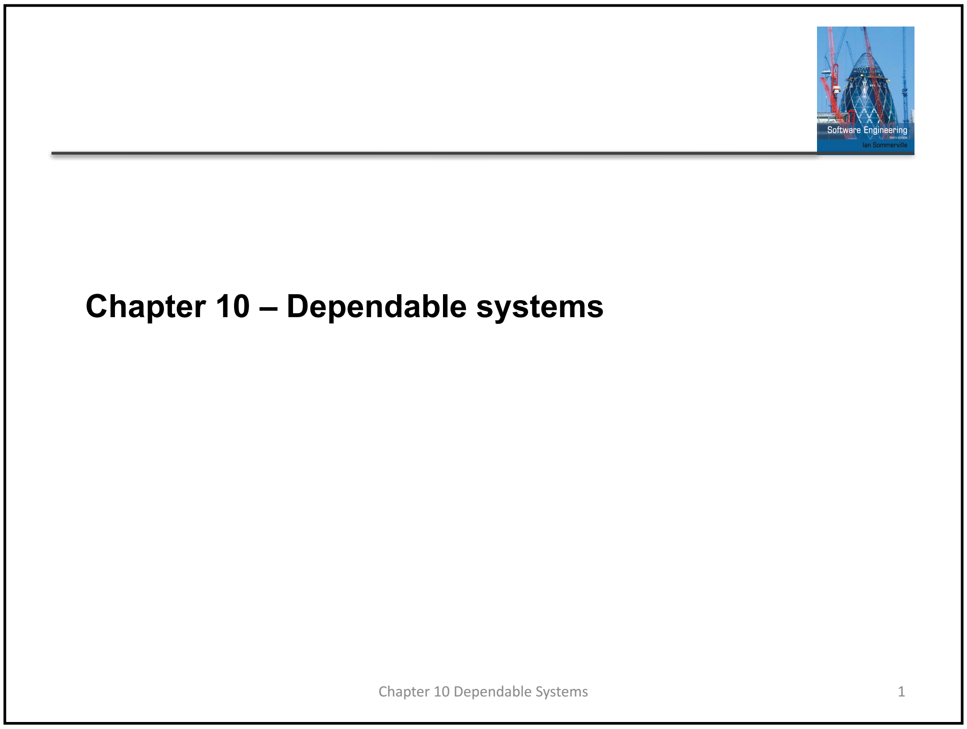
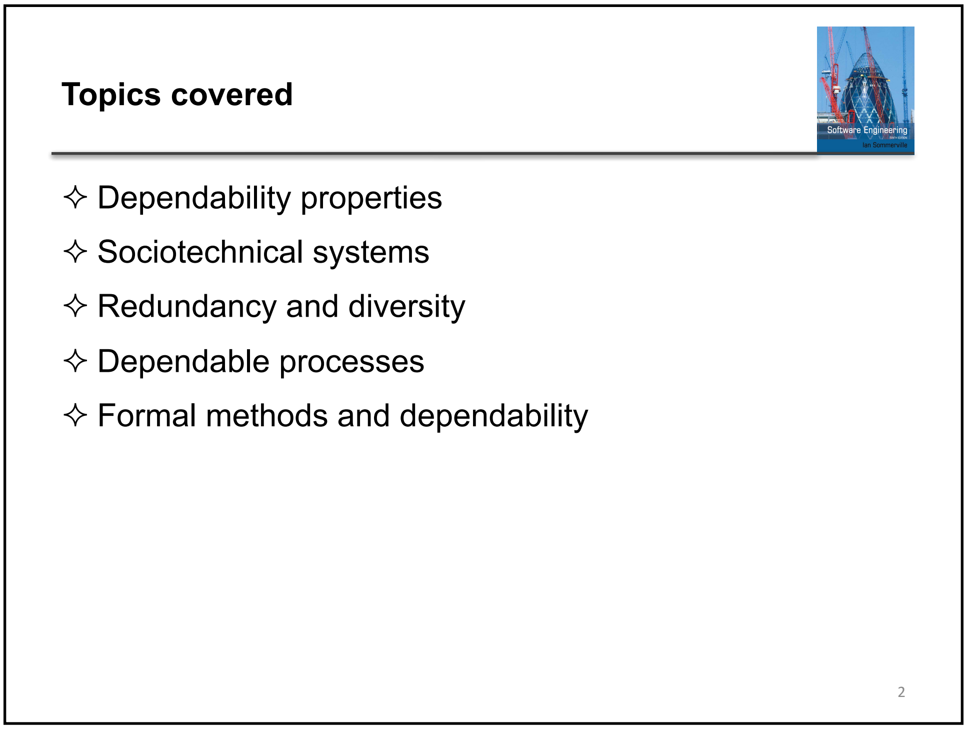


Chapter 10 – Dependable systems



Topics covered

- ✧ Dependability properties
- ✧ Sociotechnical systems
- ✧ Redundancy and diversity
- ✧ Dependable processes
- ✧ Formal methods and dependability

System dependability

- ✧ The most important system property is the dependability
- ✧ Reflect the user’s degree of trust in that system.
- ✧ Reflect the extent of the user’s confidence that it will operate as users expect.
- ✧ Cover the related attributes: reliability, availability and security.

Importance of dependability

- ✧ System failures may have widespread.
- ✧ Systems that are not dependable may be rejected.
- ✧ The costs of system failure is high if the failure leads to economic losses.
- ✧ Undependable systems may cause information loss.

Causes of failure

✧ Hardware failure

- Design and manufacturing errors.

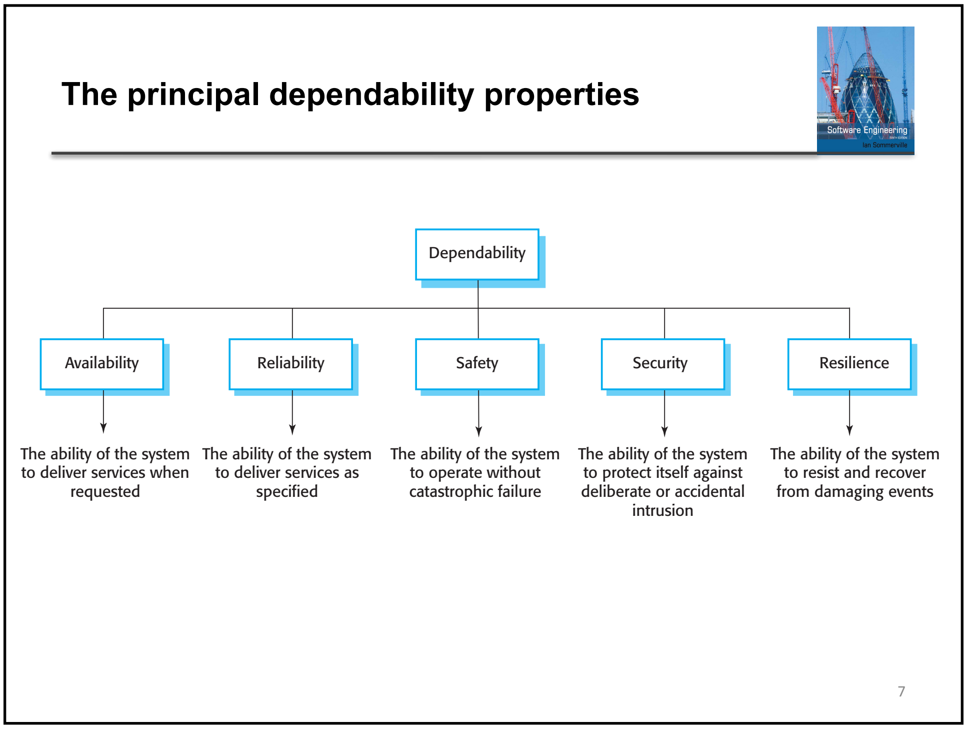
✧ Software failure

- Errors in its implementation.

✧ Operational failure

- Human operators make mistakes.

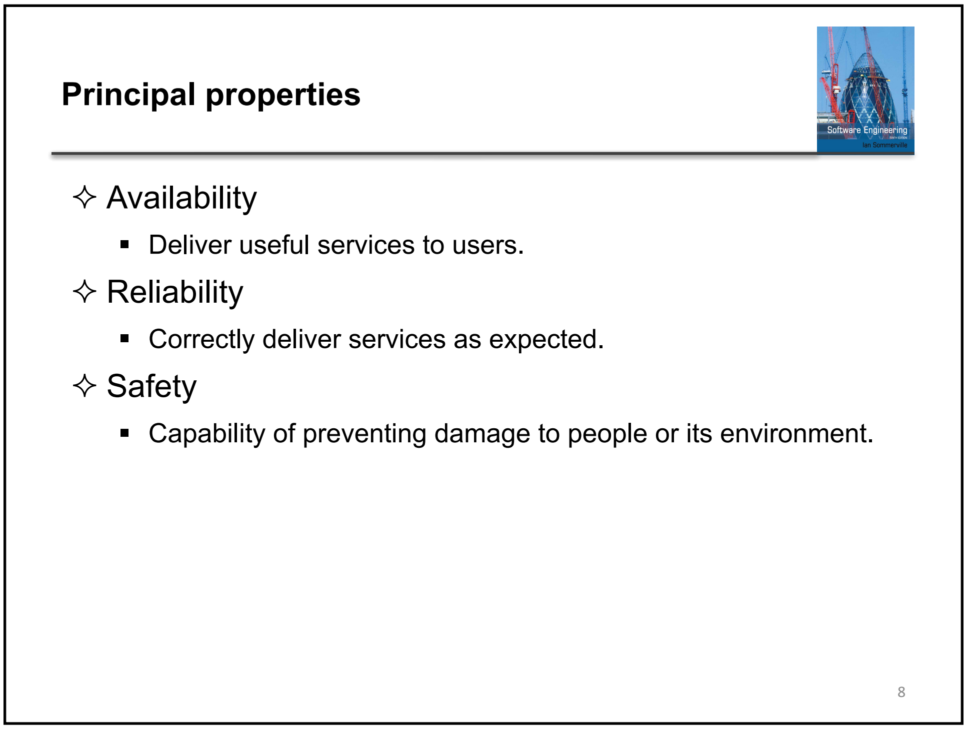
Dependability properties



The principal dependability properties

Software Engineering
Ian Sommerville

Dependability
Availability
Reliability
Safety
Security
Resilience
The ability of the system to deliver services when requested
The ability of the system to deliver services as specified
The ability of the system to operate without catastrophic failure
The ability of the system to protect itself against deliberate or accidental intrusion
The ability of the system to resist and recover from damaging events



Principal properties

Software Engineering
Ian Sommerville

- ✧ Availability
  - Deliver useful services to users.
- ✧ Reliability
  - Correctly deliver services as expected.
- ✧ Safety
  - Capability of preventing damage to people or its environment.

Principal properties

- Security
  - Capability of resisting accidental or deliberate intrusions.
- Resilience
  - A judgment of how well a system can maintain the continuity of its critical services.

Other dependability properties

- Repairability
  - Capability of being repaired in the event of a failure
- Maintainability
  - Capability of being adapted to new requirements
- Error tolerance
  - Capability to tolerate failures due to user input errors

Dependability attribute dependencies

- ✧ Depend on the system's availability and reliability.
- ✧ Corrupted data by an external attack.
- ✧ Unavailable to conduct denial of service attacks on a system.
- ✧ Malicious system virus infection and damage

Dependability achievement

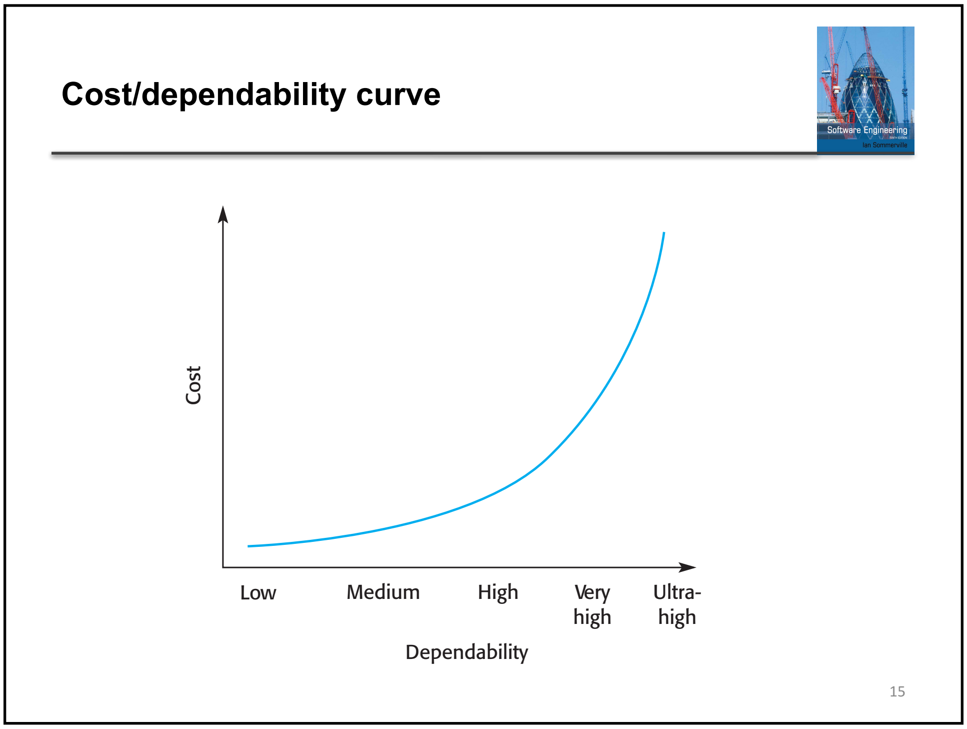
- ✧ Inspect and avoid accidental error introduction.
- ✧ Validation processes to reveal errors.
- ✧ Fault tolerant system to tolerate runtime errors.
- ✧ Protection mechanisms against external attacks.

Dependability achievement

- Correct system configuration.
- Capabilities to resist cyberattacks.
- Service recovery mechanisms after a failure.

