, ‘envelope protection’ on aircraft software prevents the pilot from making extreme maneuvers; the aim is to protect the aircraft, but it limits the pilot’s ability to cope with emergencies.

By contrast, spreadsheets and work processing packages operate on one processor. They interact only with a keyboard and screen in a safe environment. They can be tested by giving away ‘beta-releases’ to willing guinea-pigs. Wiener [29] notes that Microsoft became the world’s biggest software company by selling this kind of software, not critical software!

Slide 204

Dependable Systems

Accident: event that causes harm or damage

Hazard: a condition that can cause an accident

Risk: probability $\times$ cost (for a particular accident)

Safety: continuous delivery of hazard-free service

Reliability: continuous delivery of proper service

Availability: proportion of time having correct service

A *dependable* system is safe, reliable & available.

These definitions may look tiresome, but they make real distinctions. *Reliability v safety*. Hazard-free service does not have to be proper: a desired function could be disabled for safety reasons. Reliability measures mean time to failure. And *failure* is the absence of proper service.

A traffic light that shows green in all directions is a hazard. It is much worse than the light's going completely dead. But it is not itself an accident and will not necessarily cause an accident.

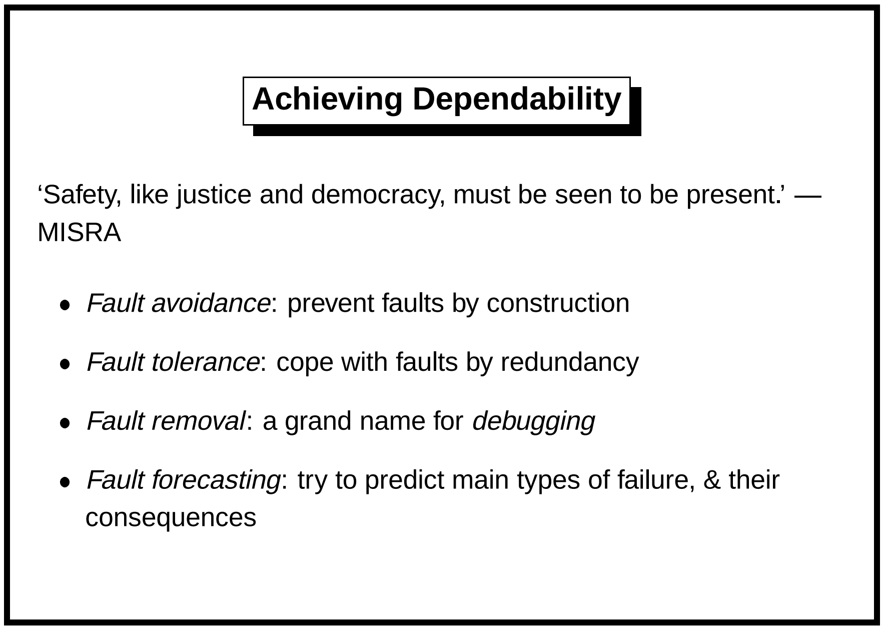
Correct service does not have to be safe: the specification could be wrong. If there are frequent, brief failures, we could have high availability together with low reliability. In a telephone network, this might be acceptable. See Sommerville [25, chapter 21], Bowen and Stavridou [3] or Littlewood and Strigini [21].

Risk is hard to calculate when the loss involves life and limb. The probability can also be hard to calculate. The calculation is hardest if the event has never occurred. What is the risk associated with nuclear war?

A *fail-safe* system is one whose design ensures that, upon failure, it enters a safe state. For instance, a lift car does not fall even if the cable snaps. Of course, things might not go according to plan. One of the 'laws of systemantics' [12] is

A fail-safe system fails by failing to fail safe.

Slide 205



Achieving Dependability

'Safety, like justice and democracy, must be seen to be present.' — MISRA

- *Fault avoidance*: prevent faults by construction
- *Fault tolerance*: cope with faults by redundancy
- *Fault removal*: a grand name for *debugging*
- *Fault forecasting*: try to predict main types of failure, & their consequences

How can safety be seen when software is invisible? We only get statistics after a system has been put into service, and it takes a long time before those statistics are scientifically meaningful. All we can observe is the *process* of producing the system. This process must be above reproach. The builders of a safety-critical system must employ best practice throughout, as described for example by Leveson [20].

Fault avoidance requires following good practice. '. . . Safety, like quality, should be built in rather than added on' [22]. Programmers should work from a precise, preferably formal, specification.

Strongly-typed programming languages eliminate a whole class of errors: type errors. The programming language should also support modularity — to allow internal data structures to be hidden from other parts of the program. The compiler must be reliable or it could introduce its own errors.

An example of *fault tolerance* is the space shuttle. It has five processors. Four run identical software, and 'vote' to ensure they are all in agreement; any mavericks are rebooted. This guards against hardware errors. You only need three processors; the fourth is a spare. The fifth runs different software, to take over if the other software fails [29]. On a smaller scale, exception handling can be used for fault tolerance.

Fault removal alone cannot achieve high reliability. Indeed, all four methods must be used.

Fault forecasting requires experience and imagination. It is impossible to imagine every possible way a system could fail.

Slide 206

Ultra-High Reliability

10^{-9} failures per hour = 100,000 years between failures

- *Debugging* suffers from diminishing returns.
  – major bugs found first
  – fixing minor bugs helps little
- *Testing* would take 10^9 hours!
  – fixing one bug may cause others
  – test conditions must be realistic

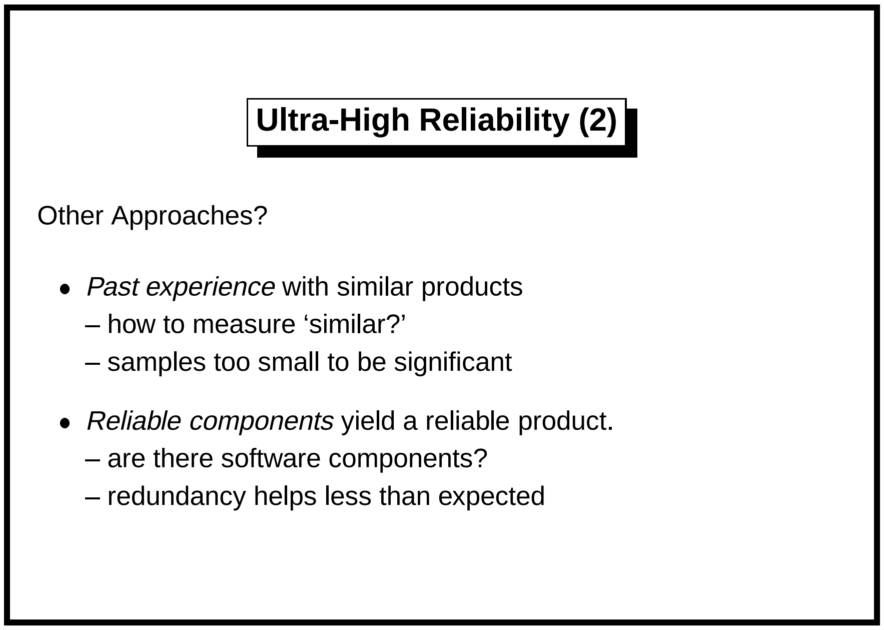
Source: Littlewood & Strigini, CACM, 1993

Littlewood and Strigini [21] say ‘The requirement for critical avionic systems of 10^{-9} failures per hour represents a probability of about 0.1 that at least one such failure will occur in a fleet of 1,000 aircraft over a 30-year lifetime. Since this safety-critical system is likely to be one of many on the aircraft, such a probability does not seem an unreasonable requirement.’ Can we meet it?

Diminishing returns: debugging does not greatly improve reliability after the main faults have been removed. In one system, 1/3 of faults occurred only once in 5,000 years. And correcting a fault runs the risk of introducing others!

Testing: by some fairly complicated statistical analysis, Littlewood and Strigini argue that to demonstrate that the median time to failure is T requires observing perfect working for time T. If T is large, we can only do so by observing many copies running in parallel. (Example: the telephone system contains many identical switches.) But the testing must be under realistic conditions. Can we test aircraft software like that?

Slide 207



Ultra-High Reliability (2)

Other Approaches?

- *Past experience* with similar products
  – how to measure 'similar?'
  – samples too small to be significant
- *Reliable components* yield a reliable product.
  – are there software components?
  – redundancy helps less than expected

Littlewood and Strigini consider other approaches to achieving dependability.

