Dependable systems

2

System dependability

- For many computer-based systems, the most important system property is the dependability of the system.
- The dependability of a system reflects the **user's degree of trust in that system**.
- It reflects the extent of the **user's confidence** that it will operate as users expect and that it will not 'fail' in normal use.
- Dependability covers the related systems attributes of **reliability**, **availability** and **security**.
- These are all inter-dependent.

3

Importance of dependability

- System failures may have widespread effects with large numbers of people affected by the failure.
- Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users.
- The costs of system failure may be very high if the failure leads to economic losses or physical damage.
- Undependable systems may cause information loss with a high consequent recovery cost.

4

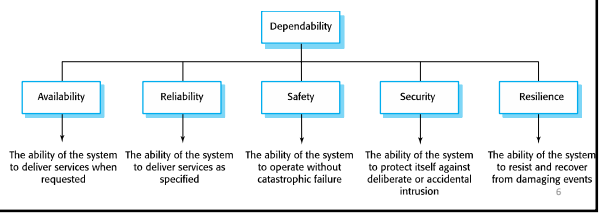
Dependability properties

- The dependability of a computer system is a property of the system that reflects its trustworthiness.
- Trustworthiness here essentially means the degree of confidence a user has that the system will operate as they expect, and that the system will not 'fail' in normal use.
- It is not meaningful to express dependability numerically.
- Rather, we use relative terms such as 'not dependable,' 'very dependable,' and 'ultra-dependable' to reflect the degrees of trust that we might have in a system.

5

Primary dependability properties

- The dependability properties are complex properties that can be broken down into a number of other, simpler properties.
  - For example, security includes 'integrity' (ensuring that the systems program and data are not damaged) and 'confidentiality' (ensuring that information can only be accessed by people who are authorized).
  - Reliability includes 'correctness' (ensuring the system services are as specified), 'precision' (ensuring information is delivered at an appropriate level of detail), and 'timeliness' (ensuring that information is delivered when it is required)



6

Principal properties

- Availability
  - The probability that the system will be up and running and able to deliver useful services to users.
- Reliability
  - The probability that the system will correctly deliver services as expected by users.
- Safety
  - A judgment of how likely it is that the system will cause damage to people or its environment.
- Security
  - A judgment of how likely it is that the system can resist accidental or deliberate intrusions.
- Resilience
  - A judgment of how well a system can maintain the continuity of its critical services in the presence of disruptive events such as equipment failure and cyberattacks.

7

Other dependability properties

- Repairability
  - Reflects the extent to which the system can be repaired in the event of a failure
- Maintainability
  - Reflects the extent to which the system can be adapted to new requirements;
- Error tolerance
  - Reflects the extent to which user input errors can be avoided and tolerated.

8

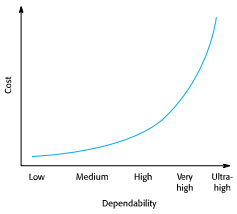
Dependability achievement

- To develop dependable software, you need to ensure that:
  - **Avoid the introduction of accidental errors** when developing the system.
  - **Design V & V processes that are effective** in discovering residual errors in the system.
  - Design systems to be **fault tolerant** so that they can continue in operation when faults occur
  - Design **protection mechanisms** that guard against external attacks.
  - **Configure the system correctly** for its operating environment.
  - Include system capabilities to **recognise and resist cyberattacks**.
  - Include **recovery mechanisms** to help restore normal system service after a failure.

9

Dependability costs

- Dependability **costs tend to increase exponentially** as increasing levels of dependability are required.
- There are 2 reasons for this
  - The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability.
  - The increased testing and system validation that is required to convince the system client and regulators that the required levels of dependability have been achieved.



10

Dependability economics

- Because of very high costs of dependability achievement, it may be more cost effective to accept untrustworthy systems and pay for failure costs
- However, this depends on social and political factors.
  - A reputation for products that can't be trusted may lose future business
- Depends on system type - for business systems in particular, modest levels of dependability may be adequate

11

Regulation & compliance

- The general model of economic organization that is now almost universal in the world is that privately owned companies offer goods and services and make a profit on these.
- To ensure the safety of their citizens, most governments **regulate** (limit the freedom of) privately owned companies so that they must follow certain **standards** to ensure that their products are **safe** and **secure**.