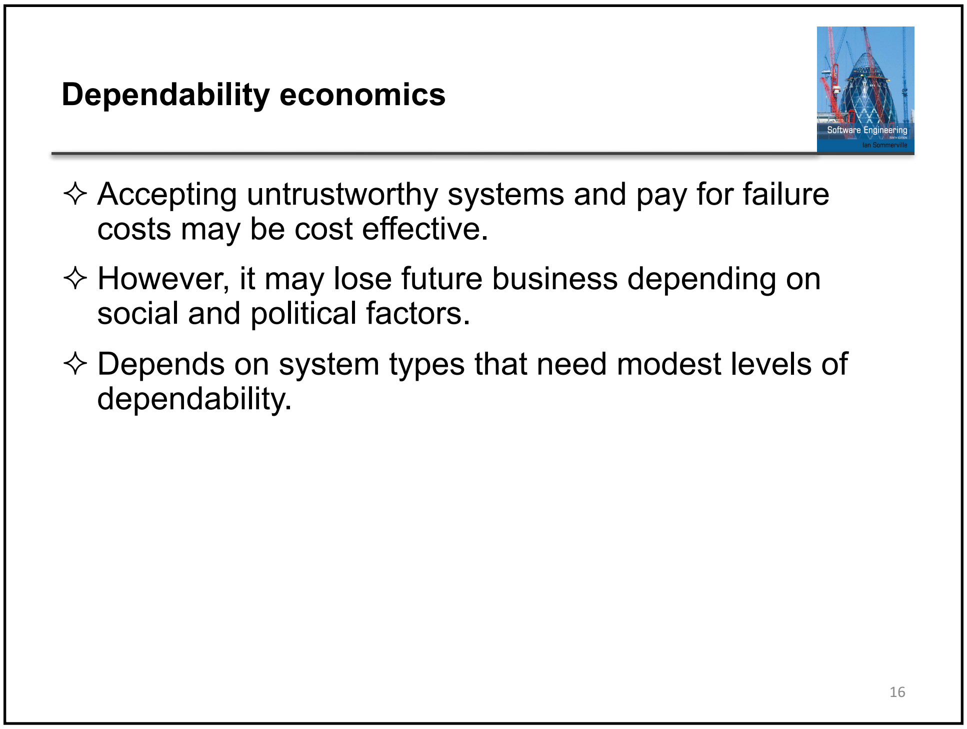
Dependability costs

- Dependability costs increase exponentially.
- There are two reasons for this
  - Expensive development techniques and hardware for higher levels of dependability.
  - Increased testing and system validation for system clients and regulators.



Cost/dependability curve

15



Dependability economics

- ✧ Accepting untrustworthy systems and pay for failure costs may be cost effective.
- ✧ However, it may lose future business depending on social and political factors.
- ✧ Depends on system types that need modest levels of dependability.

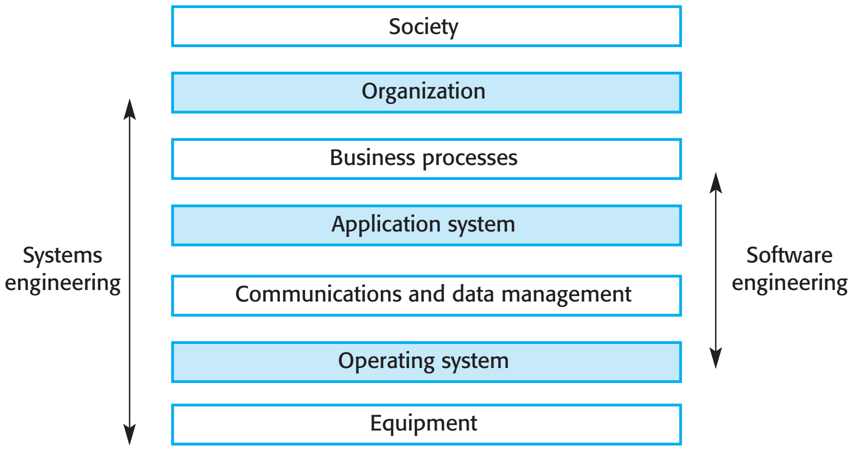
16

Sociotechnical systems (STS)

Systems and software

- ✧ Software engineering is part of system engineering process.
- ✧ Software systems are are essential components of systems based on organizational purposes.
- ✧ Example
  - The wilderness weather system is part of forecasting systems
  - Hardware and software, forecasting processes, the organizations, etc.

The sociotechnical systems (STS) stack



Layers in the STS stack

- ✧ Equipment
  - Hardware devices, including embedded systems
- ✧ Operating system
  - Common facilities for higher level applications.
- ✧ Communications and data management
  - Access to remote systems and databases.
- ✧ Application systems
  - Functionalities for specific requirements.

Layers in the STS stack

- Business processes
  - Processes involving people and systems
- Organizations
  - Business activities for system operations
- Society
  - Laws, regulation and culture

Holistic system design

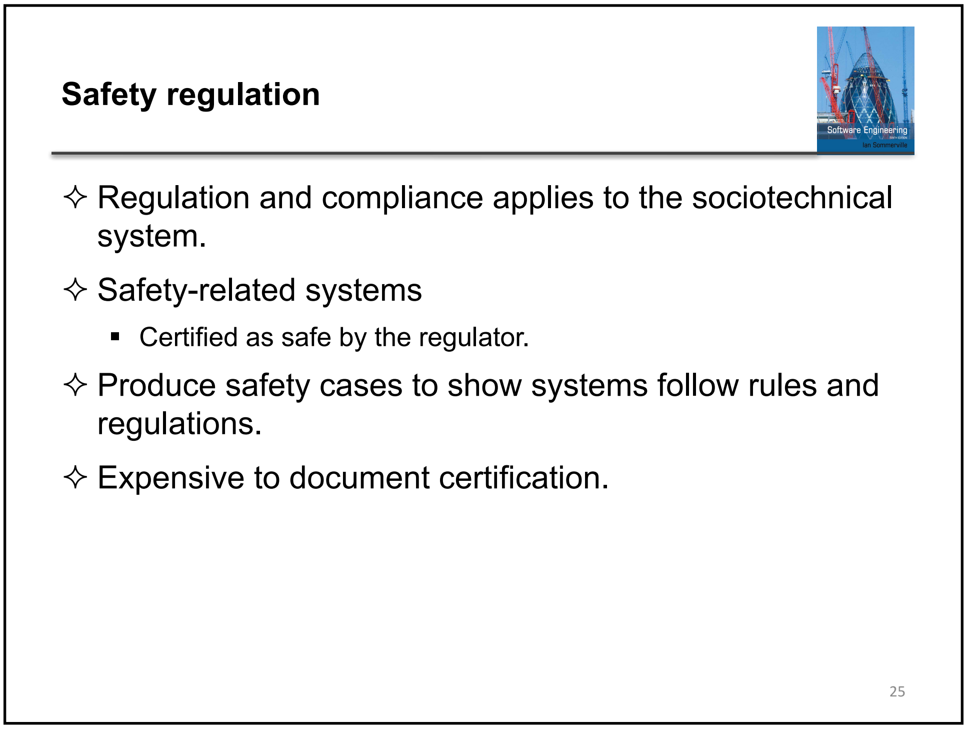
- Interactions and dependencies between system layers
  - Example: regulation changes causes changes in applications.
- For dependability, a systems perspective is essential
  - Software failures within the enclosing layers.
  - Failures in adjacent layers affects software systems.

Regulation and compliance

- ✧ The general model of economic organization
  - Universal in the world.
  - Offer goods and services and make a profit.
- ✧ Ensure the safety of their citizens
  - Follow standards to ensure that products are safe and secure.

Regulated systems

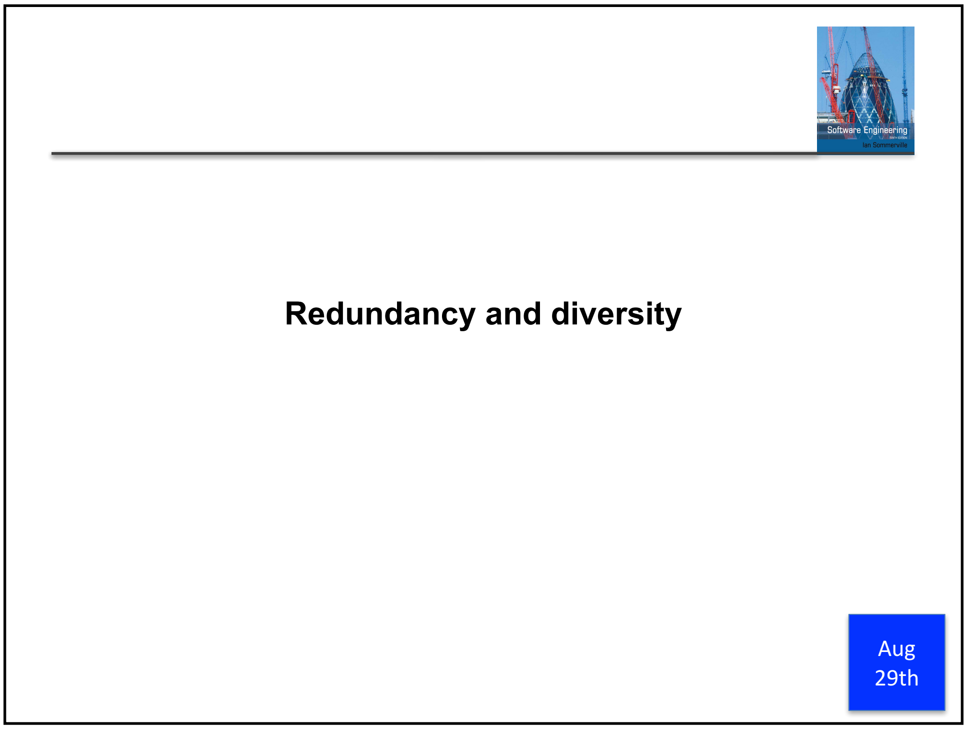
- ✧ Critical systems are regulated systems
  - Approved by an external regulator.
  - E.g., nuclear systems and air traffic control systems
- ✧ A safety and dependability case
  - Approved by the regulator.
  - Create the evidence for systems' dependability, safety and security.



Safety regulation

- ✧ Regulation and compliance applies to the sociotechnical system.
- ✧ Safety-related systems
  - Certified as safe by the regulator.
- ✧ Produce safety cases to show systems follow rules and regulations.
- ✧ Expensive to document certification.

25



Redundancy and diversity

Aug 29th

Redundancy and diversity

- ✧ Redundancy
  - Keep more than a single version.
- ✧ Diversity
  - Provide the same functionality in different mechanism.
- ✧ Redundant and diverse components should be performed independently
  - E.g., software written in different programming languages.

Diversity and redundancy examples

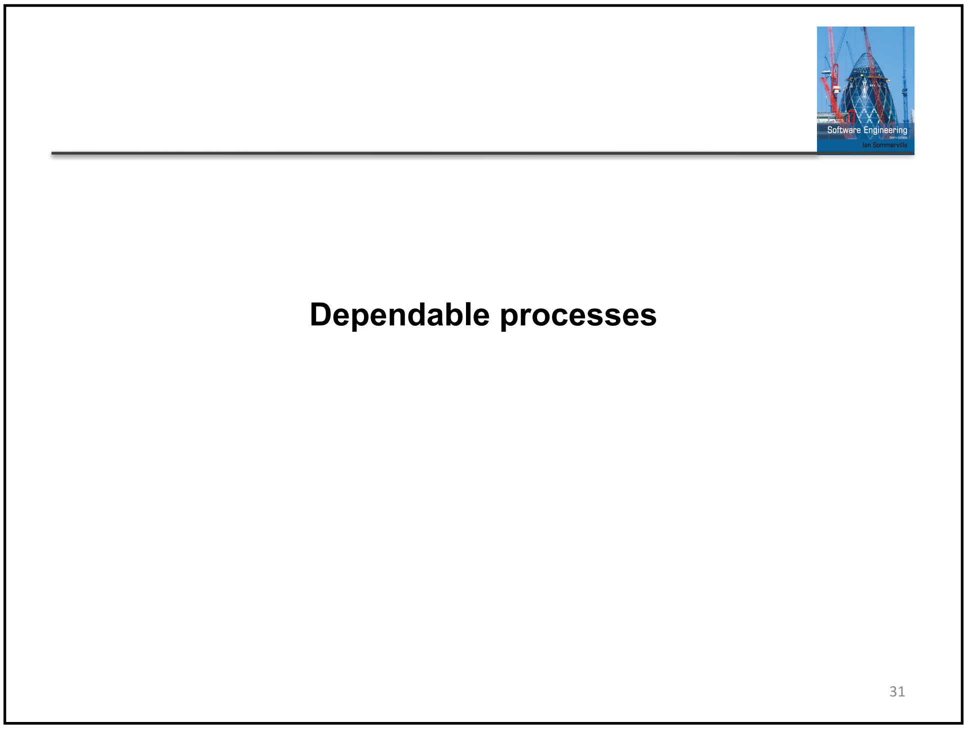
- ✧ Redundancy.
  - Backup servers to switch, when failure occurs.
- ✧ Diversity.
  - Different servers running on different operating systems.