A design could be based on a previous, successful design. But does 'based on' mean the dependability carries over? Typically a successful design has been used only a few times, so the method gives us little information.

A product could be constructed from components known to be dependable. This method says nothing about design errors in joining the components. And the only way to know that components are dependable is by observing them over a long period of time. This is possible in hardware, where many standard components exist. But there are few software components.

Redundancy in software means having several different teams code the same functions. This has been shown to improve reliability, but the improvement is much less than would be expected if the failure behaviours were independent. This suggests that different teams make similar coding errors or fail to consider similar unlikely cases.

Slide 208

High Reliability & Formal Methods

- *Formal specification*: a **precise** set of requirements
  – defines a notion of *correctness*
  – invariably simpler than the real world
- *Correctness proof*
  – extremely expensive to produce
  – small possibility of errors in proofs
  – concerns the specification — not the real world

No way to meet ultra-high dependability targets

Formal methods can play an important role in building reliable systems, but they do not solve the problem.

A formal specification is valuable because it specifies the requirements precisely. But many existing programmers and managers cannot read the necessary mathematical notations. We must hope this will change over time, with improved training. A more fundamental limitation is that all formal specifications represent an approximation of the real world. The world is simply too detailed to admit a fully formal treatment.

Proofs could be wrong, even if they are produced with the help of tools. There is ultimately no way of ensuring that a proof is valid. The risk of a faulty proof is small, but we do not know how this will affect our requirement of ultra-high reliability.

Producing formal proofs requires many months of work by skilled specialists, even with the help of tools. The cost is dropping but is still prohibitive for most software projects.

We speak of a correctness proof. But all we can prove is that a design meets its specification. The proof therefore is only as good as the specification. Realistic testing, on the other hand, evaluates the system in the real world. Thus, formal proof will never eliminate testing.

Littlewood and Strigini conclude that no known method (or combination of methods) can attain a reliability of 10^{-9} failures per hour. The best we can hope for is about 10^{-5} failures per hour. Moreover we have no means of *recognizing* such reliability, even if we had it!

Slide 209

Safety Systems: Some Questions

Do we need it?

Can we do it?

What are we trying to do?

Do we need it? Lauren Wiener [29, Chapter 5] devotes an entire chapter to discussing dubious proposals for using computers. The Advanced Vehicle Control System has been proposed to take over driving specially-equipped cars. It would permit ‘platoons of cars, separated by only a few feet, to zoom along at 90 mph while their drivers read the newspaper.’ Could this ever meet the necessary standard of dependability? We already have a cheaper and safer version of this idea: a train.

Can we do it? Littlewood and Strigini suggest that software should never be used in a situation where ultra-high reliability is required. Human reliability is much worse than 10^{-5} failures per hour; software might still increase overall system dependability by reducing the risk of human error.

But Wiener [29, page 174] notes the increasing tendency of computer systems to remove choices from users. A school bus carrying sixty girls crashed; its brakes had failed while descending a hill. The bus was found to be in too high a gear for such a steep descent. Its electronic transmission would not let the driver change to a lower gear. It was designed to protect the engine from over-revving.

What are we trying to do? Wiener [29, page 14] describes a decision by the Washington State ferry system to abandon digital controls on its ferries, after several serious malfunctions. But why were digital controls installed in the first place? Weight is hardly as significant for a ship as it is for aircraft. Computers are often installed merely for the sake of ‘progress.’ No software project can succeed unless it has clear goals.

Slide 301

Stages of Software Development

1. *Requirements analysis & definition*
2. *System & software design*
3. *Implementation & unit testing*
4. *Integration & system testing*
5. *Operation & maintenance*

Source: Sommerville, *Software Engineering*, 1992

Requirements analysis and definition. What should it do? A vague mission statement is refined, by consultation with users and clients, into a precise specification of services and constraints.

System and software design. How will it work? Services are allocated to hardware and software modules. The result is the system architecture.

Implementation and unit testing. Program modules are coded and tested individually.

Integration and system testing. Program modules are combined. The entire system is tested under realistic conditions.

Operation and maintenance. The system is installed. Errors reported at this stage are corrected.

In the *waterfall* model of software development, a project goes through these stages in order. In practice this almost never occurs. Errors or omissions in the requirements are frequently discovered later, which requires returning to stage 1. More elaborate is Boehm's *spiral* model, which has a cyclic structure. Each cycle begins with an assessment of risk; it recognizes that key decisions (which might include cancellation) will be taken regularly.

Because software is invisible, management sometimes monitor progress through documents delivered at each stage: requirements specifications, architecture specifications, detailed designs, etc. Preparing and reviewing such documents involves a considerable effort, which can slow down the project and increase its cost. Artificial documents are sometimes produced.

Slide 302

Getting the Requirements Right

- Vague initial goals
- Iterative extraction & refinement
- Discussions with customer *& users*
- Rapid prototyping
- Precise requirements documents

Customers are typically not computer specialists, but are experts in their own business. Their initial goals will be vague, and frequently will contain inconsistencies. They may have unrealistic expectations of increased profits, etc. They may be unaware of what computers can and cannot do.

The first task is to obtain a precise list of requirements. It will be long, tedious, but essential. It will drive the entire development process. Mistakes and omissions, if not discovered in time, could require costly redesign. Some software vendors impose a penalty if the customer requests late changes.

The usual method of ‘capturing’ requirements is to hold a series of interviews. These should involve the software developer, the customer and the real users. The paying customer is seldom a user and frequently there is a clash of interests. For instance, the customer may be unwilling to pay for facilities that users want, or the customer may want the system to monitor the users’ performance.

Rapid prototyping means building a throw-away system that demonstrates the main functions. Customers and users can then see in action the consequences of their requests, and can change them. Prototypes can be built using animation packages, scripting languages, and other tools. For demonstrating a user interface, you can even use paper models! A prototype will probably be inefficient, unreliable, or hard to maintain. Neither the customer nor management must ever attempt to turn the prototype into a product.

Slide 303

Some Requirements Documents

- *Requirements definition*: statement of user services
  – understandable to management & users
- *Requirements specification*: detailed list of services
  – understandable to technical staff
- *Software specification*: abstract description of units
  – formal specification methods
  – towards a design

The document titles are largely taken from Sommerville [25]. It does not matter what they are called, but note their contents and intended readership.

The requirements definition is aimed at customer and developer management. It will be fairly brief, and clear.

The requirements specification is highly detailed and forms the basis of the contract between the customer and the developer. It is sometimes called a *functional specification*, but it should specify more than functions (e.g. efficiency constraints). It defines criteria for testing; start preparing a test plan now.

The software specification describes procedures and functions abstractly, perhaps using formal methods (mathematical notation). It may specify the components of the state, the inputs and outputs of each procedure, and a logical description of the relation between the inputs and the outputs. There is evidence [10] that using formal methods results in a deeper analysis of requirements, while modestly increasing the cost of the design phase.

An *executable* specification may involve describing the system as a functional or logic program, or using an executable subset of a specification language such as VDM. It is precise and has the advantage of yielding a prototype. It has the drawback that it is harder to produce and to read than an abstract specification, and may suggest a poor implementation.

A *draft user manual* might be given to customers and users to give them an impression of the system. This may reveal problems with the requirements.

Slide 304

The Requirements Specification

- *System model*: describes system components & their environment
- *Functional requirements*: system services
- *Non-functional requirements*:
  – constraints on software
  – restrictions on the design
- *Hardware*: define typical configurations

Sommerville [25] suggests a structure of informal chapters augmented by detailed appendices, which make up the bulk of the document.

It is essential that the system and its functions should be properly explained in terms of customer objectives. This guides the developers and reassures the customers that they will get something worthwhile. An introduction should briefly describe the need for the system. Each requirement should include an individual justification, unless this is obvious.

This document should not describe how the implementation will be done (except for particular constraints). It should be complete in that it should describe, for example, the treatment of illegal inputs. But if several responses are equally valid, it should not specify a particular one merely to be definite — this is over-specification.

Typical constraints include character sets or protocols to be used, maximum allowed response times, and programming languages. Critical systems will have reliability constraints. The cost of both purchasing and using the system will be constrained. Sommerville has a comprehensive chart on page 92.

Simplicity is the strongest tool for making a project tractable. Needless complexity must be avoided. This requires discipline and enough knowledge to tell the difference between important and trivial requests. Always remember KISS (Keep It Simple Stupid).