12

Regulated systems

- Many **critical systems** are **regulated systems**, which means that their use must be approved by an **external regulator** before the systems go into service.
  - Nuclear systems
  - Air traffic control systems
  - Medical devices
- A safety and dependability case has to be approved by the regulator.
  - Therefore, critical systems development has to create the evidence to convince a regulator that the system is dependable, safe and secure.
- As well as validating that the system meets its requirements, the validation process may have to prove to an external regulator that the system is safe.
  - For example, aircraft systems have to demonstrate to regulators, such as the US Federal Aviation Authority, that the probability of a catastrophic system failure that affects aircraft safety is extremely low.

13

Safety regulation

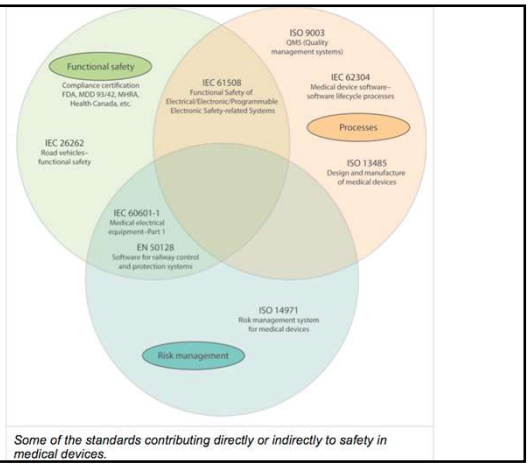
- Safety-related systems may have to be **certified** as safe by the regulator.
- To achieve certification, companies that are developing safety-critical systems have to produce an extensive safety case that shows that rules and regulations have been followed.
- It can be as expensive to develop the documentation for certification as it is to develop the system itself.

14

Safety regulation

- Did you know?
  - Medical errors are the third leading cause of death in the United States (Journal of Patient Safety)
  - In 2012, the FDA reported that software failures were behind 24% of all the medical device recalls
- Medical device software example
  - In the medical device domain, ISO 13485 identifies the requirements for regulatory purposes from a quality management system perspective.
  - ISO 13485 can be used to assess an organisation's ability to meet both customer and regulatory requirements.
  - However, ISO 13485 does not offer specific guidance on software development.
  - IEC 62304, which can be used in conjunction with ISO 13485, offers a framework for the lifecycle processes necessary for the safe design and maintenance of medical device software.

15



Some of the standards contributing directly or indirectly to safety in medical devices.

Redundancy and diversity

17

Redundancy and diversity

- Redundancy
  - Keep more than a single version of critical components so that if one fails then a backup is available.
  - Where availability is critical (e.g. in e-commerce systems), companies normally keep backup servers and switch to these automatically if failure occurs.
- Diversity
  - Provide the same functionality in different ways in different components so that they will not fail in the same way.
  - To provide resilience against external attacks, different servers may be implemented using different operating systems
- Redundant and diverse components should be independent so that they will not suffer from 'common-mode' failures
  - For example, components implemented in different programming languages means that a compiler fault will not affect all of them.

18

Analogy / Example

- Commonly, to secure our homes we use more than one lock (**redundancy**) and, usually, the locks used are of different types (**diversity**).
  - This means that if an intruder finds a way to defeat one of the locks, they have to find a different way of defeating the other lock before they can gain entry.
- In systems for which availability is a critical requirement, redundant servers are normally used.
  - These automatically come into operation if a designated server fails.
  - Sometimes, to ensure that attacks on the system cannot exploit a common vulnerability, these servers may be of different types and may run different operating systems.

19

Process diversity and redundancy

- **Process activities**, such as validation, should not depend on a single approach, such as testing, to validate the system.
- **Redundant** and **diverse** process activities are important especially for **verification** and **validation**.
- Multiple, different process activities the complement each other and allow for cross-checking help to avoid process errors, which may lead to errors in the software.

20

Problems with redundancy and diversity

- Adding diversity and redundancy to a system **increases the system complexity**.
- This can **increase the chances of error** because of unanticipated interactions and dependencies between the redundant system components.
- **Some engineers therefore advocate simplicity** and extensive V & V as a more effective route to software dependability.

21

Problems with redundancy and diversity – an example

- The Ariane launcher accident
  - There were two inertial navigation computers which were diverse and so the system could cope with hardware failure.
  - These ran in parallel during the launch so that a single computer failure would not lead to any loss of capability.
  - However, the **software installed on each of these computers was identical**, so the overflow occurred at the same time on both the principal computer and its backup system.
- Lesson learnt
  - The subsequent inquiry concluded that there was a failure to understand the importance of software redundancy (the Ariane 5 engineers mostly had a hardware background) and a failure in the verification and validation of the software. The launch simulation tests and code reviews were inadequate
  - There was a **backup system** but this was **not diverse** and so the software in the backup computer failed in exactly the same way.