Process diversity and redundancy

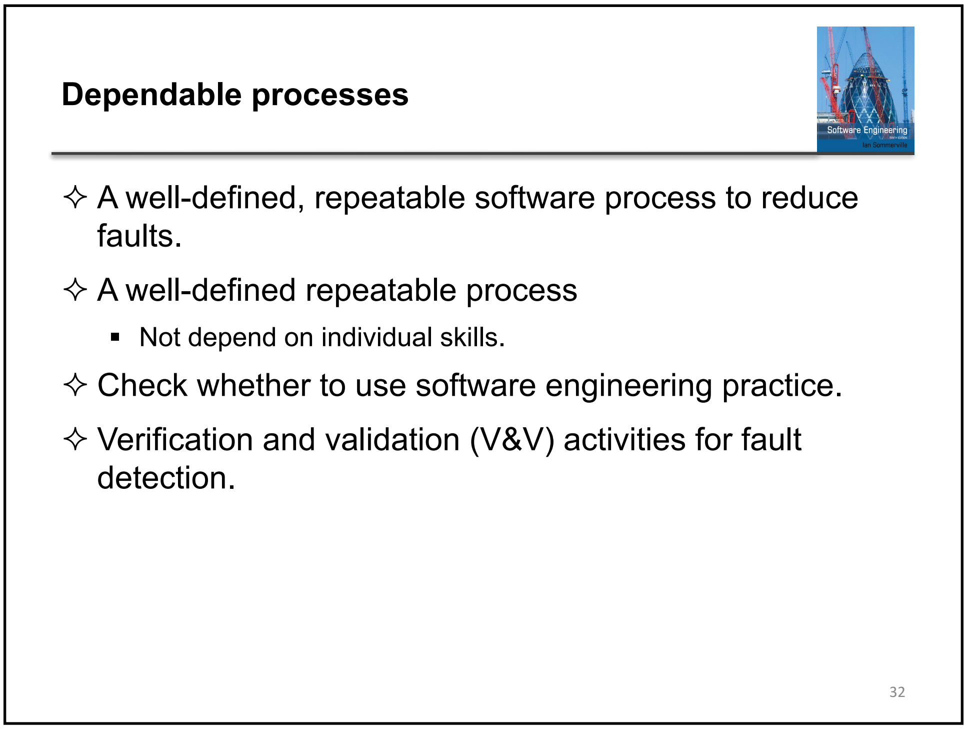
- ✧ Process activities
  - Not depend on a single approach, such as testing.
- ✧ Redundant and diverse process activities.
- ✧ Multiple process activities complement each other
  - Cross-checking techniques avoid process errors

Problems with redundancy and diversity

- ✧ Adding diversity and redundancy increases complexity.
- ✧ Increase the chances of error
  - Unanticipated interactions between redundant components.
- ✧ Advocate simplicity to decrease software dependability.
  - E.g., an Airbus product is redundant/diverse; a Boeing product has no software diversity



Dependable processes



Dependable processes

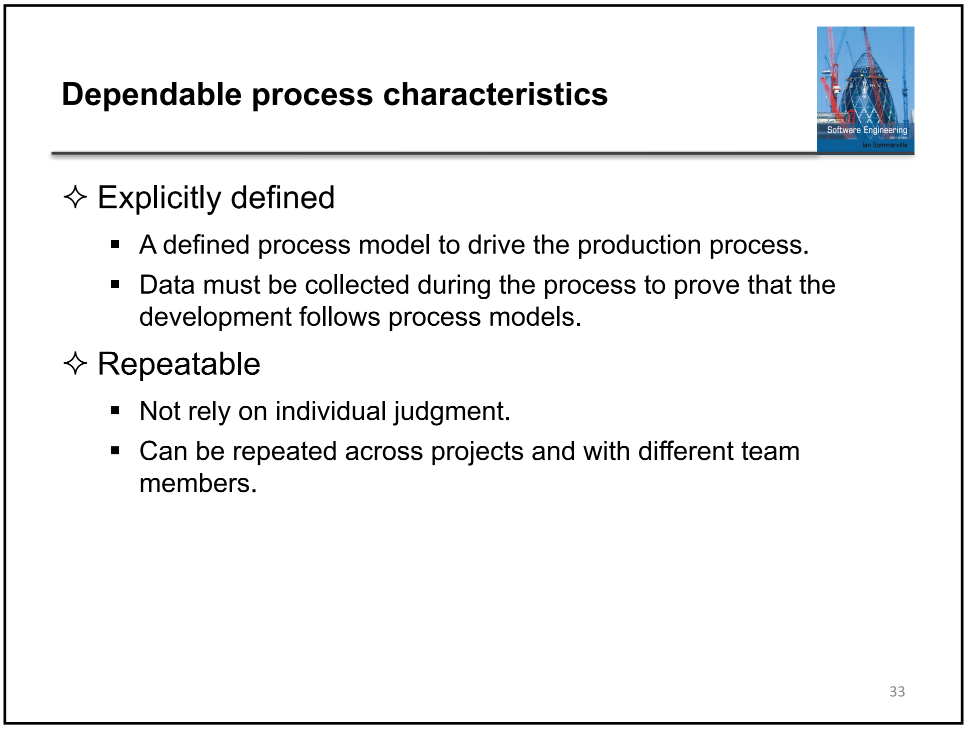
✧ A well-defined, repeatable software process to reduce faults.

✧ A well-defined repeatable process

- Not depend on individual skills.

✧ Check whether to use software engineering practice.

✧ Verification and validation (V&V) activities for fault detection.



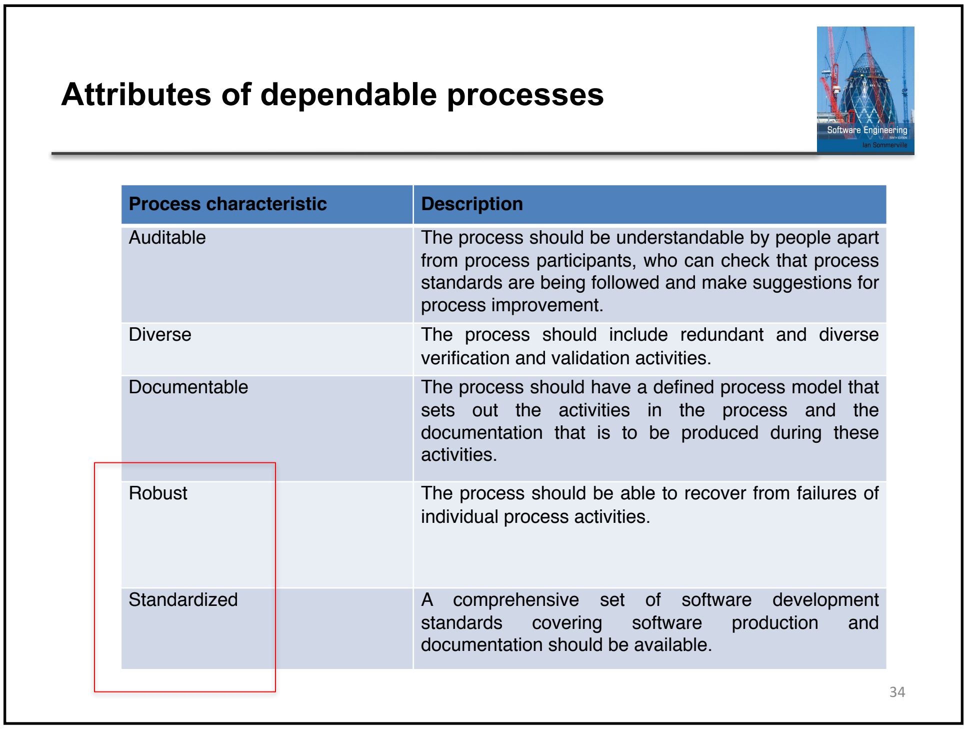
Dependable process characteristics

✧ Explicitly defined

- A defined process model to drive the production process.
- Data must be collected during the process to prove that the development follows process models.

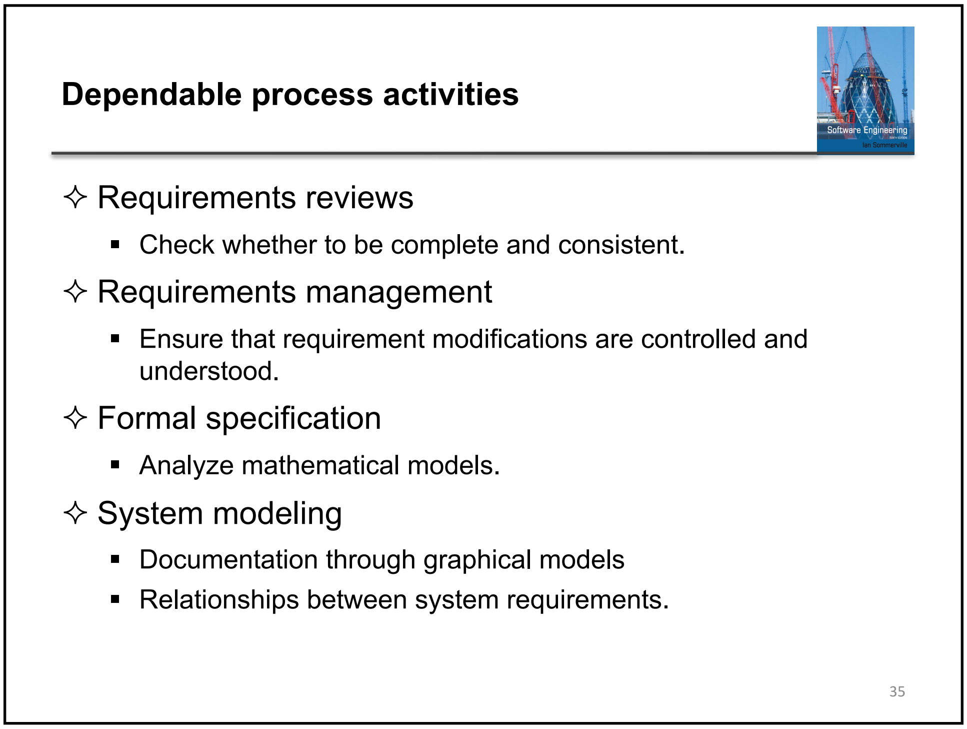
✧ Repeatable

- Not rely on individual judgment.
- Can be repeated across projects and with different team members.



Attributes of dependable processes

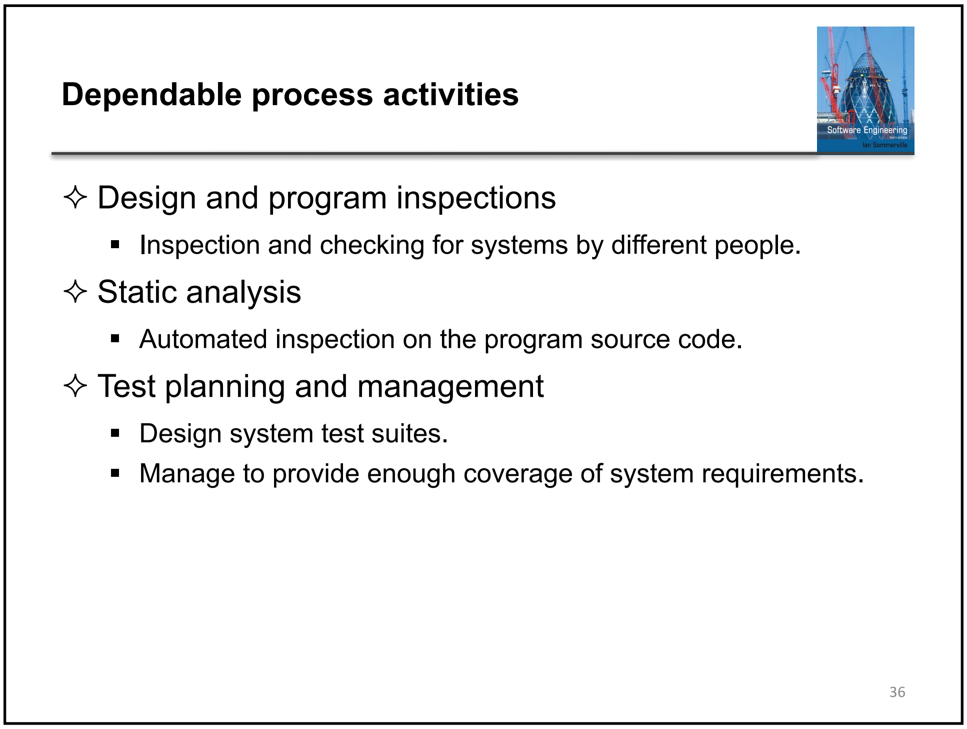
Process characteristic	Description
Auditable	The process should be understandable by people apart from process participants, who can check that process standards are being followed and make suggestions for process improvement.
Diverse	The process should include redundant and diverse verification and validation activities.
Documentable	The process should have a defined process model that sets out the activities in the process and the documentation that is to be produced during these activities.
Robust	The process should be able to recover from failures of individual process activities.
Standardized	A comprehensive set of software development standards covering software production and documentation should be available.



Dependable process activities

- Requirements reviews
  - Check whether to be complete and consistent.
- Requirements management
  - Ensure that requirement modifications are controlled and understood.
- Formal specification
  - Analyze mathematical models.
- System modeling
  - Documentation through graphical models
  - Relationships between system requirements.

35



Dependable process activities

- Design and program inspections
  - Inspection and checking for systems by different people.
- Static analysis
  - Automated inspection on the program source code.
- Test planning and management
  - Design system test suites.
  - Manage to provide enough coverage of system requirements.

36

Dependable processes and agility

✧ Produce process and product documentation.

✧ Up-front requirements analysis

- Discover requirements and requirements conflicts for system safety and security

✧ These conflict with agile development

- Minimizing documentation of system requirements

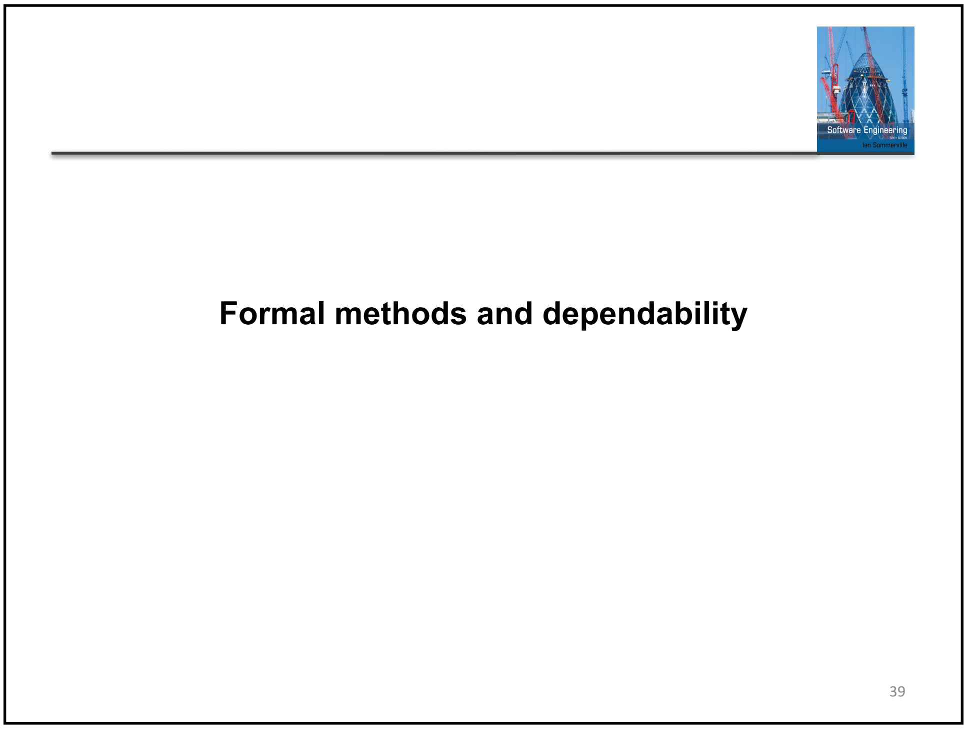
Dependable processes and agility

✧ Agile process

- iterative development, test-first development and user involvement in the development team.