Slide 305

Elements of Top-Down Design

- *Architectural design.* Identify the building blocks
- *Abstract specification.* Describe their functions and constraints
- *Interface design.* Precisely define how they fit together
- *Component design.* Recursively design each block

Source: Sommerville, *Software Engineering*, 1992

Top-down design involves repeatedly breaking down the problem into smaller components, until their implementation becomes obvious. Interfaces must be precisely defined so that each component can be designed and coded separately. Modifying or replacing a component should not affect the rest of the system provided the component still meets its specification. Other components must not depend on a component's internal variables or data representation. Such *information hiding* requires programming language support: modules, packages, etc.

Many design methods have been promoted. Computer-Assisted Software Engineering (CASE) tools support some of them. They are typically based on graphical notations. The *data flow* approach tracks data as it moves through the system. The *entity-relation* approach is concerned with data and the relations between them, in the tradition of relational databases. *Structured design* is concerned with the system components and their interactions.

Components should be coherent, consisting of related functions. Different notions of 'related' might yield different partitions of functions into components. You could for instance group all input/output services into a single component ('logical association'). However it is usually better to organise the components around different tasks to be performed ('functional cohesion'). This leads to *object-oriented design*.

Traditional flowcharts have long been regarded as useless, except perhaps for assembly-language programming.

Slide 306

Modular Design Strategies

- *Functional design.* Partition system functions
  – the traditional approach
  – a central state?
- *Object-oriented design.* Partition tasks into classes
  – objects combine state and functions
  – *inheritance*: new objects from old
  – re-usable components?!

The two main design strategies are *functional* (or *procedural*) design and *object-oriented* design. Functional design partitions the system in terms of functions operating on a central state.

Languages such as Modula-2 allow the state to be decentralised. State variables are associated with the corresponding functions to form a module. Only functions declared to be visible (or exported) have access to the state. For instance, if the module implements a dictionary that exports *lookup* and *update* functions, then other modules can call those functions but cannot manipulate the data structures directly.

Object-oriented design takes this idea further. A collection of state variables and functions defines an *object class*. New classes can *inherit* such attributes from parent classes and add new attributes. A simple class can serve as the parent (*superclass*) of many specialised classes (*subclasses*). Thus, we have a sort of **re-usable part**, thought by many to be the key to tackling the software crisis.

Modularity is essential for partitioning a large project among several programmers. Object-oriented design is exciting much interest. However, inheritance can act against modularity. You cannot understand a class unless you understand all of its superclasses; changing a superclass could affect a subclass. *Multiple inheritance* means allowing a class to have more than one superclass; it offers additional power and complexity.

Object-oriented approaches have a natural feel. However, they are poorly understood in theory and indeed in practice.

Slide 307

Implementation Techniques

- ready-made tools: libraries, scripting languages, . . .
- high-level languages
- independently-specified modules
- simple control & data structures
- up-to-date commenting

Code must be correct, clear, maintainable, portable, efficient

Proper tools reduce the need for programming. User interfaces can be built using the TCL/Tk scripting language. Parsers can be built using yacc. Any programming should be done in a high-level language.

Strong type checking is now regarded as essential, since it allows the compiler to catch many trivial errors. The C language has been tightened up; the ANSI standard now makes C strongly typed.

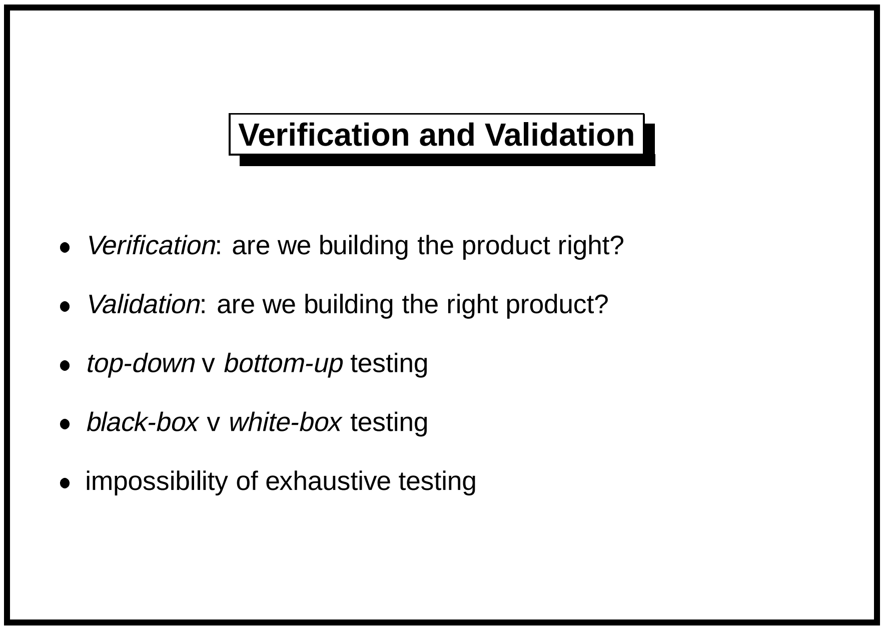
Modularity is essential too. Modula-2 and Ada provide modules, as do all object-oriented languages (C++ and Modula-3). Modules allow a top-down approach. All module (or object) specifications are written first. The compiler can check them for compatibility. Finally, programmers start to code against these machine-checked specifications.

Code should be as clear and easy to maintain as possible. These goals require a simple programming style and correct commenting. Efficiency is less important, provided it meets requirements constraints. Portability is not always required and can limit efficiency.

Assembly language is almost never used. It is neither clear nor portable; programmer productivity is very low. Two arguments in favour of it are speed and access to low-level functions. Both of these requirements can usually be isolated to small parts of the program, which might indeed be coded in assembly language. Speed is generally limited by a few bottleneck functions, which can be discovered using *profiling tools*.

Assembly language has been used in safety-critical systems to avoid compiler errors. It is now recognised that the risk of programmer error is greater.

Slide 308



Verification and Validation

- *Verification*: are we building the product right?
- *Validation*: are we building the right product?
- *top-down* v *bottom-up* testing
- *black-box* v *white-box* testing
- impossibility of exhaustive testing

Verification checks the product against its specification. It could in principle be done by static analysis, formal proof, etc., without ever running the product. *Validation* checks that the product meets the users' real needs, which may differ from the specification. This can only be done by testing.

Sommerville [25] lists several levels of testing, including unit, module, sub-system and system testing. Programmers may test their own code. An independent team tests larger components. Finally, *acceptance testing* involves the customer, who formally approves (or rejects!) the product.

Testing should proceed concurrently with implementation so that errors are found as quickly as possible. *Top-down* testing involves building top-level components first and writing trivial 'stubs' for the lower-level functions. Brooks [5] advocates this because you always have a working system. *Bottom-up* testing involves building the lower-level components first (as advocated by Knuth) and testing them using a 'test harness.' Both methods require writing artificial code to simulate the unfinished parts of the system.

Black box testing checks a component against its specification, exercising all external functions. *White-box* testing examines the component's internal structure, exercising every line of code and every decision point. It is good for testing the treatment of unusual cases. Boundary conditions are a common source of errors.

Beta-testing involves giving out copies of an uncertified product to people willing to report errors. This clearly is not possible for critical software.

Testing is expensive, consuming up to 50% of development costs. It can locate defects but never demonstrate correctness. Because digital behaviour is discrete, we cannot interpolate between a few observations. Exhaustive testing is impossible: the number of cases is astronomical.

Slide 401

What are Formal Methods?

- *not* ‘structured methods’
- Formal specification
- Refinement to code
- Formal correctness proofs
- Rigorous code analysis
- Tool support

Formal methods are grounded in mathematics. A formal specification eliminates ambiguity, giving a precise notion of correctness. Hinchey and Bowen [16] have compiled a survey of recent applications.

Formal methods are sometimes taken to include graphical methods such as dataflow analysis. But unless they are fully precise, they cannot be regarded as formal. Most CASE tools support graphical methods. Formal methods also benefit from tools: to help users write syntactically correct specifications, to run simple semantic checks on them, and to help in the refinement of specifications into code.

Formally correct code can be produced in two ways. *Program derivation* or *synthesis* involves transforming a specification into code by steps guaranteed to preserve correctness. The programmer supplies the transformations (we do not know how to automate this!); at every stage, the machine checks that the code is compatible with the specification.