22

Formal methods and dependability

23

Formal specification

- For more than 30 years, many researchers have advocated the use of formal methods of software development.
- Formal methods are mathematically-based approaches to software development where you **define a formal model of the software**.
- You may then **formally analyze this model** and use it as a basis for a formal system specification.
- In principle, it is possible to start with a formal model for the software and **prove** that a developed program is consistent with that model, thus eliminating software failures resulting from programming errors.

24

Formal specification

- Formal methods are approaches to software development that are **based on mathematical representation and analysis of software**.
- Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.
- Formal methods **significantly reduce some types of programming errors** and can be cost-effective for dependable systems engineering.

25

Formal approaches

- **Verification-based approaches**
  - Different representations of a software system such as a specification and a program implementing that specification are **proved to be equivalent**.
  - This demonstrates the **absence of implementation errors**.
- **Refinement-based approaches**
  - A representation of a system is systematically transformed into another, lower-level representation
    - e.g. a specification is transformed automatically into an implementation.
  - This means that, **if the transformation is correct, the representations are equivalent**.

26

Use of formal methods

- The principal benefits of formal methods are in reducing the number of faults in systems.
- Consequently, their main area of applicability is in dependable systems engineering.
- There have been several successful projects where formal methods have been used in this area.
- In this area, the use of formal methods is most likely to be cost-effective because high system failure costs must be avoided.

27

Benefits of formal specification

- Developing a formal specification requires the **system requirements to be analyzed in detail**. This helps to detect problems, inconsistencies and incompleteness in the requirements.
- As the **specification** is expressed in a formal language, it **can be automatically analyzed to discover inconsistencies and incompleteness**.
- If you use a formal method such as the B method, you can **transform the formal specification into a 'correct' program**.
- Program **testing costs may be reduced** if the program is formally verified against its specification.

28

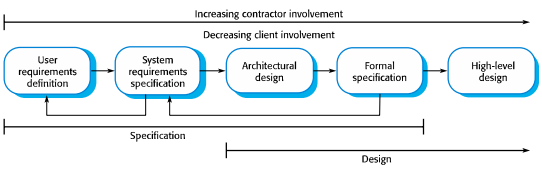
Acceptance of formal methods

- Formal methods **have not become mainstream** software development techniques as was once predicted and have had **limited impact on practical software development**:
  - Problem owners **cannot understand a formal specification** and so cannot assess if it is an accurate representation of their requirements.
  - It is easy to assess the **costs** of developing a formal specification but harder to assess the **benefits**. Managers may therefore be unwilling to invest in formal methods.
  - **Software engineers are unfamiliar** with this approach and are therefore reluctant to propose the use of FM.
  - The **scope of formal methods is limited**. They are **not well-suited** to specifying and analysing **user interfaces and user interaction**;
  - Market changes have made **time-to-market** rather than software with a low error count the **key factor**. Formal methods do not reduce time to market;
  - Formal methods are still **hard to scale up** to large systems.
  - Formal specification is **not really compatible with agile** development methods.

29

Specification in the software process

- Specification and design are inextricably intermingled.
- Architectural design is essential to structure a specification and the specification process.
- Formal specifications are expressed in a mathematical notation with precisely defined vocabulary, syntax and semantics.

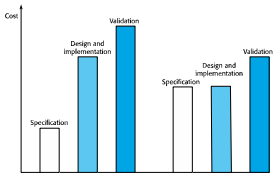


Use of formal specification

- Formal specification involves **investing more effort in the early phases** of software development.
- This **reduces requirements errors** as it forces a detailed analysis of the requirements.
- Incompleteness and inconsistencies can be discovered and resolved.
- Hence, savings as made as the amount of rework due to requirements problems is reduced.

Cost profile

- The use of formal specification means that the cost profile of a project changes
  - There are greater up front costs as more time and effort are spent developing the specification;
  - However, implementation and validation costs should be reduced as the specification process reduces errors and ambiguities in the requirements.