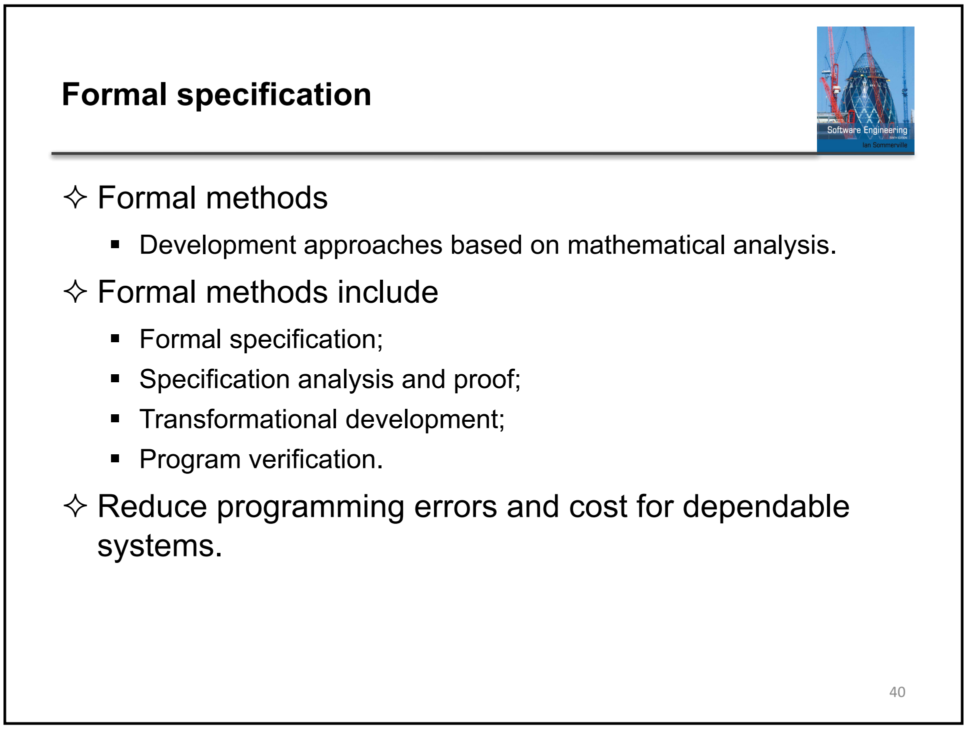
✧ Agile team follows agile process, actions, and agile methods

✧ However, ‘pure agile’ is impractical for dependable systems.



Formal methods and dependability

39



Formal specification

- ✧ Formal methods
  - Development approaches based on mathematical analysis.
- ✧ Formal methods include
  - Formal specification;
  - Specification analysis and proof;
  - Transformational development;
  - Program verification.
- ✧ Reduce programming errors and cost for dependable systems.

40

Formal approaches

- ✧ Verification-based approaches
  - Different representations of a software system are proved to be equivalent.
  - Demonstrate the absence of errors.
- ✧ Refinement-based approaches
  - A system representation is transformed into a lower-level representation.
  - Correct transformation results in equivalent representations.

Use of formal methods

- ✧ The principal benefits
  - Reduce faults or runtime errors.
- ✧ Applicable main area:
  - Dependable systems engineering.
- ✧ Cost-effective formal methods
  - Reduce high system failure costs

Classes of error

- ✧ Specification and design errors and omissions.
  - Reveal errors and omissions in requirements.
  - Models generated automatically from source code.
  - Analysis by using model checking find undesirable faults.
- ✧ Inconsistences between a specification and a program.
  - Refinement methods
  - Programmer mistakes of inconsistencies with specification
  - Discover inconsistencies between programs and specifications.

Benefits of formal specification

- ✧ Developing a formal specification
  - Analyze system requirements in detail.
  - Detect problems, inconsistencies and incompleteness.
- ✧ Specification expressed in a formal language
  - Discover inconsistencies and incompleteness.
- ✧ A formal method correctly transforms a formal specification into a program.
- ✧ Reduce program testing costs
  - Verify a program formally against its specification.

Acceptance of formal methods

✧ Formal methods limited in practical development:

- Hard to understand a formal specification
- Cannot assess if it is an accurate representation.
- Assess development costs but harder to assess the benefits.
- Unwilling to invest in formal methods.
- Unfamiliar with formal method approach.
- Difficulty in scaling up to large systems.
- Incompatibility with agile development methods.

Key points

✧ System dependability is important

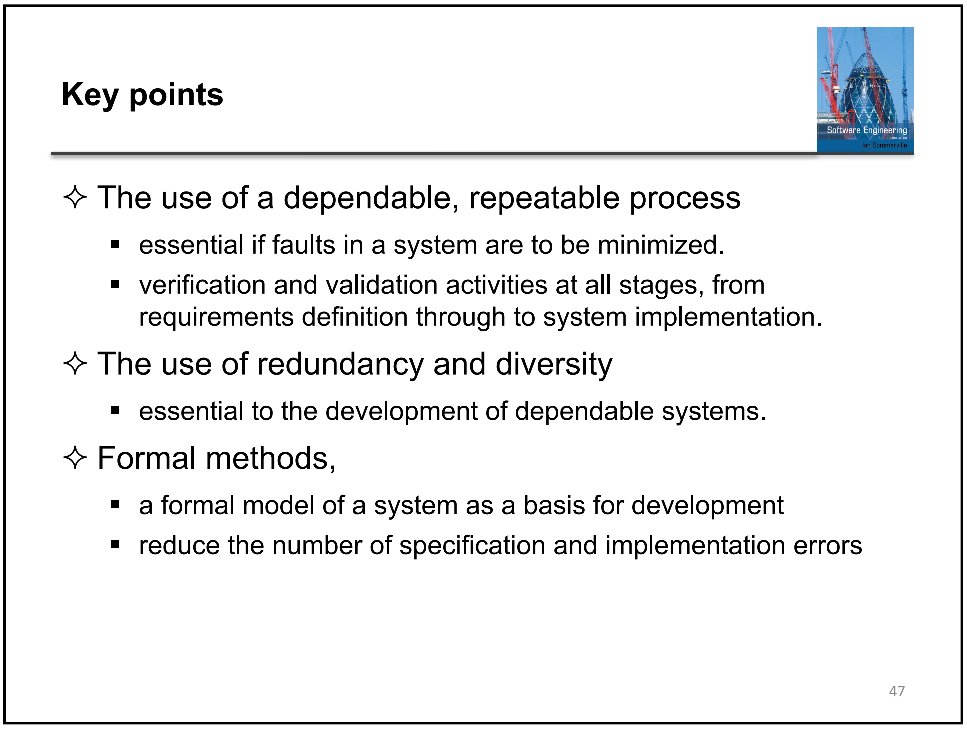
- failure of critical systems can lead to economic losses, information loss, physical damage or threats to human life.

✧ The dependability of a computer system

- a system property that reflects the user’s degree of trust in the system.
- The most important dimensions are availability, reliability, safety, security and resilience.

✧ Sociotechnical systems

- computer hardware, software and people, and are situated within an organization.
- designed to support organizational or business goals and objectives.



Key points

- ✧ The use of a dependable, repeatable process
  - essential if faults in a system are to be minimized.
  - verification and validation activities at all stages, from requirements definition through to system implementation.
- ✧ The use of redundancy and diversity
  - essential to the development of dependable systems.
- ✧ Formal methods,
  - a formal model of a system as a basis for development
  - reduce the number of specification and implementation errors

Chapter 11 – Reliability Engineering

Topics covered

- ✧ Availability and reliability
- ✧ Reliability requirements
- ✧ Fault-tolerant architectures
- ✧ Programming for reliability
- ✧ Reliability measurement

Software reliability

- ✧ Dependable software is expected
- ✧ However, some system failures are accepted.
- ✧ Software systems have high reliability requirements
  - E.g., critical software systems
- ✧ Software engineering techniques for reliability requirements.
  - E.g., medical systems and aerospace systems

Faults, errors and failures

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

Faults and failures

- ✧ Failures
  - Results of system errors resulted from faults in the system
- ✧ However, faults do not necessarily result in system errors
  - Transient and ‘corrected’ before an error arises.
  - Never be executed.
- ✧ Errors do not necessarily lead to system failures
  - Corrected by detection and recovery
  - Protected by protection facilities.

Fault management

- ✧ Fault avoidance
  - Avoid human errors to minimize system faults.
  - Organize development processes to detect and repair faults.
- ✧ Fault detection
  - Verification and validation techniques to remove faults.
- ✧ Fault tolerance
  - Design systems that faults do not cause failures.

Reliability achievement

✧ Fault avoidance

- Development technique to minimise the possibility of mistakes or reveal mistakes.

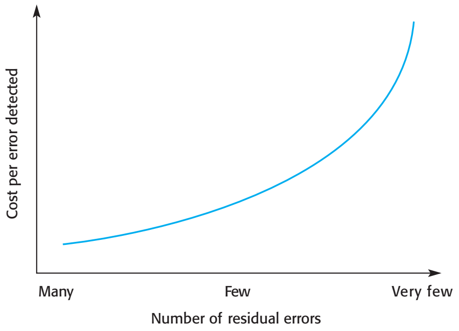
✧ Fault detection and removal

- Verification and validation techniques to increase the probability of correcting errors.

✧ Fault tolerance

- Run-time techniques to ensure that faults do not cause errors.

The increasing costs of residual fault removal



Availability and reliability

Availability and reliability

✧ Reliability

- The probability of failure-free system operation.

✧ Availability

- The probability that a system conducts requested services at a point in time.
- E.g., availability of 0.99.

Reliability and specifications

✧ Reliability

- Defined formally w.r.t. a system specification
- A deviation from a specification.

✧ Incomplete or incorrect specifications

- A system following specifications may ‘fail’.

✧ Unfamiliar with specifications

- Unaware how the system is supposed to behave.

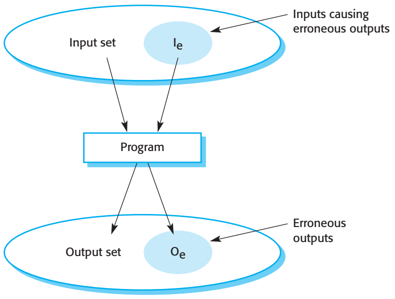
✧ Perceptions of reliability

Perceptions of reliability

✧ Not always reflect the user’s reliability perception

- The assumptions about environments for a system are incorrect
  - Different usage of a system between in an office environment and in a university environment.
- The consequences of system failures affects the perception of reliability.

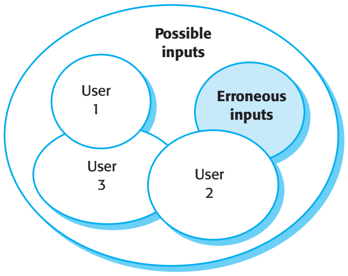
A system as an input/output mapping



Availability perception

- ✧ Expressed as a percentage of the time
  - Available to conduct services.
- ✧ Two factors not considered:
  - The number of users affected by unavailable systems.
  - The length of system failed or unavailable period.

Software usage patterns



Reliability in use

- ✧ Reliability not improved by X% by removing faults with X%
- ✧ Program defects rarely executed
  - Not encountered by users.
  - Not affect the perceived reliability.
- ✧ Users' operation patterns to avoid system features.
- ✧ Software systems with known faults
  - Considered reliable systems by users.

Reliability requirements

System reliability requirements

- ✧ Functional reliability requirements
  - Define system and software functions
  - Avoid, detect or tolerate faults
  - Not lead to system failure.
- ✧ Software reliability requirements
  - Cope with hardware failure or operator error.
- ✧ Non-functional reliability requirements
  - ✧ A measurable system attribute specified quantitatively.
  - ✧ E.g., the number of failures and the available time.

Reliability metrics

- ✧ Units of measurement of system reliability.
- ✧ Counting the number of operational failures and the period length that the system has been operational.
- ✧ Assess the reliability (e.g., critical systems)
  - Long-term measurement techniques
- ✧ Metrics
  - Probability of failure on demand
  - Rate of occurrence of failures

Probability of failure on demand (POFOD)

- ✧ The probability of the system failure when a service request is made.
  - Useful when demands for service are relatively infrequent.
- ✧ Implement appropriate protection systems
  - Demand services occasionally.
  - Serious consequence due to failed services.
- ✧ Develop for safety-critical systems
  - E.g., emergency shutdown system in a chemical plant.

Rate of fault occurrence (ROCOF)

- ✧ System failure occurrence rate
- ✧ ROCOF of 0.002
  - 2 failures are likely in each 1000 operational time units
- ✧ Reliable systems needed
  - Systems perform a number of similar requests in a short time
  - E.g., credit card processing system.
- ✧ Reciprocal of ROCOF is Mean time to Failure (MTTF)
  - Systems with long transactions
  - System processing takes a long time. MTTF should be longer than expected transaction length.

Availability

- ✧ The time that a software system is available
  - Repair and restart time considered
- ✧ Availability of 0.998
  - Software is available for 998 out of 1000 time units.
- ✧ Continuously running systems
  - E.g., railway signalling systems.

Availability specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1,440 minutes), the system will be unavailable for 144 minutes. (1440 ∗ 10%)
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period. Roughly, one minute per week.

Non-functional reliability requirements

- ✧ Non-functional reliability requirements
  - Reliability specifications using one of the reliability metrics
  - POFOD: probability of fault on demand
  - ROCOF: rate of occurrence of Fault
  - AVAIL: availability
- ✧ Used for many years in safety-critical systems
  - Uncommon for business critical systems.
- ✧ Need precise measurement about reliability and availability expectations.

Benefits of reliability specification

- Help to clarify stakeholders' needs.
- Provide a measurement basis for system tests.
- Improve the reliability by different design strategies.
- Evidence of including required reliability implementations.

Specifying reliability requirements

- Availability and reliability requirements for different types of failure.
  - Low probability of high-cost failures
- Availability and reliability requirements for different types of system service.
  - Tolerate failures in less critical services.
- A high level of reliability required.
  - Other mechanisms for reliable system services.

ATM reliability specification

- Key concerns
  - ATMs conduct services as requested
  - Record customer transactions
  - ATM systems are available when required.
- Database transaction mechanisms make a correction of transaction problems

ATM availability specification

- System services
  - The customer account database service;
  - ‘withdraw cash’, ‘provide account information’, etc.
- Specify a high level of availability in database service.
  - Database availability: 0.9999, between 7 am and 11pm.
  - A downtime of less than 1 minute per week.

ATM availability specification

- Key reliability issues depends on mechanical reliability.
- A lower level of software availability is acceptable.
- The overall availability
  - Specify availability with 0.999
  - A machine might be unavailable for between 1 and 2 minutes each day.

Insulin pump reliability specification

- Probability of failure (POFOD) metric.
- Transient failures
  - Repaired by user actions, such as, recalibration of the machine.
  - Low POFOD is acceptable. A failure occurs in every 500 demands.
- Permanent failures
  - Re-installed by the manufacturer.
  - Occur no more than once per year.
  - POFOD < 0.00002.