Alternatively, the programmer could write the entire code and submit it for proof as a separate step. This is often called *program verification*, but note that *verification* is also used in the context of testing. Proving correctness requires a lot of time and skill; for most projects, it is too expensive. Unless the program was coded with verification in mind — avoiding low-level tricks — it may be practically impossible to prove correct.

Code can be analysed systematically without constructing a completely formal proof. This was used to certify nuclear reactor shutdown software; see below. Real software projects seldom involve formal proofs. The main use of formal methods is in writing formal specifications.

Testing also requires correctness to be defined precisely. But testing encompasses other things, such as customer satisfaction, that lie outside the scope of formal methods.

Slide 402

What are Specifications For?

- deeper analysis of requirements
- detecting inconsistencies
- specify *what* not *how*
- communication with implementors
- communication with testing team

A formal specification is essential if you are going to prove correctness, or to support transformation into correct code. Less ambitiously, formal proof can be used to derive properties from a specification; this could reveal inconsistencies early. The specification is also useful in itself. Studies have shown that attempting to write a formal specification stimulates deeper thinking about the requirements, showing up ambiguities hidden in English.

The ConForm Project [10] is investigating the costs and benefits of using formal methods in building a small security-critical system. Two teams are independently developing a so-called trusted gateway. One team is using fairly conventional structured methods; the other augments these methods by writing a formal specification (in VDM). The project is monitoring the development process, comparing the effort required to complete each phase, the quality of the documents produced, etc.

Early in the project they noticed the team using formal methods asked many more questions concerned with clarifying the requirements. The job of the trusted gateway is to take a stream of messages and forward each message either to a 'secret' or 'non-secret' output port; the decision is based upon certain keywords that may appear in messages.

Messages are limited to 10K. The formal methods team asked whether this limit included the message delimiters (it did). If a message contains both 'secret' and 'non-secret' keywords then it is regarded as secret. However, the formal methods team noticed the possibility that a 'non-secret' keyword could contain a 'secret' keyword as a substring. The developers had to go back to the customers to find out that such occurrences of 'secret' keywords should be ignored.

These are perfect examples of ambiguities that lurk in English descriptions, and that could lead to obscure errors. How many messages will be under 10K if delimiters are ignored, and over 10K if they are counted? The precision of a formal specification will help the implementors build a correct system, particularly if they have tool support. And the specification will help the testing team identify awkward cases to cover in test data.

It's not a bug, it's a feature! — formal specifications can help put an end to this (though it is partly a problem of requirements).

Slide 403

What is a Specification Language?

- *precisely* defined syntax and semantics (meaning)
- *executable* specifications: functional or logic program, . . .
  - rapid prototype
  - implementation bias
- specification languages for *sequential* programs:
  – Z, VDM, Larch, . . .
- specification languages for *concurrent* systems:
  – LOTOS, Unity, CRL, TLA, . . .

There are many specification languages, with different purposes. All have a precise definitions of their syntax and semantics. A given piece of text is either legal or not; if legal, it has a precise meaning. However, the meaning does *not* determine the implementation uniquely; rather it defines precise grounds for judging whether an implementation is correct.

A program counts as a specification. Programming languages are precisely defined (or should be), both their syntax and semantics. *Executable* specifications consist of programs written in very high-level languages paying no attention to efficiency [28]. They are precise, and (compared with a real implementation) they are easy to write, read and reason about. They also yield an executable prototype. They have many drawbacks, though. They may be too inefficient to serve even as prototypes. Making them executable will introduce implementation bias; they will not be abstract enough. They will map every input to a unique output, when normally for each input there is a set of legal outputs.

Consider a sorting program: its output should be an ordered permutation of its input. It is easier to say that than to write even a highly inefficient functional sorting program. Consider a compiler: its output is a string of machine instructions. If we specify the output uniquely, we shall not be allowed to include optimisations.

The meaning of a specification is defined in terms of mathematical abstractions. Early work concentrated on specifying data types, such as lists, stacks, queues and symbol tables; such work (e.g. Larch) was based on the theory of algebras.

Most modern specification languages treat computation as a whole, though still abstractly. A *sequential* program can be regarded as a function from inputs to outputs, or more generally as a relation between inputs and acceptable outputs. Z and VDM specify programs by modeling their data structures using elementary set theory.

A *concurrent* program is normally viewed as a system of communicating agents. This requires an abstract notion of agent behaviour, based upon something like a process algebra. Temporal logic is usually involved, for making statements about time dependencies: *A* and *B* cannot happen simultaneously; if *A* happens then *B* must happen eventually, etc.

Slide 404

Seven Myths of Formal Methods

1. *Formal methods guarantee perfection.*
2. *They work by proving correctness.*
3. *They are only good for critical systems.*
4. *They involve complex mathematics.*
5. *They increase costs.*
6. *They are incomprehensible to clients.*
7. *Nobody uses them for real projects.*

This landmark paper [15] by Anthony Hall of Praxis Systems is based upon industrial usage of formal methods. Here is a summary of how he refutes each myth.

1. All human methods are fallible. In particular, the specification could be an inadequate model of the real world. Errors can also occur in machine-checked proofs. The proving program could itself be faulty. Using it to prove itself (‘verifying the verifier’) does not solve the problem; as an extreme case, suppose it regarded all programs as correct?

But formal specifications do help find errors, because they eliminate arguments about what the specification actually says.

2. This myth reflects the US emphasis. European work is more oriented towards specification.

3. Praxis uses formal methods merely to help ensure high quality, even for non-critical software.

4. Formal methods are in fact based on (the easier parts of) discrete mathematics: set theory and logic. Staff training only takes about three weeks. Compare with the complexity of programming languages and client applications! But correctness proofs require more complex mathematics.

5. Development may be *cheaper* with formal methods. However, the requirements phase may take longer and cost more. It takes time to write any specification at all. The initial specification can usually be simplified as the problem is better understood. Time spent here is repaid during the implementation and maintenance phases.

6. You can paraphrase the specification in natural language and use it to derive consequences of the requirements.

7. Hall describes applications by IBM, Tektronix, Rolls-Royce as well as his own firm. Since his article was published, many other industrial uses have been reported — see below.

Slide 405

Experience with Formal Methods

- **SSADM tool set**, by Praxis Systems. 37,000 lines of code
- **CICS transaction system**, by IBM Hursley. 50,000 lines
- **Oscilloscope software**, by Tektronix.
- **Cobol/SF** by IBM Federal Systems. 800,000 lines
- **Air Traffic Collision-Avoidance System**, by FAA.
- **Multinet Gateway**, by Ford Aerospace. 6,000 lines

A major study by Susan Gerhart and others [13] investigated 12 cases involving the use of formal methods. These included five commercial projects, three exploratory uses and four projects involving critical software. In those last four, government agencies required the use of formal methods. Two of them (the Darlington nuclear power plant and the Paris Metro signalling system) are discussed in separate slides below.

The SSADM design tool built by Praxis inspired Hall's paper [15]. It involved 450 staff-weeks of effort, two devoted to writing the Z specification.

IBM's Customer Information Control system is large, 800,000 lines of code. IBM is now using the Z specification language to re-engineer this system; the 50,000 lines quoted above were developed in this way.

Tektronix used the Z specification language to help design the software in oscilloscopes.

Cobol/SF is a tool for tidying up old Cobol programs while preserving their meaning. IBM built it using the Cleanroom methodology, which is based upon (informal) proof.

The US Federal Aviation Authority (FAA) hired Nancy Leveson to apply formal methods to subsystems of TCAS (Traffic Alert and Collision Avoidance System) because they were worried about the 'loss of intellectual control over the specification.' She applied a graphical formal method (a variant of Statecharts).

The Multinet Gateway delivers messages to Internet hosts, while protecting confidential information. It was developed using the Gypsy Verification Environment.

Some of the projects reported by Gerhart started in the early 1980s, using methods now obsolete. Some used archaic tools or no tools at all. A tiny but growing number of software development projects use formal methods.

Slide 406

Darlington Nuclear Power Station

- two independent shutdown systems
- 26,000 lines of code (including assembler!)
- formal methods used to certify *existing* code
  – formalise requirements as specification tables
  – analyse code as program-function tables
  – compare the tables
- No tool support
- cost $2-4 million Canadian

This nuclear power station is roughly 40 miles from Toronto, Canada. Lauren Wiener's account of the project [29] is quite different in tone from Craigen et al.'s [7].