Specification techniques

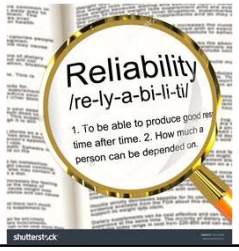
- Two fundamental approaches to formal specification have been used to write detailed specifications for industrial software systems.
- **Algebraic specification**
  - The **system is specified in terms of its operations and their relationships**.
- **Model-based specification**
  - The system is **specified in terms of a state model** that is constructed using mathematical constructs such as sets and sequences.
  - **Operations are defined by modifications to the systems state**.

Formal specification languages

- Different languages in these families have been developed to specify sequential and concurrent systems
- Most of these languages were developed in the 1980s. It takes several years to refine a formal specification language, so **most formal specification research is now based on these languages** and is not concerned with inventing new notations.

	Sequential	Concurrent
Algebraic	Larch (Guttag et al., 1993) , OBJ (Futatsugi et al., 1985)	Lotos (Bolognesi and Brinksma, 1987) ,
Model-based	Z (Spivey, 1992) VDM (Jones, 1980) B (Wordsworth, 1996)	CSP (Hoare, 1985) Petri Nets (Peterson, 1981)

Reliability Engineering



35

Software reliability and reliability achievement

- In general, software customers expect all software to be dependable.
- However, for non-critical applications, they **may be willing to accept some system failures**.
- Some applications (critical systems) have very high reliability requirements and special software engineering techniques may be used to achieve this.
  - Medical systems
  - Telecommunications and power systems
  - Aerospace systems

36

Fault management

- **Fault avoidance**
  - The system is **developed in such a way that human error is avoided** and thus system faults are minimised.
  - The development process is organised so that faults in the system are detected and repaired before delivery to the customer.
  - Development technique are used that either minimise the possibility of mistakes or trap mistakes before they result in the introduction of system faults.
- **Fault detection and removal**
  - **Verification and validation techniques are used to discover and remove faults in a system before it is deployed**.
  - Verification and validation techniques are used that increase the probability of detecting and correcting errors before the system goes into service are used.
- **Fault tolerance**
  - The **system is designed so that faults in the delivered software do not result in system failure**.
  - Run-time techniques are used to ensure that system faults do not result in system errors and/or that system errors do not lead to system failures.

37

Availability and reliability

- Reliability
  - The probability of failure-free system operation **over a specified time** in a given environment for a given purpose
- Availability
  - The probability that a system, **at a point in time**, will be operational and able to deliver the requested services
- Both of these attributes can be expressed quantitatively
  - e.g. availability of 0.999 means that the system is up and running for 99.9% of the time.

38

Perceptions of reliability

- The formal definition of reliability does not always reflect the user's perception of a system's reliability
  - The assumptions that are made about the environment where a system will be used may be incorrect
    - Usage of a system in an office environment is likely to be quite different from usage of the same system in a university environment
  - The consequences of system failures affects the perception of reliability
    - Unreliable windscreen wipers in a car may be irrelevant in a dry climate
    - Failures that have serious consequences (such as an engine breakdown in a car) are given greater weight by users than failures that are inconvenient

39

Availability perception

- Availability is usually expressed as a percentage of the time that the system is available to deliver services e.g. 99.95%.
- However, this does not take into account two factors:
  - The number of users affected by the service outage.
    - Loss of service in the middle of the night is less important for many systems than loss of service during peak usage periods.
  - The length of the outage.
    - The longer the outage, the more the disruption.
    - Several short outages are less likely to be disruptive than 1 long outage.
    - Long repair times are a particular problem.

40

Fault tolerance

- In critical situations, software systems must be fault tolerant.
- Fault tolerance is required where there are high availability requirements or where system failure costs are very high.
- **Fault tolerance means that the system can continue in operation in spite of software failure**.
- Even if the system has been proved to conform to its specification, it must also be fault tolerant as there may be specification errors or the validation may be incorrect.

41

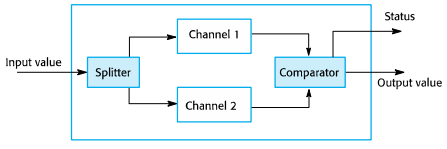
Fault-tolerant system architectures

- Fault-tolerant systems architectures are used in situations where fault tolerance is essential. These **architectures are generally all based on redundancy and diversity**.
- Examples of situations where dependable architectures are used:
  - Flight control systems, where system failure could threaten the safety of passengers
  - Reactor systems where failure of a control system could lead to a chemical or nuclear emergency
  - Telecommunication systems, where there is a need for 24/7 availability.