Functional reliability requirements

- Checking requirements
  - Identify incorrect data before it leads to a failure.
- Recovery requirements
  - Help the system recover from a failure.
- Redundancy requirements
  - Specify redundant system features.
- Process requirements
  - Specify software development processes.

Examples of functional reliability requirements

RR1: A pre-defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre-defined range. (Checking)

RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)

RR3: N-version programming shall be used to implement the braking control system. (Redundancy)

RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Fault-tolerant architectures

Fault tolerance

- Fault tolerant in critical situations.
- Fault tolerance required
  - High availability requirements
  - Failure costs are very high.
- Fault tolerance
  - Able to continue in operation despite software failures.
- Fault tolerant required against incorrect validation or specification errors, although a system is proved to conform to its specification

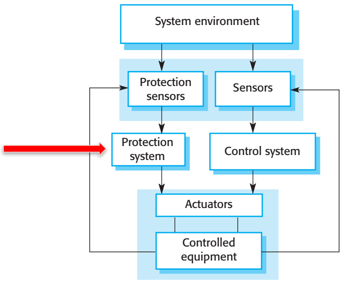
Fault-tolerant system architectures

- Fault-tolerant systems architectures
  - Fault tolerance is essential based on redundancy and diversity.
- Examples of situations for dependable architectures:
  - Flight control systems for safety of passengers
  - Reactor systems for a chemical or nuclear emergency
  - Telecommunication systems for 24/7 availability.

Protection systems

- A specialized system
  - Associated with other control system.
  - Take emergency action to deal with failures.
  - E.g., System to stop a train or system to shut down a reactor
- Monitor the controlled system and the environment.
- Take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture



Protection system functionality

✧ Protection systems for redundancy

- Control capabilities to replicate in the control software.

✧ Diverse protection systems

- Different technology used in the control software.

✧ Need to expend in validation and dependability assurance.

✧ A low probability of failure for the protection system.

Self-monitoring architectures

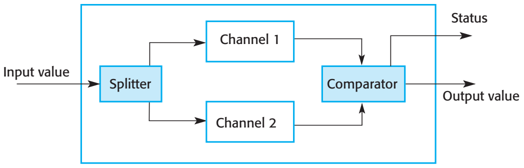
✧ Multi-channel architectures

- System monitoring its own operations
- Take action if inconsistencies are discovered

✧ The same computation is carried out on each channel

- Compare the results
- Producing identical results assumes correct system operation
- A failure exception is reported when different results arise.

Self-monitoring architecture



The first half of the chapter

Chapter 11 – Reliability Engineering

Topics covered

- ✧ Availability and reliability
- ✧ Reliability requirements
- ✧ Fault-tolerant architectures
- ✧ Programming for reliability
- ✧ Reliability measurement

Software reliability

- ✧ Dependable software is expected
- ✧ However, some system failures are accepted.
- ✧ Software systems have high reliability requirements
  - E.g., critical software systems
- ✧ Software engineering techniques for reliability requirements.
  - E.g., medical systems and aerospace systems

Faults, errors and failures

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system. For example, in the wilderness weather system, a programmer might decide that the way to compute the time for the next transmission is to add 1 hour to the current time. This works except when the transmission time is between 23.00 and midnight (midnight is 00.00 in the 24-hour clock).
System fault	A characteristic of a software system that can lead to a system error. The fault is the inclusion of the code to add 1 hour to the time of the last transmission, without a check if the time is greater than or equal to 23.00.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users. The value of transmission time is set incorrectly (to 24.XX rather than 00.XX) when the faulty code is executed.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users. No weather data is transmitted because the time is invalid.

Faults and failures

- ✧ Failures
  - Results of system errors resulted from faults in the system
- ✧ However, faults do not necessarily result in system errors
  - Transient and ‘corrected’ before an error arises.
  - Never be executed.
- ✧ Errors do not necessarily lead to system failures
  - Corrected by detection and recovery
  - Protected by protection facilities.

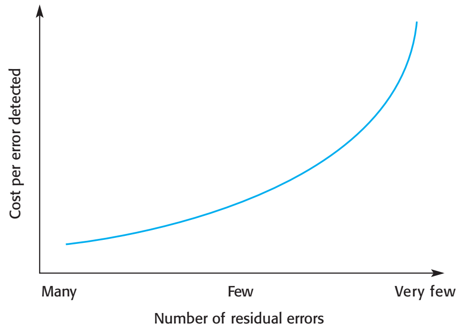
Fault management

- ✧ Fault avoidance
  - Avoid human errors to minimize system faults.
  - Organize development processes to detect and repair faults.
- ✧ Fault detection
  - Verification and validation techniques to remove faults.
- ✧ Fault tolerance
  - Design systems that faults do not cause failures.

Reliability achievement

- Fault avoidance
  - Development technique to minimise the possibility of mistakes or reveal mistakes.
- Fault detection and removal
  - Verification and validation techniques to increase the probability of correcting errors.
- Fault tolerance
  - Run-time techniques to ensure that faults do not cause errors.

The increasing costs of residual fault removal



Availability and reliability

Availability and reliability

- Reliability
  - The probability of failure-free system operation.
- Availability
  - The probability that a system conducts requested services at a point in time.

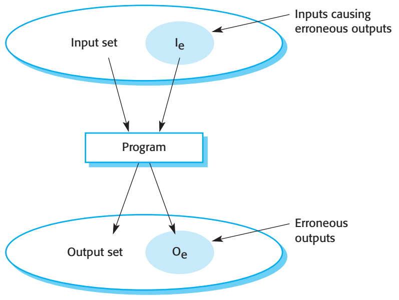
Reliability and specifications

- Reliability
  - Defined formally w.r.t. a system specification
  - A deviation from a specification.
- Incomplete or incorrect specifications
  - A system following specifications may ‘fail’.
- Unfamiliar with specifications
  - Unaware how the system is supposed to behave.
- Perceptions of reliability

Perceptions of reliability

- Not always reflect the user’s reliability perception
  - The assumptions about environments for a system are incorrect
    - Different usage of a system between in an office environment and in a university environment.
  - The consequences of system failures affects the perception of reliability.

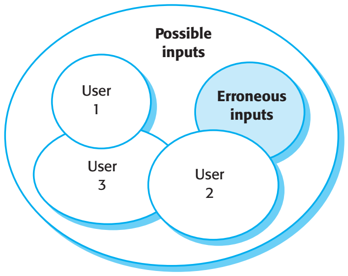
A system as an input/output mapping



Availability perception

- ✧ Expressed as a percentage of the time
  - Available to conduct services.
- ✧ Two factors not considered:
  - The number of users affected by unavailable systems.
  - The length of system failed or unavailable period.

Software usage patterns



Reliability in use

- ✧ Reliability not improved by X% by removing faults with X%
- ✧ Program defects rarely executed
  - Not encountered by users.
  - Not affect the perceived reliability.
- ✧ Users' operation patterns to avoid system features.
- ✧ Software systems with known faults
  - Considered reliable systems by users.

Reliability requirements

System reliability requirements

- ✧ Functional reliability requirements
  - Define system and software functions
  - Avoid, detect or tolerate faults
  - Not lead to system failure.
- ✧ Software reliability requirements
  - Cope with hardware failure or operator error.
- ✧ Non-functional reliability requirements
  - ✧ A measurable system attribute specified quantitatively.
  - ✧ E.g., the number of failures and the available time.

Reliability metrics

- Units of measurement of system reliability.
- Counting the number of operational failures and the period length that the system has been operational.
- Assess the reliability (e.g., critical systems)
  - Long-term measurement techniques
- Metrics
  - Probability of failure on demand
  - Rate of occurrence of failures

Probability of failure on demand (POFOD)

- The probability of the system failure when a service request is made.
  - Useful when demands for service are relatively infrequent.
- Implement appropriate protection systems
  - Demand services occasionally.
  - Serious consequence due to failed services.
- Develop for safety-critical systems
  - E.g., emergency shutdown system in a chemical plant.

Rate of fault occurrence (ROCOF)

- System failure occurrence rate
- ROCOF of 0.002
  - 2 failures are likely in each 1000 operational time units
- Reliable systems needed
  - Systems perform a number of similar requests in a short time
  - E.g., credit card processing system.
- Reciprocal of ROCOF is Mean time to Failure (MTTF)
  - Systems with long transactions
  - System processing takes a long time. MTTF is longer than expected transaction length.

Availability

- The time that a software system is available
  - Repair and restart time considered
- Availability of 0.001
  - Software is available for 1 out of 1000 time units.
- Continuously running systems
  - E.g., railway signalling systems.

Availability specification

Availability	Explanation
0.9	The system is available for 90% of the time. This means that, in a 24-hour period (1,440 minutes), the system will be unavailable for 144 minutes. (1440 ∗ 10%)
0.99	In a 24-hour period, the system is unavailable for 14.4 minutes.
0.999	The system is unavailable for 84 seconds in a 24-hour period.
0.9999	The system is unavailable for 8.4 seconds in a 24-hour period. Roughly, one minute per week.

Non-functional reliability requirements

- Non-functional reliability requirements
  - Reliability specifications using one of the reliability metrics (POFOD, ROCOF or AVAIL).
- Used for many years in safety-critical systems
  - Uncommon for business critical systems.
- Need precise measurement about reliability and availability expectations.

Benefits of reliability specification

- ✧ Help to clarify stakeholders' needs.
- ✧ Provide a measurement basis for system tests.
- ✧ Improve the reliability by different design strategies.
- ✧ Evidence of including required reliability implementations.

Specifying reliability requirements

- ✧ Availability and reliability requirements for different types of failure.
  - Low probability of high-cost failures
- ✧ Availability and reliability requirements for different types of system service.
  - Tolerate failures in less critical services.
- ✧ A high level of reliability required.
  - Other mechanisms for reliable system services.

ATM reliability specification

- ✧ Key concerns
  - ATMs conduct services as requested
  - Record customer transactions
  - ATM systems are available when required.
- ✧ Database transaction mechanisms make a correction of transaction problems

ATM availability specification

- ✧ System services
  - The customer account database service;
  - ‘withdraw cash’, ‘provide account information’, etc.
- ✧ Specify a high level of availability in database service.
  - Database availability: 0.9999, between 7 am and 11pm.
  - A downtime of less than 1 minute per week.

ATM availability specification

- ✧ Key reliability issues depends on mechanical reliability.
- ✧ A lower level of software availability is acceptable.
- ✧ The overall availability
  - Specify availability with 0.999
  - A machine might be unavailable for between 1 and 2 minutes each day.

Insulin pump reliability specification

- ✧ Probability of failure (POFOD) metric.
- ✧ Transient failures
  - Repaired by user actions, such as, recalibration of the machine.
  - Low POFOD is acceptable. A failure occurs in every 500 demands.
- ✧ Permanent failures
  - Re-installed by the manufacturer.
  - Occur no more than once per year.
  - POFOD < 0.00002.

Functional reliability requirements

✧ Checking requirements
- Identify incorrect data before it leads to a failure.

✧ Recovery requirements
- Help the system recover from a failure.

✧ Redundancy requirements
- Specify redundant system features.

✧ Process requirements
- Specify software development processes.

Examples of functional reliability requirements

RR1: A pre-defined range for all operator inputs shall be defined and the system shall check that all operator inputs fall within this pre-defined range. (Checking)

RR2: Copies of the patient database shall be maintained on two separate servers that are not housed in the same building. (Recovery, redundancy)

RR3: N-version programming shall be used to implement the braking control system. (Redundancy)

RR4: The system must be implemented in a safe subset of Ada and checked using static analysis. (Process)

Fault-tolerant architectures

Fault tolerance

✧ Fault tolerant in critical situations.

✧ Fault tolerance required
- High availability requirements
- Failure costs are very high.

✧ Fault tolerance
- Able to continue in operation despite software failures.

✧ Fault tolerant required against incorrect validation or specification errors, although a system is proved to conform to its specification

Fault-tolerant system architectures

✧ Fault-tolerant systems architectures
- Fault tolerance is essential based on redundancy and diversity.

✧ Examples of situations for dependable architectures:
- Flight control systems for safety of passengers
- Reactor systems for a chemical or nuclear emergency
- Telecommunication systemsfor 24/7 availability.

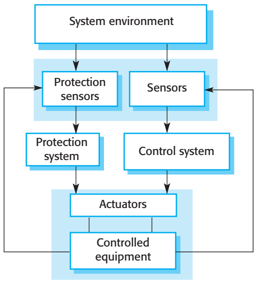
Protection systems

✧ A specialized system
- Associated with other control system.
- Take emergency action to deal with failures.
- E.g., System to stop a train or system to shut down a reactor

✧ Monitor the controlled system and the environment.

✧ Take emergency action to shut down the system and avoid a catastrophe.

Protection system architecture



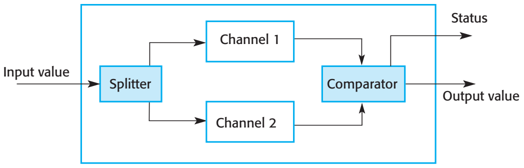
Protection system functionality

- ✧ Protection systems for redundancy
  - Control capabilities to replicate in the control software.
- ✧ Diverse protection systems
  - Different technology used in the control software.
- ✧ Need to expend in validation and dependability assurance.
- ✧ A low probability of failure for the protection system.

Self-monitoring architectures

- ✧ Multi-channel architectures
  - System monitoring its own operations
  - Take action if inconsistencies are discovered
- ✧ The same computation is carried out on each channel
  - Compare the results
  - Producing identical results assumes correct system operation
  - A failure exception is reported when different results arise.

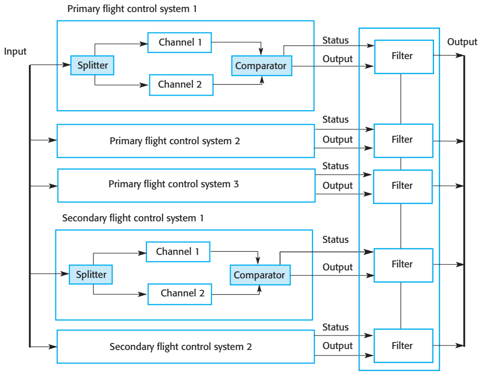
Self-monitoring architecture



Self-monitoring systems

- ✧ Diverse hardware systems on each channel
  - Prevent common failures producing the same results.
- ✧ Diverse software applications in each channel
  - Prevent same errors affecting each channel.
- ✧ Several self-checking systems in parallel.
  - High-availability required.
  - E.g., Airbus aircraft's flight control systems.