Emergency shutdown systems are normally controlled using 'switches and relays and analogue meters' [29] . The Darlington nuclear power station, unusually, built its emergency shutdown systems in software. There were 6,000 lines of assembly, 7,000 lines of Fortran and 13,000 lines of Pascal among the two systems. The Canadian authorities refused to licence the plant after problems were found in the software.

A formal code inspection was organised by David Parnas using the SCR method (Software Cost Reduction). Each process was analysed by three independent teams. One used the informal requirements document to generate a specification table. The second examined the existing code and generated program-function tables. The third examined the two sets of tables and reported discrepancies. The work was tedious and labour-intensive. They effected a hundred or so minor changes to the system, but found no serious errors.

A remarkable feature of this work was that it dealt with existing code, including assembly language. It involved rigorous analysis but not formal proof.

Wiener [29] claims that certifying the software delayed the plant's opening by six months, at a cost of $20 million per month in lost production (Canadian dollars). The software verification cost $2-4 million. A hardware shutdown system costing $1 million would therefore have been much cheaper. That is an argument against using software in nuclear power stations. It is no argument against formal methods, without which the software might not have been approved at all. One has to ask what safety criteria are used to certify traditional control systems?

Slide 407

Paris Metro Signalling

- reduce train separation from 2:30 to 2 minutes
- by GEC Alsthom. 9,000 lines of verified code
- 4-stage validation process
  – requirements validation
  – testing
  – safety/hazard studies
  – certification
- Hoare logic, for proving correctness
- B method, for refinement

The Paris Metro's new signalling system allows trains to run two minutes apart, a savings of 30 seconds. The increased capacity has eliminated the need for another railway line. The project was funded in 1982, a prototype was finished in 1985 and the system was deployed in 1989. Initially the developers used Hoare logic for correctness proofs, as the best available technique in 1982. Hoare logics are the basis for most approaches to proving correctness of software, but they can be complicated to use. The developers were unsure how to apply them on such a large scale. Jean-Raymond Abrial (one of the developers of Z) helped them to re-specify and re-verify the software.

Validation was divided into four stages: *validation of requirements, verification and testing, operations and maintenance,* and *certification*. They used other tools such as SADT (Structured Analysis and Design Technique) and performed hazard studies using fault-tree analysis. They used extensive testing, finding many problems with the specification. Testing is the only way to find out whether a program meets its real-world requirements; a correctness proof can only show that a program meets its specification.

Hoare logic [17] concerns statements of the form $\{P\}S\{Q\}$, meaning 'if P holds beforehand, and if execution of S terminates, then Q will hold afterwards.' In its pure form it says nothing at all if S fails to terminate, but it can be augmented to prove termination as well. It is not a specification language but a method for proving properties of code.

The B method models a process as an abstract machine. One abstract machine can be implemented by means of another. This accounts for the different levels of abstraction found in computer systems (machine language, operating systems functions, library functions, modules, subsystems, etc.). It supports development by top-down refinement, where an abstract machine is implemented in terms of increasingly lower-level machines.

Hoare logic dates from 1969, while the B method is still under development.

GEC Alsthom, the developer, is now using the approach for other railway products. One is a safety system covering all electrified lines in the French railways.

Slide 408

Research into Formal Correctness

- protocol verification
- hardware verification
- model checking
- system verification
- program design calculi

Protocols are used in consumer electronics (e.g. remote controls) and telecommunications. They are a common source of errors, since they are usually designed to work in the presence of unreliable media. Verifying a protocol is easier than verifying the software itself. Proofs depend on a model of unreliability; we assume, for example, that a network may re-order or lose messages, but not corrupt them.

Cryptographic protocols are used in security-critical systems, for example to deliver encryption keys. The so-called BAN-logic [6] is a formal method of proving that a protocol contains sufficient information for the agents to know they have received a fresh key.

Hardware verification is well advanced. The most successful method, based on higher-order logic, is to M. J. C. Gordon here at Cambridge. Correctness properties have been proved for many real chips.

Model checking is complementary to formal proof; it works for finite-state systems. It simply consists of enumerating all possible states and checking the desired property. The latest technique, *symbolic* model checking, is claimed to handle 10^9 states or more. Current research is investigating ways to prove properties of infinite-state systems by viewing them as finite-state systems.

System verification involves proving the correctness of subsystems, and of their integration, so that the whole system is proved correct. Bevier et at. [1] describe the proof of a 'stack' of components ranging from a simple high-level language to a microprocessor design. The aim is to have a computer system that is entirely free of logical errors, and that can only fail due to environmental conditions. (Note that for real-world applications, environmental conditions will remain a significant cause of failures.)

Program design calculi provide a precise way of constructing code to meet a formal specification. Many calculi are under investigation. Some use functional programming languages, which are particularly easy to reason about. Other methods apply to the usual (imperative) sort of language, although real languages like C are difficult to handle. A popular line of research involves deriving programs from suitably constructive proofs.

Slide 501

The Z Specification Language

Schemas used to define

- the legal state space
- operations that change the state
- operations that inspect the state
- special cases of an operation

Incremental development of a specification

Data described using set theory

This lecture is based on Spivey [27]. It presents his trivial example, the *Birthday Book*, a system that can record people's birthdays and issue a reminder for them.

Schemas are peculiar to Z. They are a bit like record operations: they describe a collection of named fields, specifying relations that hold among them and actions involving them. You can define a schema for each operation. But an operation can, in fact, be defined in terms of several schemas: one schema for the normal case, and other schemas for various exceptional cases. Schemas can be introduced one at a time.

Another popular specification language is VDM (the Vienna Development Method). VDM is unusual for its use of a three-valued logic, as a way of reasoning about definedness (particularly, termination). VDM includes methods to help refine the specification into code.

Z was developed at Oxford University by Jean-Raymond Abrial, Bernard Sufrin, Carroll Morgan and others. VDM was developed at the IBM Laboratory in Vienna by Cliff Jones, Dines Bjørner, Peter Lucas and others. The two languages look quite different, but in most essential respects they are the same.

One key difference is the treatment of an operation's *precondition*: a property that must hold before the operation may be invoked. In VDM, you specify the precondition directly. In Z, if an operation is built out of several schemas, the precondition is specified in bits and pieces.

Both languages use basic concepts from set theory to describe data and operations. This is called the *model-oriented* approach; such a specification is a bit like an implementation in set theory (so, of course, it is not executable). So-called *property-oriented* specification languages involve stating the desired properties of a module without exhibiting a mathematical model for it.

Slide 502

Some Z Notation

$\mathbb{P}X$ is the set of subsets of X

$x \in A$ means x is an element of A (and $x \notin A$ is its negation)

$A \subseteq B$ means A is a subset of B

$A \cup B$ is the union of A and B

$f : A \nrightarrow B$ means f is a *partial* function from A to B

$\mathrm{dom} f$ is the domain of f

$f \cup \{x \mapsto y\}$ extends f to map x to y

$f : A \to B$ means f is a *total* function from A to B: it maps *all* elements of A to elements of B. It is not used below, but is the natural way of specifying arithmetic operations, for instance.

$f : A \nrightarrow B$ is used below to represent a table. We specify a *partial* function as we do not expect a table to contain an output for every conceivable input.

$\mathrm{dom} f$ is not interesting for total functions; if $f : A \to B$ then $\mathrm{dom} f = A$. But if f is a partial function, then $x \in \mathrm{dom} f$ if and only if $f(x)$ is defined.

$f \cup \{x \mapsto y\}$ is the function that agrees with f except that its domain is enlarged to map x to y. Here $\{x \mapsto y\}$ is a trivial function whose domain is $\{x\}$. Since a function is a set of pairs, $\{x \mapsto y\}$ is simply a nicer syntax for the ordered pair of x and y. Also $f \cup g$ combines the functions f and g, but the result will not be a function unless f and g agree where their domains intersect.

More generally, $f \oplus g$ combines f and g, with g overriding f where their domains intersect. So $f \oplus g$ will always be a function provided f and g are. The function $f \oplus \{x \mapsto y\}$ is a version of f *modified* to map x to y. It can be used to modify any function (partial or total), or to extend a partial function's domain.

This sort of abstract notation allows us to express data without concern for the implementation. A partial function could be implemented as an array, a list, a tree, a B-tree on disc, etc.; such decisions are taken later in the design stage.