42

Self-monitoring architectures

- Multi-channel architectures where the **system monitors its own operations and takes action if inconsistencies are detected**.
- The same computation is carried out on each channel and the results are compared.
  - If the results are identical and are produced at the same time, then it is assumed that the system is operating correctly.
  - If the results are different, then a failure is assumed and **a failure exception is raised**.



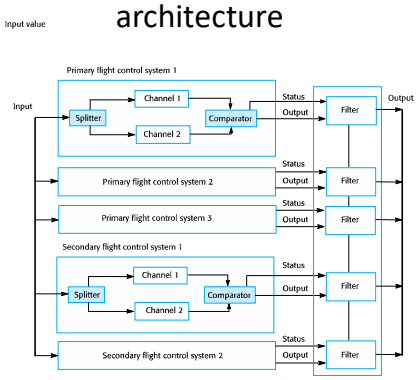
43

Self-monitoring systems

- **Hardware in each channel has to be diverse** so that common mode hardware failure will not lead to each channel producing the same results.
- **Software in each channel must also be diverse**, otherwise the same software error would affect each channel.
- If high-availability is required, you **may use several self-checking systems in parallel**.
  - This is the approach used in the Airbus family of aircraft for their flight control systems.

44

Airbus flight control system architecture



45

Airbus architecture discussion

- The Airbus FCS has 5 separate computers, any one of which can run the control software.
- Extensive use has been made of diversity
  - Primary systems use a **different processor** from the secondary systems.
  - Primary and secondary systems **use chipsets from different manufacturers**.
  - Software in secondary systems is less complex than in primary system – provides only critical functionality.
  - **Software in each channel is developed in different programming languages by different teams**.
  - **Different programming languages** used in primary and secondary systems.

46

Security Engineering

60

Security engineering

- **Tools, techniques and methods** to support the development and maintenance of **systems that can resist malicious attacks** that are intended to damage a computer-based system or its data.
- A sub-field of the broader field of computer security.

61

Security dimensions

- **Confidentiality**
  - Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information.
- **Integrity**
  - Information in a system may be damaged or corrupted making it unusual or unreliable.
- **Availability**
  - Access to a system or its data that is normally available may not be possible.

62

Security levels

- **Infrastructure security**, which is concerned with **maintaining the security of all systems** and networks that provide an infrastructure and a set of shared services to the organization.
- **Application security**, which is concerned with the **security of individual application systems** or related groups of systems.
- **Operational security**, which is concerned with the **secure operation and use of the organization's systems**.

63

Security requirements

- Security specification has something in common with safety requirements specification – in both cases, your concern is to avoid something bad happening.
- Four major differences
  - **Safety problems are accidental** – the software is not operating in a hostile environment. In security, you must assume that attackers have knowledge of system weaknesses
  - When safety failures occur, you can look for the **root cause or weakness that led to the failure**. When failure results from a deliberate attack, the attacker may conceal the cause of the failure.
  - **Shutting down** a system can avoid a safety-related failure. Causing a shut down may be the aim of an attack.
  - Safety-related events are not generated from an intelligent adversary. An attacker can probe defenses over time to discover weaknesses.

64

Types of security requirement

- Identification requirements.
- Authentication requirements.
- Authorisation requirements.
- Immunity requirements.
- Integrity requirements.
- Intrusion detection requirements.
- Non-repudiation requirements.
- Privacy requirements.
- Security auditing requirements.
- System maintenance security requirements.

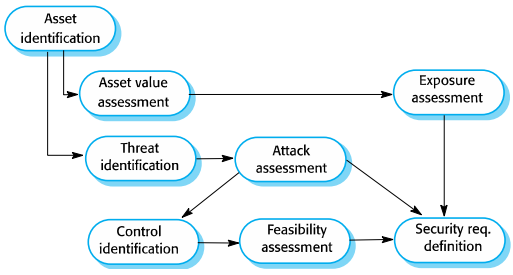
65

Security requirement classification

- **Risk avoidance requirements** set out the **risks that should be avoided** by designing the system so that these **risks simply cannot arise**.
- **Risk detection requirements** define mechanisms that **identify the risk if it arises** and **neutralise the risk before losses occur**.
- **Risk mitigation requirements** set out how the system should be designed so that it can **recover from and restore system assets** after some loss has occurred.