Airbus flight control system architecture


Primary flight control system 1
Input
Splitter
Channel 1
Channel 2
Comparator
Status
Output
Filter
Output
Primary flight control system 2
Primary flight control system 3
Secondary flight control system 1
Secondary flight control system 2

Airbus architecture discussion

- ✧ The Airbus FCS has 5 separate computers
  - Each system is able to perform the control software.
- ✧ Extensive diverse systems between primary and secondary systems:
  - Different processors
  - Different chipsets from different manufacturers.
  - Different complexity -- only critical functionality in secondary.
  - Different programming languages by different teams.

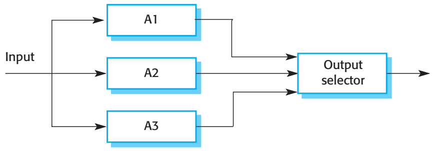
N-version programming

- ✧ Multiple versions of a program execute computations.
  - Odd number of computers involved, e.g., three versions.
- ✧ The results are compared using a voting system.
- ✧ The correct result is determine by the majority result.
- ✧ The notion of triple-modular redundancy, as used in hardware systems.

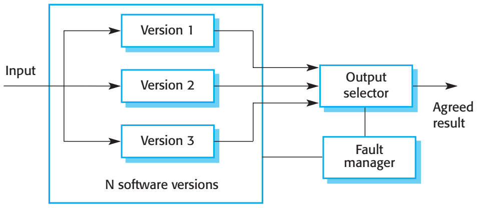
Hardware fault tolerance

- ✧ Triple-modular redundancy (TMR).
- ✧ Three replicated identical components
  - Receive the same input and their outputs are compared.
- ✧ One different output
  - Ignored based on the assumption of component failure.
- ✧ Most faults caused by component failures
  - A low probability of simultaneous component failure.

Triple modular redundancy (TMR)



N-version programming



N-version programming

- ✧ The different versions of a system
  - Designed and implemented by different teams.
- ✧ Assuming a low probability of making same mistakes
  - Different algorithms used
- ✧ Empirical evidence
  - Commonly misinterpret specifications
  - Use same algorithms in different systems.

Software diversity

✧ Fault tolerance depend on software diversity

✧ Assume that different implementations fail differently

- Independent implementations
- Uncommon errors

✧ Strategies

- Different programming languages
- Different design methods and tools
- Explicit specification of different algorithms

Problems with design diversity

✧ Tend to solve problems using same methods

✧ Characteristic errors

- Different teams making same mistakes. Making mistakes in same parts.
- Specification errors propagated to all implementations

Specification dependency

✧ Software redundancy susceptible to specification errors.

- Incorrect specification causes system failures.

✧ Complex software specifications

- Hard to perform validation and verification.

✧ Developing separate software specifications.

Improvements in practice

✧ Multi-version programming leads to significant improvements in reliability and availability.

- Diversity and independence

✧ In practice, observed improvements are much less significant

- Reliability improvements of between 5 and 9 times.

✧ Considerable costs to develop multi-versions of systems.

Programming for reliability

Dependable programming

✧ Standard programming practices

- Reduce program fault introduction rate.

✧ Support fault avoidance, detection and tolerance

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**

```java
ProcessBuilder pb =
 new ProcessBuilder("external-program");
Timer t = new Timer();
Process p = pb.start();

TimerTask killer =
  new TimeoutProcessKiller(p);
t.schedule(killer, 5000);
```

```java
class TimeoutProcessKiller extends TimerTask {
  Process p;
  public TimeoutProcessKiller(Process p) {
    this.p = p;
  }
  @Override
  public void run() { p.destroy(); }
}
```

(1) Limit the visibility of information in a program

✧ Limited access to data for their implementation.

✧ Reduce possibilities of accidental corruption of program state by other components

✧ Control visibility by using abstract data types

- Private data representation.
- Limited access to data through predefined operations.

(2) Check all inputs for validity

✧ Taking inputs from their environment based on assumptions about the inputs.

✧ Program specifications rarely defined

- Inconsistent inputs with the assumptions.

✧ Unpredictable program behavior

- Unusual inputs
- Threats to the security of the system.

✧ Check program inputs before processing

- Considering the assumptions about the inputs.

Validity checks

✧ Range checks

- Check the input's range.

✧ Size checks

- Check the input's maximum or minimum size.

✧ Representation checks

- Check the input's expression for the representation
- E.g. names do not include numerals.

✧ Reasonableness checks

- Check the input's logical information.
- E.g., reasonable rather than an extreme value.

Example

```java
    try {
      int a[]= {1, 2, 3, 4};
      for (int i = 1; i <= SIZE; i++) {
        System.out.println ("a[" + i + "]=" + a[i] + "\n");
      }
    }
    catch (Exception e) {
      System.out.println ("error = " + e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println ("ArrayIndexOutOfBoundsException");
    }
```

```
A.java: error: exception ArrayIndexOutOfBoundsException has already been caught
           catch (ArrayIndexOutOfBoundsException e)

^ 1 error
```

(3) Provide a handler for all exceptions

✧ An error or some unexpected event

- E.g., a power failure.

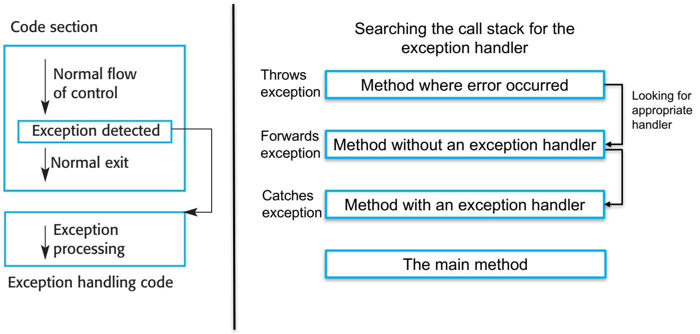
✧ Exception handling constructs

- Responding and handling exception events
- Change the program execution flow

✧ Using normal control constructs to handle exceptions?

- A number of additional statements
- Significant overhead
- Tedious and error-prone

Exception handling



Code section

Normal flow of control

Exception detected

Normal exit

Exception processing

Exception handling code

Searching the call stack for the exception handler

Throws exception

Method where error occurred

Looking for appropriate handler

Forwards exception

Method without an exception handler

Catches exception

Method with an exception handler

The main method

Exception handling

- ✧ Three possible exception handling strategies
  - Report exceptions and provide the related information.
  - Conduct alternative processes
    - The related information required to recover from the problem.
  - Pass control to a run-time support system.
- ✧ A mechanism to provide fault tolerance
  - Recovering from fault-caused errors
  - Eliminating faults

(4) Minimize the use of error-prone constructs

- ✧ Human error of misunderstanding or losing track of the relationships between the different parts of the system
- ✧ Error-prone constructs in programming languages
  - Inherently complex
- ✧ Avoid or minimize the use of error-prone constructs.

Error-prone constructs

- ✧ Unconditional branch (goto) statements
- ✧ Floating-point numbers
  - Inherently imprecise. The imprecision may lead to invalid comparisons.
  - E.g., 7.00000000 (6.99999999 or 7.00000001)
- ✧ Pointers
  - Pointers referring to the wrong memory areas can corrupt data. Aliasing can make programs difficult to understand and change.
- ✧ Dynamic memory allocation
  - Run-time allocation can cause memory overflow.

Common Mistakes in Dynamic Memory Allocation

- ✧ No matter how much we try, it is very difficult to free all dynamically allocated memory. Even if we can do that, it is often not safe from exceptions.
- ✧ If an exception is thrown, the "a" object is never deleted.

```
void SomeMethod() {
  ClassA *a = new ClassA;

  // it can throw an exception
  foo();

  delete a;
}
```

- ✧ Detect memory leaks by Valgrind

Error-prone constructs

- ✧ Parallelism
  - Can result in subtle timing errors because of unforeseen interaction between parallel processes.
- ✧ Recursion
  - Errors in recursion can cause memory overflow as the program stack fills up.
- ✧ Interrupts
  - Interrupts can cause a critical operation to be terminated and make a program difficult to understand.
- ✧ Inheritance
  - Code is not localised. This can result in unexpected behaviour when changes are made and problems of understanding the code.

Common Mistake in Inheritance and Dynamic Binding

```
abstract class Base {
   int f = 0;
   public int m1() {
      return 0;
   }
}

class ClassA extends Base {
   int f = 1;

   public int m1() {
      return f;
   }
}

class ClassB extends Base {
   public int m1() {
      return f;
   }
}

class ClassC extends ClassB {
   int f = 2;

   public int m1() {
      return f;
   }
}
```

```
class Main {
  Base x1;

  void thread1() {
     x1 = new ClassA();
     System.out.println(x1.m1());
  }

  void thread2() {
     x1 = new ClassB();
     System.out.println(x1.m1());
  }

  void thread3() {
     x1 = new ClassC();
     System.out.println(x1.m1());
  }
}
```

Inheritance when combined with dynamic binding can cause timing problems at run-time.

Error-prone constructs

✧ Aliasing

- Using more than 1 name to refer to the same state variable.

✧ Unbounded arrays

- Buffer overflow failures can occur if no bound checking on arrays.

✧ Default input processing

- Occur irrespective of the input.
- The default action changes the program control flow.
- Malicious inputs trigger a program failure.

Common Errors in Default Input Processing

✧ Unexpectedly execute a call to method foo(int) with default input '0', instead of a call to method foo(int, int).

- Method overloading.
- Integer '0' is a default value for argument 'b' in the overloaded method foo(int).

```
   void foo(int a, int b) {
      // the implementation goes here
   }

   void foo(int a) {
      foo(a, 0);
   }
```

(5) Provide restart capabilities

✧ The system restart capability

- Preserve the program data or status.
- Long transactions or user interactions.

✧ Restart scenarios

- Web applications keeping copies of forms that users fill in before there is a problem
- Text editors restarting from the checkpoint saving periodically the program state or memory

(6) Check array bounds

✧ Address a memory location outside of the range of an array declaration.

✧ Bounded buffer vulnerability

- E.g., Writing executable code into memory by deliberately writing beyond the top element in an array.

✧ Bound checking for an array access

- Within the bounds of the array.

Common Errors in Bounded Buffer Vulnerability

✧ Writing data past the end of allocated memory can be detected by OS

- Generate a segmentation fault error that terminates the process.

```
char str[8] = "";             // 8-byte-long string buffer
unsigned short year = 2017; // two-byte integer
```

var	str	year
value	[null string]	2017

```
strcpy(str, "boundless");   // boundless" 9 characters long
```

var	str								year
value	'b'	'o'	'u'	'n'	'd'	'l'	'e'	's'	????

Common Errors in Bounded Buffer Vulnerability

- ✧ Return a heap-allocated copy of the string with all uppercase letters.
- ✧ No bounds checking on its input.
- ✧ Overflow buf with the unbounded call to strcpy().

```
char *lccopy(const char *str) {
  char buf[BUFSIZE];
  char *p;

  strcpy(buf, str);
  for (p = buf; *p; p++) {
    if (isupper(*p)) {
      *p = tolower(*p);
    }
  }
  return strdup(buf);
}
```

(7) Include timeouts when calling external components

- ✧ May never receive services from failed systems
  - No indication of a failure.
  - Failure of a remote computer can be 'silent' In a distributed system.
- ✧ Set timeouts on all calls to external components.
- ✧ Assume failure and take actions to recover from errors
  - After a defined time period without a response.

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

- ✧ Use constants reflecting real-world values names
  - Do not use numeric values and always refer to them by name.
- ✧ Reduce mistakes for wrong values by using a name rather than a value.
- ✧ Changing constant values
  - Easy to maintain and localize edit locations to make the change.

Constant Interface Pattern

- ✧ Use final class for Constants
- ✧ Declare public static final and static import all constants

```
public final class Constants {
   private Constants() {
      // restrict instantiation
   }

   public static final double PI              = 3.14159;
   public static final double PLANCK_CONSTANT = 6.62606896e-34;
}
```

```
import static math.Constants.PLANCK_CONSTANT;
import static math.Constants.PI;

class Calculations {
   public double getReducedPlanckConstant() {
      return PLANCK_CONSTANT / (2 * PI);
   }
}
```

Reliability measurement

Reliability measurement

- ✧ Collect data about system operation:
- ✧ The number of failures for system service requests
  - Probability of failure on demand (POFOD)
- ✧ The time or the number of transactions between system failures plus the total elapsed time or total number of transactions
  - Mean time to failure (MTTF)
  - Rate of occurrence of fault (ROCOF)
- ✧ The repair or restart time after a system failure
  - Availability
  - The time between failures
  - The time required to restore the system from failures

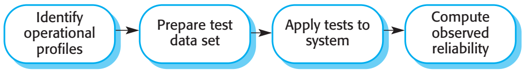
Reliability testing

- ✧ Reliability testing (Statistical testing)
  - Assess whether a system reaches the required level of reliability.
- ✧ Not part of a defect testing process
  - Datasets for defect testing do not include actual usage data.
- ✧ Data sets required to replicate actual inputs to be processed.

Statistical testing

- ✧ Testing software for reliability rather than fault detection.
- ✧ Measuring the number of errors
  - Predict reliability of the software
  - More errors, compared to the one in the specification.
- ✧ An acceptable level of reliability
  - Test software systems and repair or improve them until systems reach the level of reliability.

Reliability measurement



Identify operational profiles → Prepare test data set → Apply tests to system → Compute observed reliability

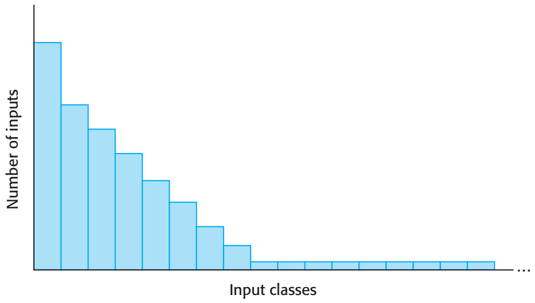
Reliability measurement problems

- ✧ Operational profile uncertainty
  - Inaccurate operational profile that does not reflect the real use of the system.
- ✧ High costs of test data generation
  - Expensive costs to generate test datasets for the system.
- ✧ Statistical uncertainty
  - A statistically significant number of failures for computation.
  - Highly reliable systems rarely fail.
- ✧ Recognizing failure
  - Conflicting interpretations of a specification about unobvious failures.