Z includes many more symbols: for sequences, Cartesian products, tuples, etc. In addition, there are all the logical symbols: and, or, not, implies, etc. Unfortunately, VDM frequently uses different symbols for the same concepts. Both languages often differ from standard mathematical usage.

Slide 503

Defining the State Space

$$\begin{array}{|l}\hline \textit{BirthdayBook} \\ known : \mathbb{P}\, NAME \\ birthday : NAME \nrightarrow DATE \\ \hline known = \operatorname{dom} birthday \\ \hline \end{array}$$

State variables

- *known*: a set of *NAME*s
- *birthday*: a partial map from *NAME*s to *DATE*s

Invariant: $known = \operatorname{dom} birthday$

Our description is very abstract. We have not specified anything about the structure of a *NAME* or *DATE*. We have placed no limit on the number of names stored. Such points can be specified later. But since *birthday* is a function, we have specified that a name can be assigned at most one birthday.

A state space has two key features. The *state variables* are the components that make up the state. The *invariant* is the relation that must hold of the components. For the birthday book, the state has two components, *known* and *birthday*, where *known* is entirely determined by *birthday*.

A more realistic system would have a more complicated relationship among its components. We could add a new component, mapping names to addresses say, with the restriction that you can only record an address if you also record the same person's birthday.

$$\begin{array}{|l}\hline \textit{BirthdayAndAddressBook} \\ known : \mathbb{P}\, NAME \\ birthday : NAME \nrightarrow DATE \\ address : NAME \nrightarrow ADDRESS \\ \hline known = \operatorname{dom} birthday \land \operatorname{dom} address \subseteq known \\ \hline \end{array}$$

We could have expressed this schema by combining *BirthdayBook* with a small schema specifying *address*. It is hardly worth the trouble here, but for larger specifications the ability to combine schemas is invaluable.

Every operation on the state must *preserve the invariant*: it may assume that the invariant holds at the start, and must ensure that it holds at the finish. The concept of invariant is not specific to Z, but is fundamental to Computer Science. The ConForm Project [10] found that specifying the invariant helped the designers identify pathological cases.

Slide 504

A State-Changing Operation

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

$name? \notin known$

$birthday' = birthday \cup \{name? \mapsto date?\}$

Precondition: $name? \notin known$

Operation: $birthday' = birthday \cup \{name? \mapsto date?\}$

Invariant: implicitly present

AddBirthday adds *name*? to the state, assigning to it the birthday *date*?. Since this operation changes the state, we specify it using a Δ schema that includes *BirthdayBook*. The schema contains two copies of *BirthdayBook*'s state. The variables *known* and *birthday* represent the initial values, while the primed variables $known'$ and $birthday'$ represent the final values.

Variables ending with a question mark, such as *name*? and *date*?, represent the operation's inputs. Output variables end with an exclamation mark; this schema has none, but see below. An equation such as

$$birthday' = birthday \cup \{name? \mapsto date?\},$$

looks like an assignment statement, but actually it *defines* a final value in terms of initial values and inputs. The equation specifies that the *birthday* function will be extended to map *name*? to *date*?. The relation between initial and final states does not have to be given by equations, especially if the input state does not constrain the final state uniquely.

The schema *AddBirthday* is subject to the precondition $name? \notin known$: the name must not already have a birthday assigned. Otherwise $birthday'$ might assign two different birthdays to *name*?; it would no longer be a function! A schema specifies an operation *provided* the precondition holds.

The invariants are added implicitly: $known = \mathrm{dom}\, birthday$ is part of the precondition, while $known' = \mathrm{dom}\, birthday'$ is part of the effect. The latter equation allows us to derive an explicit value for $known'$:

$$\begin{aligned} known' &= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) \\ &= \mathrm{dom}\, birthday \cup \mathrm{dom}\{name? \mapsto date?\} \\ &= \mathrm{dom}\, birthday \cup \{name?\} \end{aligned}$$

Using the invariants, we obtain $known' = known \cup \{name?\}$. We have also used basic properties of domains, $\mathrm{dom}(f \cup g) = \mathrm{dom}\, f \cup \mathrm{dom}\, g$ and $\mathrm{dom}\{x \mapsto y\} = \{x\}$.

Slide 505

A State-Inspecting Operation

$$\begin{array}{|l}
\hline \textit{FindBirthday} \\
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}! : \textit{DATE} \\
\hline
\textit{name}? \in \textit{known} \\
\textit{date}! = \textit{birthday}(\textit{name}?) \\
\hline
\end{array}$$

Precondition: $\textit{name}? \in \textit{known}$

Operation: $\textit{date}! = \textit{birthday}(\textit{name}?)$

No effect on state — instead, yields an *output*

FindBirthday looks up *name*? in the state, returning the associated birthday as *date*!. Since this operation never changes the state, we specify it using a Ξ schema that includes *BirthdayBook*. Strangely enough, this schema also contains two copies of *BirthdayBook*'s state, just as a Δ schema would. But it also contains implicit constraints that the state cannot change: $\textit{known}' = \textit{known}$ and $\textit{birthday}' = \textit{birthday}$. This means that Δ and Ξ schemas have the same internal structure, allowing them to be combined easily.

The equation

$$\textit{date}! = \textit{birthday}(\textit{name}?)$$

defines the output variable *date*! in terms of the input variable *name*? and the state variable *birthday*.

The schema *FindBirthday* is subject to the precondition $\textit{name}? \in \textit{known}$: the name must have a birthday assigned. If it does not, $\textit{birthday}(\textit{name}?)$ is undefined. Several schemas for one operation, specifying different preconditions, can be combined to yield a more general operation; we can specify error situations separately.

Slide 506

Two More Schemas

```
Remind
  ΞBirthdayBook
  today? : DATE
  cards! : ℙ NAME
  ───────────
  cards! = { n : known | birthday(n) = today? }
```

```
InitBirthdayBook
  BirthdayBook
  ───────────
  known = ∅
```

Remind is a sort of inverse to *FindBirthday*: it looks up the date *today*? in the state, returning the associated names as the set *cards*!. This set is specified to consist of all names in *known* whose birthday equals *today*?. We are not constrained to find the set of names by searching, as the formula may suggest; any implementation technique, such as hashing, is acceptable. (The variable is called *cards*! because it will hold the names of people you must send cards to.)

InitBirthdayBook is a schema to specify the initial state for *BirthdayBook*. This is an example of extending an existing schema with additional constraints, here $known = \varnothing$. Writing it in this way is more concise than writing out the *BirthdayBook* schema and including the additional equation.

The invariant, $known = \operatorname{dom} birthday$, is still present. Since *InitBirthdayBook* specifies $known = \varnothing$ we obtain $\operatorname{dom} birthday = \varnothing$. Therefore $birthday = \varnothing$; initially, no birthdays are recorded. (The empty set, $\varnothing$, is also the empty function.)

Slide 507

Specifying Exceptional Cases

Success

$$\begin{array}{|l}
result! : REPORT \\
\hline
result! = ok
\end{array}$$

AlreadyKnown

$$\begin{array}{|l}
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already\_known
\end{array}$$

We shall deal with exceptional situations by augmenting each operation to return a status report. The report can be *ok* or an error value such as *already_known*.

The trivial schema *Success* simply returns a report indicating success. It is useless by itself. But we can express a schema that combines *AddBirthday* with a success report by the conjunction $AddBirthday \wedge Success$. This denotes the schema whose state variables are those of the two schemas combined, and whose logical specifications are joined using $\wedge$. The new schema does everything that *AddBirthday* does, and also reports $result! = ok$.

The schema *AlreadyKnown* handles the case of attempting to add a birthday for a name already present. Its precondition, $name? \in known$, is the negation of *AddBirthday*'s. We use a Ξ schema to specify that the state does not change; instead, the output variable $result!$ receives the value *already_known*. We may interpret this as an error condition; Z (unlike VDM) has no built-in notion of exception.