66

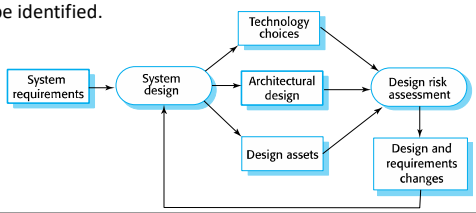
The preliminary risk assessment process for security requirements



67

Design risk assessment

- Risk assessment while the system is being developed and after it has been deployed
- More information is available - system platform, middleware and the system architecture and data organisation.
- Vulnerabilities that arise from design choices may therefore be identified.



68

Design guidelines for security engineering

- Design guidelines encapsulate good practice in secure systems design
- Design guidelines serve a number of purposes:
  - They **raise awareness of security issues** in a software engineering team.
  - **Security is considered** when design decisions are made.
  - They can be used as the **basis of a review checklist** that is applied during the system validation process.
- Design guidelines here are applicable during software specification and design

69

Design guidelines 1-3

- Base decisions on an **explicit security policy**
  - Define a security policy for the organization that sets out the fundamental security requirements that should apply to all organizational systems.
- **Avoid a single point of failure**
  - Ensure that a security failure can only result when there is more than one failure in security procedures.
    - For example, have password and question-based authentication.
- **Fail securely**
  - When systems fail, for whatever reason, ensure that sensitive information cannot be accessed by unauthorized users even although normal security procedures are unavailable.

70

Design guidelines 4-6

- **Balance security and usability**
  - Try to avoid security procedures that make the system difficult to use.
  - Sometimes you have to accept weaker security to make the system more usable.
- **Log user actions**
  - Maintain a log of user actions that can be analyzed to discover who did what. If users know about such a log, they are less likely to behave in an irresponsible way.
- **Use redundancy and diversity to reduce risk**
  - Keep multiple copies of data and use diverse infrastructure so that an infrastructure vulnerability cannot be the single point of failure.

71

Design guidelines 7-10

- **Specify the format of all system inputs**
  - If input formats are known then you can check that all inputs are within range so that unexpected inputs don't cause problems.
- **Compartmentalize your assets**
  - Organize the system so that assets are in separate areas and users only have access to the information that they need rather than all system information.
- **Design for deployment**
  - Design the system to avoid deployment problems
- **Design for recoverability**
  - Design the system to simplify recoverability after a successful attack.

72

Security testing

- Testing the **extent to which the system can protect itself from external attacks**.
- Problems with security testing
  - Security requirements are 'shall not' requirements i.e. they **specify what should not happen**.
    - It is not usually possible to define security requirements as simple constraints that can be checked by the system.
  - The people **attacking a system are intelligent and look for vulnerabilities**.
    - They can **experiment** to discover weaknesses and loopholes in the system.

73

Security validation

- **Experience-based testing**
  - The system is reviewed and analysed against the **types of attack that are known** to the validation team.
- **Penetration testing**
  - A team is established whose **goal is to breach the security of the system by simulating attacks** on the system.
- **Tool-based analysis**
  - Various **security tools** such as password checkers are used to analyse the system in operation.
- **Formal verification**
  - The system is verified against a **formal security specification**.

74

Programming for reliability

75

Dependable programming

- Good programming practices can be adopted that help reduce the incidence of program faults.
- These programming practices support
  - Fault avoidance
  - Fault detection
  - Fault tolerance

76

Good practice guidelines for dependable programming

- Dependable programming guidelines
  1. Limit the visibility of information in a program
  2. Check all inputs for validity
  3. Provide a handler for all exceptions
  4. Minimize the use of error-prone constructs
  5. Provide restart capabilities
  6. Check array bounds
  7. Include timeouts when calling external components
  8. Name all constants that represent real-world values

77

(1) Limit the visibility of information in a program

- Program components should only be allowed access to data that they need for their implementation.
- This means that accidental corruption of parts of the program state by these components is impossible.
- You can control visibility by using abstract data types where the data representation is private and you only allow access to the data through predefined operations such as get () and put ().

78

(2) Check all inputs for validity

- All program take inputs from their environment and make assumptions about these inputs.
- However, program specifications rarely define what to do if an input is not consistent with these assumptions.
- Consequently, many programs behave unpredictably when presented with unusual inputs and, sometimes, these are threats to the security of the system.
- Consequently, you should always check inputs before processing against the assumptions made about these inputs.