Operational profiles

- ✧ A set of test data
  - Frequency matches the actual frequency from ‘normal’ usage of the system.
  - The number of times the failure event occurred.
- ✧ A close match with actual usage
  - The measured reliability
  - Reflect the actual usage of the system.
- ✧ Generate from real data
  - Collect from an existing system
  - Assumption of the usage pattern of a system

An operational profile



Operational profile generation

- Automatic data generation, if possible.
  - Difficult for interactive systems.
  - Easy for ‘normal’ inputs
- Difficult to generate ‘unlikely’ inputs (anomalies) and test data for these anomalies.
- Unknown usage pattern of new systems.
- Changeable operational profiles
  - Non-static but dynamic
  - E.g., learn about new systems, changing usage patterns.

Key points

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment and by including fault tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification.
- Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF) and availability (AVAIL).

Key points

- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.
- Dependable system architectures are system architectures that are designed for fault tolerance.
- There are a number of architectural styles that support fault tolerance including protection systems, self-monitoring architectures and N-version programming.

Key points

- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.

Chapter 12 – Safety Engineering

Topics covered

- Safety-critical systems
- Safety requirements
- Safety engineering processes
- Safety cases

Safety

- A property of a system
- The system's ability to operate services
  - Prevent danger causing human injury or death
  - Avoiding damage to the system's environment.
- Software safety issues become important
  - Most devices incorporate software-based control systems.
  - Control real-time, safety-critical processes.

Software in safety-critical systems

- Software-controlled systems
  - Decisions are made by the software.
  - Subsequent actions are safety-critical.
  - Software behaviour is related to safety of the system.
- Checking and monitoring safety-critical components
  - E.g., monitoring aircraft engine components for fault detection.
- Monitoring software is safety-critical
  - Other components may fail due to the failure of fault detection.

Safety and reliability

- Safety and reliability
  - Reliability and availability are not sufficient for system safety
- Reliability
  - Conformance to a given specification and delivery of service
- Safety
  - Ensuring system cannot cause damage.
- System reliability is essential for safety
  - However, reliable systems can be unsafe

Unsafe reliable systems

- Dormant system faults
  - Undetected for a number of years and only rarely arise.
- Specification errors
  - Software system behaves as specified but cause an accident.
- Hardware failures at runtime
  - E.g., generating spurious inputs
  - Hard to anticipate in the specification
- Context-sensitive commands
  - E.g., a system command is executed at the wrong time.

Safety-critical systems

Safety critical systems

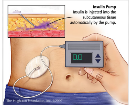
- ✧ Essential that system operation is always safe
  - Must not cause damage to people or the system's environment
- ✧ Examples
  - Process control systems in chemical manufacture
  - Automobile control systems such as braking management systems

Safety criticality

- ✧ Primary safety-critical systems
  - Embedded software systems
  - Cause associated hardware failures, directly threatening people.
  - E.g., the insulin pump control system.
- ✧ Secondary safety-critical systems
  - Result in faults in other connected systems, affecting safety consequences.
  - E.g., the Mentcare system producing inappropriate treatment being prescribed.
  - Infrastructure control systems.

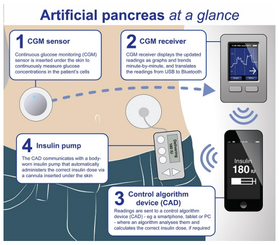
Insulin pump control system

- ✧ Collects data from a blood sugar sensor and calculates the amount of insulin required to be injected.
- ✧ Calculation based on the rate of change of blood sugar levels.


Insulin Pump
Insulin is injected into the subcutaneous tissue automatically by the pump.
0.8
The Hughston Foundation, Inc. ©2007

Insulin pump control system (cont.)

- ✧ Sends signals to a micro-pump to deliver the correct dose of insulin.
- ✧ Safety-critical system as low blood sugars can lead to brain malfunctioning, coma and death; high-blood sugar levels have long-term consequences such as eye and kidney damage.


Artificial pancreas *at a glance*
1 CGM sensor
Continuous glucose monitoring (CGM) sensor is inserted under the skin to continuously measure glucose concentrations in the patient's cells
2 CGM receiver
CGM receiver displays the updated readings as graphs and trends minute-by-minute, and translates the readings from USB to Bluetooth
3 Control algorithm device (CAD)
Readings are sent to a control algorithm device (CAD) - eg a smartphone, tablet or PC - where an algorithm analyses them and calculates the correct insulin dose, if required
4 Insulin pump
The CAD communicates with a body-worn insulin pump that automatically administers the correct insulin dose via a cannula inserted under the skin
Delivering 180 IU
Insulin 180 IU

Safety criticality

- ✧ Primary safety-critical systems
  - Embedded software systems
  - Cause associated hardware failures, directly threatening people.
  - E.g., the insulin pump control system.
- ✧ Secondary safety-critical systems
  - Result in faults in other connected systems, affecting safety consequences.
  - E.g., the Mentcare system producing inappropriate treatment being prescribed.
  - Infrastructure control systems.

Hazards

✧ Situations or events that can lead to an accident

- Incorrect computation by software in navigation system
- Failure to detect possible disease in medication prescribing system

✧ Perform accident prevention actions

- Hazards do not inevitably lead to accidents

Safety achievement

✧ Hazard avoidance

- Appling hazard avoidance design to software systems.
- Prevent some classes of hazard.

✧ Hazard detection and removal

- Detecting and removing hazard before causing accidents.

✧ Damage limitation

- Protection features to minimise the damage.

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.

Normal accidents

✧ Rarely have a single cause in complex systems

✧ Designed to be resilient to a single point of failure

✧ A fundamental principle of safe systems design

- A single point of failure does not cause an accident.

✧ A result of combinations of malfunctions.

✧ Hard to anticipate all combinations in software systems

- Difficult to achieve complete safety.
- Accidents are inevitable.

Software safety benefits

✧ Software control systems contributes to system safety

- A large number of conditions to be monitored and controlled.
- Reducing human efforts and time in hazardous environments.
- Detecting and repairing safety-critical operator errors.

Safety requirements

Functional and non-functional requirements

- ✧ Functional requirements
  - Statements of services the system should provide,
  - How the system should react to particular inputs and how the system should behave in particular situations.
- ✧ Non-functional requirements
  - Constraints on the services or functions of the system.
  - Apply to the whole system rather than individual features.

Functional requirements

- ✧ Describe functionality or system services.
  - Depending on the type of software systems and users.
- ✧ Functional user requirements
  - High-level statements of what the system should do.
- ✧ Functional system requirements
  - The system services in detail.

Safety specification

- ✧ Goal
  - Identifying protection requirements.
  - Preventing injury or death or environmental damage.
- ✧ Safety requirements
  - Shall Not requirements.
  - Define situations and events that should never occur.
- ✧ Functional safety requirements
  - Checking and recovery features in a system.
  - Protection feature against failures and external attacks.

Hazard-driven analysis

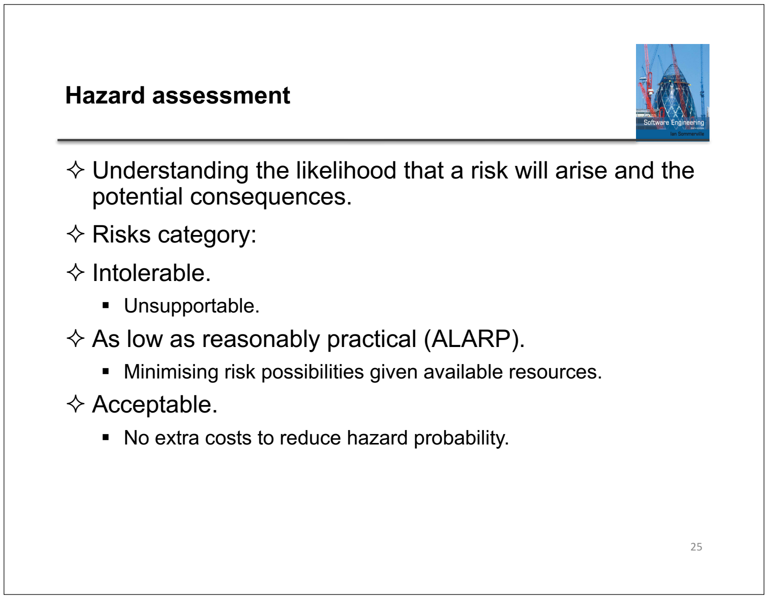
- ✧ Hazard identification
- ✧ Hazard assessment
- ✧ Hazard analysis
- ✧ Risk reduction
  - Safety requirements specification

Hazard identification

- ✧ Identify the hazards threatening the system.
- ✧ Different types of hazard:
  - Physical hazards
  - Electrical hazards
  - Biological hazards
  - Service failure hazards
  - Etc.

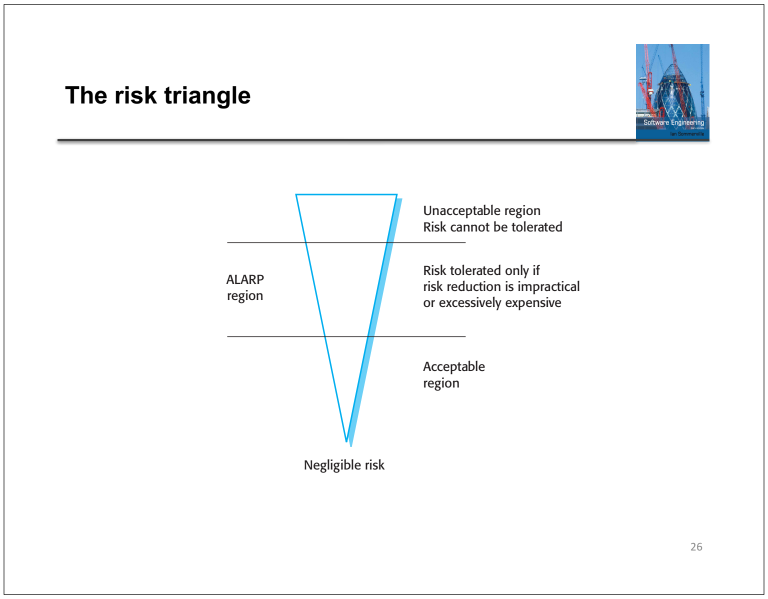
Insulin pump risks

- ✧ Insulin overdose (service failure).
- ✧ Insulin underdose (service failure).
- ✧ Power failure due to exhausted battery (electrical).
- ✧ Electrical interference with other medical equipment (electrical).
- ✧ Poor sensor and actuator contact (physical).
- ✧ Infection caused by introduction of machine (biological).
- ✧ Allergic reaction to materials or insulin (biological).



Hazard assessment

- ✧ Understanding the likelihood that a risk will arise and the potential consequences.
- ✧ Risks category:
- ✧ Intolerable.
  - Unsupportable.
- ✧ As low as reasonably practical (ALARP).
  - Minimising risk possibilities given available resources.
- ✧ Acceptable.
  - No extra costs to reduce hazard probability.



The risk triangle

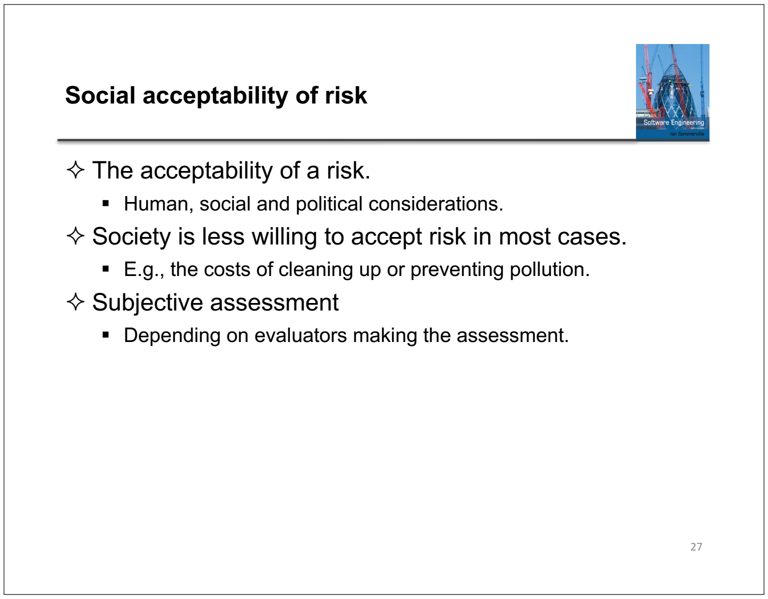
Unacceptable region
Risk cannot be tolerated

ALARP
region

Risk tolerated only if
risk reduction is impractical
or excessively expensive

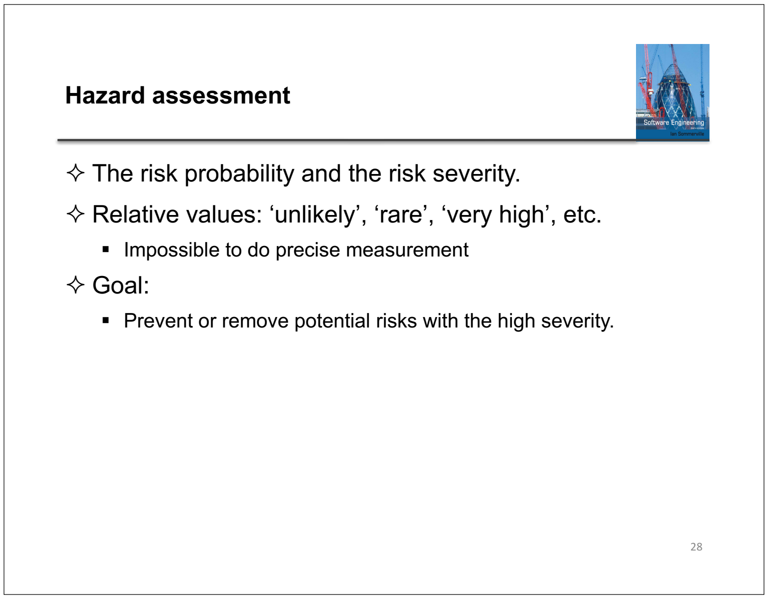
Acceptable
region

Negligible risk



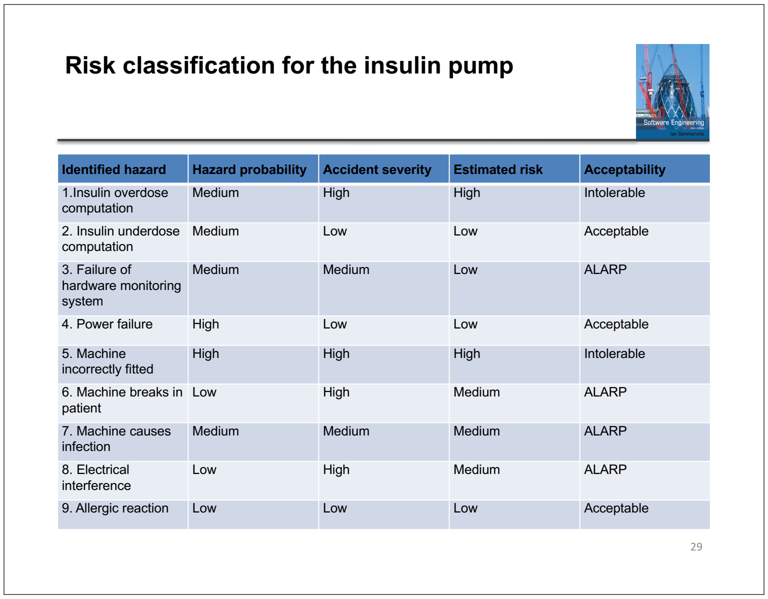
Social acceptability of risk

- ✧ The acceptability of a risk.
  - Human, social and political considerations.
- ✧ Society is less willing to accept risk in most cases.
  - E.g., the costs of cleaning up or preventing pollution.
- ✧ Subjective assessment
  - Depending on evaluators making the assessment.