A *robust* operation to add birthdays, which handles the error condition, can be defined to be a combination of the schemas presented above:

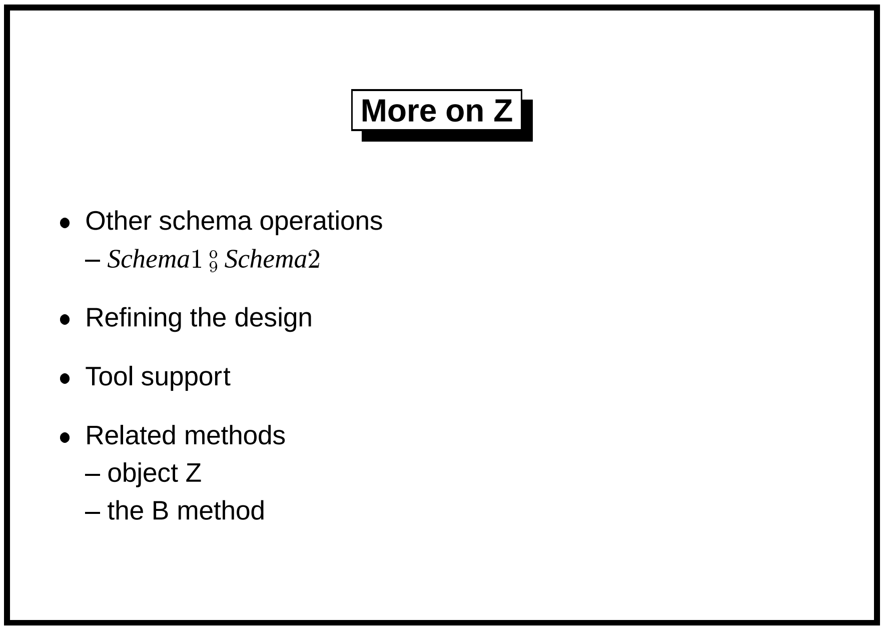
$$RobustAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$$

If $name? \in known$ then the specified effect is $result! = already\_known$; otherwise it adds the birthday and yields $result! = ok$. Specifying an operation in pieces, as here, has many advantages over writing one huge specification that covers all error conditions. It is easier to read, easier to write, easier to extend and modify.

Spivey [27] goes on to define *RobustFindBirthday* in precisely the same manner. Finally he defines $RobustRemind \mathrel{\widehat{=}} Remind \wedge Success$; since *Remind* has no precondition, all we must do is make it report success.

One problem with Z is understanding what a schema really means. At first, schemas were regarded as shorthand for long formulæ. Later it was decided that schemas required some kind of a formal semantics, and this has taken many years to get right. Intuitively, a schema abbreviates a formula of the form *precondition* implies *effects*, where *effects* contains all specified constraints on the final state and output variables.

Slide 508



More on Z

- Other schema operations
  – $Schema1 \fatsemi Schema2$
- Refining the design
- Tool support
- Related methods
  – object Z
  – the B method

Z contains many other means of building new schemas. For example, $Schema1 \fatsemi Schema2$ is intended to specify the effect of applying $Schema1$ followed by $Schema2$. It expands to a schema that equates $Schema1$'s final state variables with $Schema2$'s initial state variables, without specifying their actual values. (It does this using existential quantifiers.) Both schemas' input and output variables are gathered together to form the inputs and outputs of $Schema1 \fatsemi Schema2$. From the schema $AddBirthday \fatsemi FindBirthDay$ one can derive $date! = date?$. This illustrates Z's power and complexity — as with a programming language, one must use this power with care.

Refinement. Z does not supply a method of refining the specification into a design, but it can be used for this purpose. Spivey [27] describes how to write more concrete Z schemas for the birthday book that use arrays to implement the *birthday* function, and to show that a concrete type (here arrays) faithfully implements the abstract type (functions).

Tool support. Part of the effort of writing a Z specification is neat presentation. These lecture notes were produced with the help of `oz.sty`, a LaTeX style file. More elaborate tools perform type checking and other simple consistency checks. Z is not directly concerned with theorem proving, but there has been some research into support for Z using theorem provers such as HOL and Isabelle. Commercial tools (suitably priced!) are available too.

Z has been under development for a long time, and the Z Standard is nearing maturity. But research is continuing; methods under development include Object Z and B.

Object Z [9] extends Z with object-oriented features. 'The main reason for this extension is to improve the clarity of large specifications through enhanced structuring.' Object-Z introduces a class structure with a private state schema, packaged together with the operations that may affect that state. This attacks the problem, also found in programming, that a global state can be modified by any operation anywhere.

The B method, developed by J.-R. Abrial, has been described in a previous lecture. Sophisticated tools have been developed to support it.

Slide 601

Large Software Systems: A Study

17 large, demanding systems (many real-time)

97 interviews: system engineers, designers, managers, . . .

role of team and organizational factors

problems identified:

- *thin spread of application domain knowledge*
- *fluctuating and conflicting requirements*
- *communication and coordination breakdowns*

This is a classic study [8] by three researchers at MCC (Microelectronics and Computer Technology Corporation), Austin. They employed field research methods and investigated the effect of human factors. This was in response to findings that new methods and tools for software engineering did not yield the expected benefits.

They employed a layered behavioural model because software is a human artifact and is subject to psychological, social and organisational processes. The layers considered were individual, team, project, company and business milieu. When they identified a problem they examined its effect at each of those layers.

Projects ranged in size from 24,000 to a million lines of code, nearly all over 100,000 lines of code. Typical applications were defence, communications and system software. Large projects deserve study because, when they go wrong, the result is disaster. Large projects are not just scaled-up small projects, but exhibit different phenomena.

Slide 602

Application Domain Knowledge

- avionics? telephony? transaction processing?
- few people with *global* knowledge of application
  – Understanding one component is not good enough!
  – results in specification mistakes
- *exceptional designers*: ‘intellectual core of project’
  – extremely familiar with domain
  – interdisciplinary: integrated several domains
  – could see constraints & exceptions

Many of the projects involved extremely demanding applications with no connection with Computer Science. The only way staff could obtain knowledge of such applications was by additional training or experience with previous projects. Such people were few; those who did have the knowledge (‘exceptional designers’ or ‘project gurus’) often took over the project. They were often said to have ‘saved’ the project.

‘They usually possessed *exceptional communication skills* and often spent much of their time educating others about the application domain. . . ’ (italics mine). They tended to take over project meetings because of their superior knowledge.

Exceptional designers undergo exceptional stress, with attendant health risks. They become essential to the project and to their company.

The authors found that building a failed system at least spread application knowledge through the team. They were then able to try again and succeed. This sort of prototype yields more insight than would a ‘rapid prototype’ that does not explore the subtle problems. This echoes Fred Brooks’s advice, ‘plan to throw one away.’

Companies had to spend large sums on training, which customers were unwilling to pay for. Managers often could not assess their staff’s ability to cope with a demanding application. The problem was exacerbated if the company undertook projects in too many different domains.

Changing Requirements

Slide 603

Powerful forces to change requirements:

- *external*: technological change, competing products, new standards
- *internal*: corporate politics, marketing plans, financial conditions
- new customers, or customers wanting more
- team understanding the requirements better

‘Requirements caused problems on every large project. . . ’

It is easy to say that the requirements should just be frozen. But what if technological change threatens to make your product obsolete? What if a competitor comes out with something better than your product? What if updated standards affect your product? These are some external factors.

The internal factors (politics, marketing, . . .) all boil down to demands from management. Even if we pretend that corporate politics do not exist, we are left with marketing and financial considerations, which could be impossible to resist.

Customers often request frequent changes — for many reasons. One of the more interesting is that they learn as their software develops, and see that it might do more. But sometimes a project is undertaken for one customer and then generalized as a second customer comes along. And then comes a third customer . . . .

Change requests from customers can be forbidden or discouraged by penalties. An extreme case is the Docklands Light Railway, which stopped at a station that had never been built [29, page 100]. The software had already been tested, and the developers refused to change it after the station was cancelled. Luckily the doors did not open!

Finally, requirements may appear to change if team lacks domain knowledge. This happens if coding starts before the problem has been digested fully. Another cause is hasty formulation of requirements due to the pressures of competitive tendering.

Slide 604

Requirements: Ultimate Constraints

- the capacity of the hardware
- tight timing & storage constraints
- delivery date
- market conditions
- regulation & legislation

There are grim tales of projects that staggered on when everybody knew the existing hardware could never do the job. Hoare [18] describes building software for the Elliot 503 computer; the software was simply too big for the computer's store, and the whole project had to be abandoned. This was a small-scale flop compared with modern disasters. Sometimes the developer insists on using its own hardware when only a competitor's hardware is up to the job.

There are innumerable cases of 'creeping featurism' causing products to become more and more elaborate, until they exceed the capacity of the hardware. Owners of personal computers often discover this (to their cost) when they upgrade their software. Some programmers find it hard to resist the urge to add features beyond the requirements.