79

Validity checks

- Range checks
  - Check that the input falls within a known range.
- Size checks
  - Check that the input does not exceed some maximum size e.g. 40 characters for a name.
- Representation checks
  - Check that the input does not include characters that should not be part of its representation e.g. names do not include numerals.
- Reasonableness checks
  - Use information about the input to check if it is reasonable rather than an extreme value.

80

(3) Provide a handler for all exceptions

- A program exception is an error or some unexpected event such as a power failure.
- Exception handling constructs allow for such events to be handled without the need for continual status checking to detect exceptions.
- Using normal control constructs to detect exceptions needs many additional statements to be
added to the program. This adds a significant overhead and is potentially error-prone.

81

(4) Minimize the use of error-prone constructs

- Program faults are usually a consequence of human error because programmers lose track of the relationships between the different parts of the system
- This is exacerbated by error-prone constructs in programming languages that are inherently complex or that don't check for mistakes when they could do so.
- Therefore, when programming, you should try to avoid or at least minimize the use of these error-prone constructs.

82

(5) Provide restart capabilities

- For systems that involve long transactions or user interactions, you should always provide a restart capability that allows the system to restart after failure without users having to redo everything that they have done.
- Restart depends on the type of system
  - Keep copies of forms so that users don't have to fill them in again if there is a problem
  - Save state periodically and restart from the saved state

83

(6) Check array bounds

- In some programming languages, such as C, it is possible to address a memory location outside of the range allowed for in an array declaration.
- This leads to the well-known 'bounded buffer' vulnerability where attackers write executable code into memory by deliberately writing beyond the top element in an array.
- If your language does not include bound checking, you should therefore always check that an array access is within the bounds of the array.

84

(7) Include timeouts when calling external components

- In a distributed system, failure of a remote computer can be 'silent' so that programs expecting a service from that computer may never receive that service or any indication that there has been a failure.
- To avoid this, you should always include timeouts on all calls to external components.
- After a defined time period has elapsed without a response, your system should then assume failure and take whatever actions are required to recover from this.

85

(8) Name all constants that represent real-world values

- Always give constants that reflect real-world values (such as tax rates) names rather than using their numeric values and always refer to them by name
- You are less likely to make mistakes and type the wrong value when you are using a name rather than a value.
- This means that when these 'constants' change (for sure, they are not really constant), then you only have to make the change in one place in your program.

86

Safety Engineering

Safety critical systems

- Systems where it is **essential** that system operation is always **safe** i.e. the system should **never cause damage** to people or the system's environment
- Examples
  - Control and monitoring systems in aircraft
  - Process control systems in chemical manufacture
  - Automobile control systems such as braking and engine management systems

Software in safety-critical systems

- Safety is a property of a system that reflects the **system's ability to operate, normally or abnormally, without danger of causing human injury or death and without damage to the system's environment**.
- It is important to consider software safety as most devices whose failure is critical now incorporate software-based control systems.
- The system may be software-controlled so that the decisions made by the software and subsequent actions are safety-critical.
- Therefore, the software behaviour is directly related to the overall safety of the system.
- Software is extensively used for checking and monitoring other safety-critical components in a system.
  - For example, all aircraft engine components are monitored by software looking for early indications of component failure.
  - This software is safety-critical because, if it fails, other components may fail and cause an accident.

Safety terminology

Term	Definition
Accident (or mishap)	An unplanned event or sequence of events which results in human death or injury, damage to property, or to the environment. An overdose of insulin is an example of an accident.
Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that measures blood glucose is an example of a hazard.
Damage	A measure of the loss resulting from a mishap. Damage can range from many people being killed as a result of an accident to minor injury or property damage. Damage resulting from an overdose of insulin could be serious injury or the death of the user of the insulin pump.
Hazard severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic, where many people are killed, to minor, where only minor damage results. When an individual death is a possibility, a reasonable assessment of hazard severity is 'very high'.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (say 1/100 chance of a hazard occurring) to 'implausible' (no conceivable situations are likely in which the hazard could occur). The probability of a sensor failure in the insulin pump that results in an overdose is probably low.
Risk	This is a measure of the probability that the system will cause an accident. The risk is assessed by considering the hazard probability, the hazard severity, and the probability that the hazard will lead to an accident. The risk of an insulin overdose is probably medium to low.

Safety achievement

- **Hazard avoidance**
  - The system is designed so that some classes of hazard simply cannot arise.
- **Hazard detection and removal**
  - The system is designed so that hazards are detected and removed before they result in an accident.
- **Damage limitation**
  - The system includes protection features that minimise the damage that may result from an accident.

Normal accidents

- Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure
  - Designing systems so that a single point of failure does not cause an accident is a fundamental principle of safe systems design.
- Almost all accidents are a result of **combinations of malfunctions** rather than single failures.
- It is probably the case that anticipating all problem combinations, especially, in software controlled systems is impossible so achieving complete safety is impossible.
  - Accidents are inevitable.

Safety specification

- The goal of safety requirements engineering is to identify protection requirements that ensure that system failures do not cause injury or death or environmental damage.
- Safety requirements may be 'shall not' requirements i.e. they define situations and events that should never occur.
- Functional safety requirements define:
  - Checking and recovery features that should be included in a system
  - Features that provide protection against system failures and external attacks

93

Safety engineering processes

- Safety engineering processes are based on reliability engineering processes
  - Plan-based approach with reviews and checks at each stage in the process
  - General goal of fault avoidance and fault detection
  - Must also include safety reviews and explicit identification and tracking of hazards

94

Regulation

- Regulators may require evidence that safety engineering processes have been used in system development
- For example:
  - The specification of the system that has been developed and records of the checks made on that specification.
  - Evidence of the verification and validation processes that have been carried out and the results of the system verification and validation.
  - Evidence that the organizations developing the system have defined and dependable software processes that include safety assurance reviews.
  - There must also be records that show that these processes have been properly enacted.

95

Agile methods and safety

- Agile methods are not traditionally used for safety-critical systems engineering
  - Extensive process and product documentation is needed for system regulation. Contradicts the focus in agile methods on the software itself.
  - A detailed safety analysis of a complete system specification is important.
  - Contradicts the interleaved development of a system specification and program.
- Some agile techniques such as test-driven development may be used

96

Safety assurance processes

- Process assurance involves **defining a dependable process and ensuring that this process is followed during the system development**.
- Process assurance focuses on:
  - Do we have the right processes?
    - Are the processes appropriate for the level of dependability required.
    - Should include requirements management, change management, reviews and inspections, etc.
  - Are we doing the processes right?
    - Have these processes been followed by the development team.
- Process assurance generates documentation
  - Agile processes have therefore not traditionally been used for critical systems. Are they now?

97

Processes for safety assurance

- Process assurance is important for safety-critical systems development:
  - Accidents are rare events so testing may not find all problems;
  - Safety requirements are sometimes 'shall not' requirements so cannot be demonstrated through testing.
- **Safety assurance activities may be included in the software process that record the analyses that have been carried out and the people responsible for these**.
  - Personal responsibility is important as system failures may lead to subsequent legal actions.

98

Formal verification

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the **ultimate static verification technique** that may be used at different stages in the development process:
  - A formal specification may be developed and mathematically analyzed for consistency.
    - This helps discover specification errors and omissions.
  - Formal arguments that a program conforms to its mathematical specification may be developed.
    - This is effective in discovering programming and design errors.

99

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- Concurrent systems can be analysed to discover race conditions that might lead to deadlock. Testing for such problems is very difficult.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

100

Arguments against formal methods

- Require specialized notations that cannot necessarily be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- Proofs may contain errors.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

101

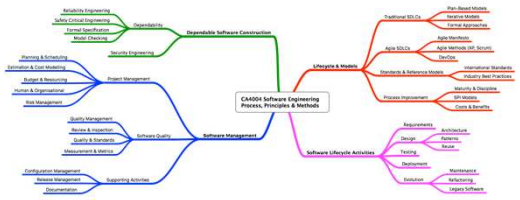
Formal methods cannot guarantee safety

- The **specification may not reflect the real requirements of system users**.
  - Users rarely understand formal notations so they cannot directly read the formal specification to find errors and omissions.
- The **proof may contain errors**.
  - Program proofs are large and complex, so, like large and complex programs, they usually contain errors.
- The **proof may make incorrect assumptions** about the way that the **system is used**.
  - If the system is not used as anticipated, then the system's behavior lies outside the scope of the proof.

102

Any questions?

The "Big Picture"



113

Exam Paper

Paper Structure

Duration: 3 hours

You MUST answer Question 1 and any 3 other questions.

Q1: 40 marks, multiple sub-parts (i.e. a, b, c, d…)

Q2, Q3, Q4, Q5: 20 Marks – 2 or 3 sub-parts (i.e. a, b and [possibly] c)

Read the questions carefully.

Stick to the subject of the question.