'Some of the toughest decisions involved tradeoffs between system features and the current market trends.' Features might be omitted in order to meet the delivery date. They might be added to beat a competitor or keep up with technological advances.

Government action can have a dramatic effect. The 1991 Budget simultaneously increased VAT by 2.5% and decreased everyone's Poll Tax by £200. Both changes required urgent responses by programmers. Changing the rate of VAT sounds trivial, but utility companies had to work out a fair way of charging for the increase without reading all meters on the changeover date. The Poll Tax subsidy was complicated by changes to transitional relief. To compound the problems, the Chancellor (Norman Lamont) postponed the announcement until the Budget Speech, after millions of Poll Tax bills had already been sent out.

Slide 605

Communication Breakdowns

- colleagues, other teams, project management
- other projects, marketing, senior management
- prime contractor, subcontractors, Government departments

Who is the real customer?

Scaling up communication strategies?

Design to avoid communication?

People communicate to understand the problem, to work out requirements conflicts, to resolve technical issues, to diagnose faults, to monitor progress, etc. The authors list dozens of different points of communication, at different levels from team to business milieu. A few are shown above.

None of the projects had a unique customer representative. Different senior members of the purchasing organisation would regard themselves as *the* customer, though they wanted different things. Military projects appeared to be the worst: officials, commanders, and related agencies all meddled in the requirements specification. If the software developer is only a subcontractor, they still have to deal with these people and with the prime contractor as well.

Secrecy hindered communication in military projects. Information needed to understand the problem thoroughly was classified. Security needs, and also political rivalry, hindered communication even between departments of a single company. '. . . many projects spent tremendous time rediscovering information that, in many cases, had already been generated by customers, but not transmitted to developers.' 'Even the product description . . . is a secret document . . . '

The authors note that 'good practice' communication strategies did not scale up to large projects, especially under time pressure. Sometimes this influenced the system design: '. . . it was more important to minimize the interfaces between system engineers than it was to make the system logical from the viewpoint of the user.'

Slide 606

Face-To-Face: the Only Way

- distortion/filtering over long distances
- misunderstandings due to different contexts
- inadequacies of documentation
- role of *boundary spanners*
- talk 10 hours per day, work for 5?

Communication over long chains of command is difficult. Messages can be suppressed or altered. This ‘massaging’ takes time; senior management get their distorted messages very late.

Lauren Wiener [29, page 80] reports a case where September’s progress report had to be written in August in order to get through six layers of management by mid-October. The CONFIRM project continued much longer than it deserved because management refused to believe bad news [24].

Messages are also misunderstood because the recipient’s assumptions and background knowledge differ from the senders. For instance, senior management may be unable to understand the seriousness of a technical problem. Programmers may not understand the user’s working environment.

People obviously find talking more efficient than writing and reading documentation. Documentation therefore suffers, particularly because of the usual time pressure. If the original staff then leave (some firms have a high turn-over rate) then future staff will be in difficulties.

The authors give ‘boundary spanners’ credit for avoiding some communication breakdowns. They have good communication skills: they can translate information from one group to another, expressing it in a form that can be understood. They are willing to talk all day. They are good negotiators and carry messages for others.

One systems engineer said he spent a third of his time talking. Another said the only way was 10 hours of meetings per day and 5 hours work. More talking than work? And what about the engineer’s health?

Slide 607

Implications for Methods and Tools

- sharing/integration of knowledge across staff
- change management and propagation
- coordination

Use tools *where relevant*

No methods or tools can make much of a difference in large projects unless they address the key limitations identified above. Indeed, the project was undertaken in order to understand why such methods had failed.

Computer-Assisted Software Engineering (CASE) tools typically let you draw data and control flow diagrams, etc., and assist their refinement into code. Such tools can do nothing about lack of domain knowledge and nothing about pressures to change requirements. By improving the quality of documentation, they may do something about communication.

But tools can actually impede communication. If information from outside has to be entered manually, then either the information will be lost or the tools will not be used. Tools must be well designed to fit into the work environment.

Formal methods suffer similar limitations. They can make the requirements and documentation precise, but their intended readers may be unable to understand them!

Methods and tools aimed at individuals cannot do much in large projects where training and communication costs predominate. But they are not useless either. For example, in a large safety-critical system, often only a small part has a true safety function; other parts may be mainly for convenience. We can use formal methods just in the safety-critical components.

Tools will never replace face-to-face discussions. Some disasters might be averted if senior management would visit development staff in person.

Slide 608

Implications for Management

- limitations of ‘waterfall’ model
  – include learning, communication and negotiation
  – requirements: always an issue
  – Boehm’s spiral model
- importance of a track record?
- importance of good people

The three problems — knowledge, requirements, communication — are interrelated. Lack of domain knowledge leads to inaccurate requirements, which hinder communication, etc. The ‘natural tension between getting requirements right and getting them stable’ could be eased if the procurer would appoint a *single* person as its representative. Requirements issues occur throughout a development, not just at the official requirements phase.

The basic waterfall model does not account for these problems, which are characteristically human. It does not take account of the role of learning and negotiation. Boehm’s spiral model is more elaborate and calls for risk analysis at each iteration, when key decisions are made.

What does a good track record signify? If the software developer has already built a similar system, then it ought to have some means of tackling these problems. In particular, it must already possess the necessary domain knowledge. Yet the CONFIRM disaster involved the developer of one of the world’s most successful reservation systems.

The role of key people is stressed time and time again, by Brooks [5] as well as in this study. Good people are not just those who can code quickly or solve hard Computer Science problems. They must be willing to learn a lot about the application domain, which may have nothing to do with Computer Science. Above all they must be good at negotiating with customers, management and rival groups.

Recruiting good people is important, but it is also necessary to ‘grow’ good people and teams in-house through training.

Acknowledgements. Michael Hinchey provided considerable literature and advice, as did Ross Anderson and Michael Gordon. Ajay Chander supplied numerous corrections.

References

[1] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, 1989.

[2] Alan Borning. Computer system reliability and nuclear war. *Communications of the ACM*, 30(2):112–131, 1987.

[3] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, July 1993.

[4] Frederick P. Brooks. *The Mythical Man-Month*. Reading, 1975.

[5] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, pages 10–19, April 1987.

[6] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 426:233–271, 1989.

[7] Dan Craigen, Susan Gerhart, and Ted Ralston. Case study: Darlington nuclear generating station. *IEEE Software*, pages 30–32, January 1994.

[8] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, November 1988.

[9] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.

[10] J. S. Fitzgerald, P. G. Larsen, T. M. Brookes, and M. A. Green. Developing a security-critical system using formal and conventional methods. In Hinchey and Bowen [16], pages 333–356.

[11] Steve Flowers. One huge crash. *Guardian*, page 21, 1994. 28 April.

[12] John Gall. *Systemantics: How Systems Work and Especially How They Fail*. Fontana, 1979.

[13] Susan Gerhart, Dan Craigen, and Ted Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

[14] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, pages 72–81, September 1994.

[15] Anthony Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[16] Michael Hinchey and Jonathan P. Bowen, editors. *Applications of Formal Methods*. Prentice-Hall, 1995.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. In Hoare and Jones [19], pages 45–58. Originally published in 1969.

[18] C. A. R. Hoare. The emperor's old clothes. In Hoare and Jones [19], pages 1–18. Turing Award Lecture delivered in 1980.

[19] C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. Prentice-Hall, 1989.

[20] Nancy Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[21] Bev Littlewood and Lorenzo Strigini. Validation of ultrahigh dependability for software-based systems. *Communications of the ACM*, 36(11):69–80, 1993.

[22] Motor Industry Software Reliability Association. *Development Guidelines For Vehicle Based Software*. Motor Industry Research Association, 1994.

[23] Peter G. Neumann. *Computer-Related Risks*. ACM Press, 1994.

[24] Effy Oz. When professional standards are lax: The CONFIRM failure and its lessons. *Communications of the ACM*, 37(10):29–36, October 1994.

[25] Ian Sommerville. *Software Engineering*. Addison-Wesley, 4th edition, 1992.

[26] South West Thames Regional Health Authority. Report of the inquiry into the london ambulance service. 40 Eastbourne Terrace, London W2 3QR, February 1993.

[27] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.

[28] D. A. Turner. Functional programs as executable specifications. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 29–54. Prentice-Hall, 1985.

[29] Lauren Ruth Wiener. *Digital Woes: Why We Should Not Depend on Software*. Addison-Wesley, 1993.