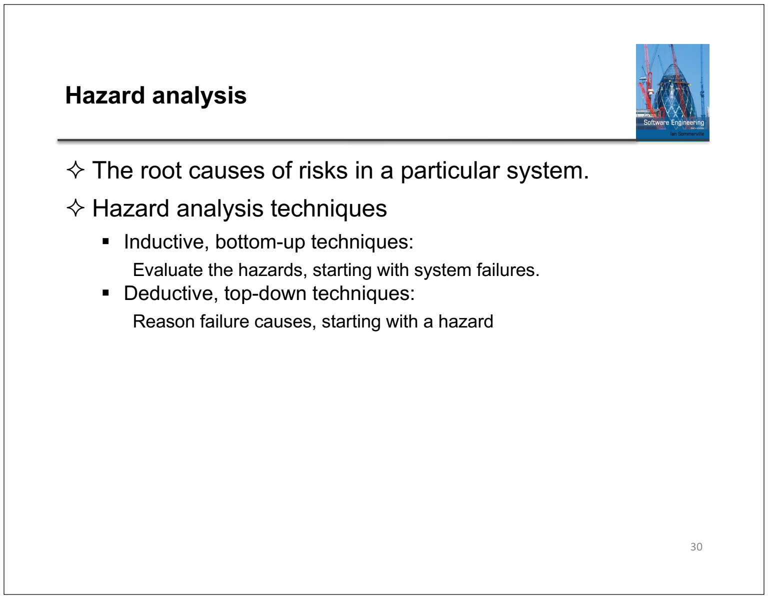
Hazard assessment

- ✧ The risk probability and the risk severity.
- ✧ Relative values: ‘unlikely’, ‘rare’, ‘very high’, etc.
  - Impossible to do precise measurement
- ✧ Goal:
  - Prevent or remove potential risks with the high severity.



Risk classification for the insulin pump

Identified hazard	Hazard probability	Accident severity	Estimated risk	Acceptability
1.Insulin overdose computation	Medium	High	High	Intolerable
2. Insulin underdose computation	Medium	Low	Low	Acceptable
3. Failure of hardware monitoring system	Medium	Medium	Low	ALARP
4. Power failure	High	Low	Low	Acceptable
5. Machine incorrectly fitted	High	High	High	Intolerable
6. Machine breaks in patient	Low	High	Medium	ALARP
7. Machine causes infection	Medium	Medium	Medium	ALARP
8. Electrical interference	Low	High	Medium	ALARP
9. Allergic reaction	Low	Low	Low	Acceptable



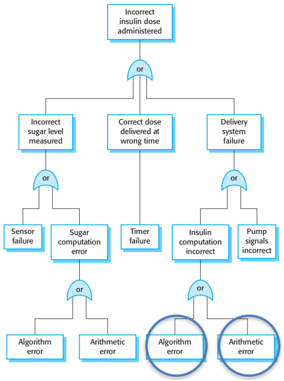
Hazard analysis

- ✧ The root causes of risks in a particular system.
- ✧ Hazard analysis techniques
  - Inductive, bottom-up techniques:
    Evaluate the hazards, starting with system failures.
  - Deductive, top-down techniques:
    Reason failure causes, starting with a hazard

Fault-tree analysis

- ✧ A deductive top-down technique.
- ✧ Hazard at the root of the tree
  - Identify states causing hazards.
- ✧ Linking conditions by relationships (e.g., ‘and’ or ‘or’)
- ✧ Goal
  - Minimizing the number of single failure causes.

An example of a software fault tree



Fault tree analysis

- ✧ Possible conditions of incorrect dose of insulin:
  - Incorrect measurement of blood sugar level
  - Failure of delivery system
  - Dose delivered at wrong time
- ✧ Root causes of these hazards:
  - Algorithm error
  - Arithmetic error

Risk reduction

- ✧ Goal:
  - Identify requirements for risk managements to avoid accidents.
- ✧ Risk reduction strategies
  - Hazard avoidance
  - Hazard detection and removal
  - Damage limitation

Strategy use

- ✧ Combining multiple risk reduction strategies
- ✧ E.g., a chemical plant control system:
  - Detecting and correcting excess pressure in the reactor.
  - Opening a relief valve as independent protection system

Insulin pump - software risks

- ✧ Arithmetic error
  - Data variable overflow or underflow during a computation.
  - Handing runtime exception.
- ✧ Algorithmic error
  - Comparison between previous and current values
  - Checking the maximum value to control dose.

Examples of safety requirements

SR1: The system shall not deliver a single dose of insulin that is greater than a specified maximum dose for a system user.

SR2: The system shall not deliver a daily cumulative dose of insulin that is greater than a specified maximum daily dose for a system user.

SR3: The system shall include a hardware diagnostic facility that shall be executed at least four times per hour.

SR4: The system shall include an exception handler for all of the exceptions that are identified in Table 3.

SR5: The audible alarm shall be sounded when any hardware or software anomaly is discovered and a diagnostic message, as defined in Table 4, shall be displayed.

SR6: In the event of an alarm, insulin delivery shall be suspended until the user has reset the system and cleared the alarm.

Safety engineering processes

Chapter 12 – Safety Engineering

- ~~Safety-critical systems~~
- ~~Safety requirements~~
- Safety engineering processes
- Safety cases

Safety engineering processes

Safety engineering processes

- Reliability engineering processes
  - Reviews and checks at each stage in the process
  - General goal of fault avoidance and fault detection
  - Safety reviews and explicit identification of hazards

Regulation

- Evidence that safety engineering processes used.
- For example:
  - The specification and records of the checks.
  - Evidence of the verification and validation the results.
  - Organizations for dependable software processes.

Agile methods and safety

- Agile methods are not usually used for safety-critical systems engineering
  - Extensive documentation.
  - A detailed safety analysis of a complete system specification.
- Test-driven development may be used

Safety assurance processes

- Defining and ensuring a dependable process.
- Process assurance focuses on:
  - The processes are appropriate for dependability required.
  - The processes are followed by the development team.
- Should generate documentation
  - Agile processes are not used for critical systems.

Processes for safety assurance

- ✧ Process assurance is important for safety-critical systems development:
  - Testing may not find all problems.
- ✧ Safety assurance activities
  - Record the analyses.
  - Personal responsibility.

Safety related process activities

- ✧ A hazard logging and monitoring system.
- ✧ Safety engineers who responsible for safety.
- ✧ Extensive use of safety reviews.
- ✧ A safety certification system.
- ✧ Detailed configuration management

Hazard analysis

- ✧ Identifying hazards and their root causes.
- ✧ Traceability from identified hazards
  - Analysis to to ensure that the hazards have been covered.
- ✧ A hazard log may be used to track hazards.

Safety reviews

- ✧ Driven by the hazard register.
- ✧ Assess the system and judge whether to cope with hazards in a safe way.

Formal verification

- ✧ A mathematical specification of the system is produced.
- ✧ Static verification technique used in development:
  - A formal specification -- mathematically analyzed for consistency.
    - Discover specification errors and omissions.
  - A program conforms to its mathematical specification
    - Programming and design errors.

Arguments for formal methods

- ✧ A mathematical specification requires a detailed analysis
- ✧ Concurrent systems
  - Discover race conditions.
  - Testing is difficult.
- ✧ Detect implementation errors before testing
  - Program is analyzed alongside the specification.

Arguments against formal methods

- ✧ Require specialized notations
  - Cannot be understood by domain experts.
- ✧ Expensive to develop a specification
- ✧ Proofs may contain errors.
- ✧ More cheaply using other V & V techniques.
- ✧ The proof making incorrect assumptions
  - System's behavior lies outside the scope of the proof.

Model checking

- ✧ Create a state model of using a specialized system
  - Checking the model for errors.
- ✧ The model checker explores all possible paths
  - Checks that a user-specified property is valid for each path.
- ✧ Verifying concurrent systems, which are hard to test.
- ✧ Model checking is computationally very expensive
  - Verification of small to medium sized critical systems.

Static program analysis

- ✧ Tools for source text processing.
- ✧ Parse the program text to discover erroneous conditions.
- ✧ Effective as an aid to inspections
  - A supplement to inspections.

Automated static analysis checks

Fault class	Static analysis check
Data faults	Variables used before initialization Variables declared but never used Variables assigned twice but never used between assignments Possible array bound violations Undeclared variables
Control faults	Unreachable code Unconditional branches into loops
Input/output faults	Variables output twice with no intervening assignment
Interface faults	Parameter-type mismatches Parameter number mismatches Non-usage of the results of functions Uncalled functions and procedures
Storage management faults	Unassigned pointers Pointer arithmetic Memory leaks

Levels of static analysis

- ✧ Characteristic error checking
  - Check for patterns in the code for characteristic of errors.
- ✧ User-defined error checking
  - Define error patterns, extending error types by specific rules
- ✧ Assertion checking
  - Formal assertions in their program
  - Symbolically executes to find potential problems.

Example for Symbolic Execution

```
x = readInput();
y = x * 2;

if (y == 12) {
  fail();
} else {
  print("OK");
}
```

Use of static analysis

- ✧ Particularly valuable when a language has weak typing
  - Many errors are undetected by the compiler.
- ✧ Security checking
  - Discover areas of vulnerability such as buffer overflows.
- ✧ The development of safety and security critical systems.

Safety cases

Sep 19

Safety and dependability cases

- ✧ Safety and dependability cases
  - Structured documents
  - Evidence of a required level of safety or dependability.
- ✧ Regulators check a system is as safe or dependable.
- ✧ Regulators and developers work together for a system safety/dependability case.

The system safety case

- ✧ A safety case is:
  - A documented body of evidence.
  - A system is adequately safe for a given environment.
- ✧ Formal proof, design rationale, safety proofs, etc.
- ✧ Wider system safety case that takes hardware and operational issues into account.

The contents of a software safety case

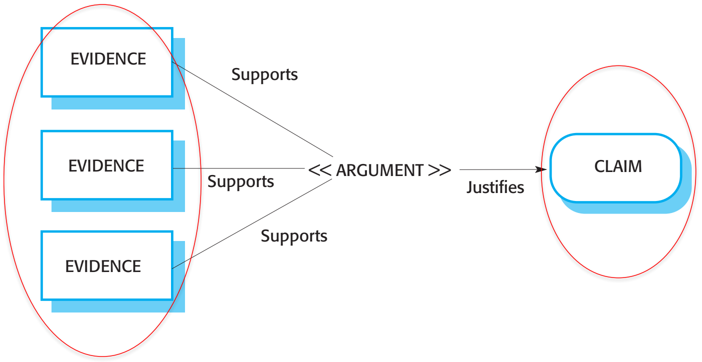
Chapter	Description
System description	An overview of the system and a description of its critical components.
Safety requirements	The safety requirements abstracted from the system requirements specification. Details of other relevant system requirements may also be included.
Hazard and risk analysis	Documents describing the hazards and risks that have been identified and the measures taken to reduce risk. Hazard analyses and hazard logs.
Design analysis	A set of structured arguments that justify why the design is safe.
Verification and validation	A description of the V & V procedures used and, where appropriate, the test plans for the system. Summaries of the test results showing defects that have been detected and corrected. If formal methods have been used, a formal system specification and any analyses of that specification. Records of static analyses of the source code.

Chapter	Description
Review reports	Records of all design and safety reviews.
Team competences	Evidence of the competence of all of the team involved in safety-related systems development and validation.
Process QA	Records of the quality assurance processes (see Chapter 24) carried out during system development.
Change management processes	Records of all changes proposed, actions taken and, where appropriate, justification of the safety of these changes. Information about configuration management procedures and configuration management logs.
Associated safety cases	References to other safety cases that may impact the safety case.

Structured arguments

- ✧ Safety cases be based on structured arguments
- ✧ Claims of safety and security justified by evidences.

Structured arguments



EVIDENCE

EVIDENCE

EVIDENCE

Supports

Supports

Supports

<< ARGUMENT >>

Justifies

CLAIM

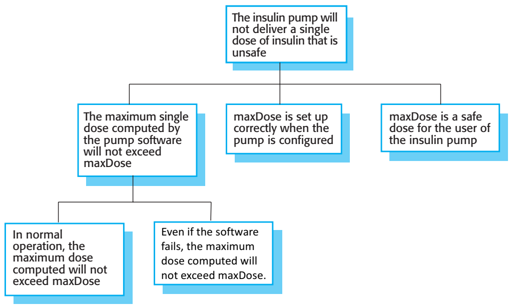
Insulin pump safety argument

- ✧ Arguments are based on claims and evidence.
- ✧ Insulin pump safety:
  - Claim: The maximum single dose of insulin to be delivered (CurrentDose) will not exceed MaxDose.
  - Evidence: Safety argument for insulin pump
  - Evidence: Test data for insulin pump. The value of currentDose was correctly computed in 400 tests
  - Evidence: Static analysis report for insulin pump software revealed no anomalies that affected the value of CurrentDose
  - Argument: The evidence presented demonstrates that the maximum dose of insulin that can be computed = MaxDose.

Structured safety arguments

- ✧ Structured arguments of a system safety obligations.
- ✧ Generally based on a claim hierarchy.

A safety claim hierarchy for the insulin pump



The insulin pump will not deliver a single dose of insulin that is unsafe

The maximum single dose computed by the pump software will not exceed maxDose

maxDose is set up correctly when the pump is configured

maxDose is a safe dose for the user of the insulin pump

In normal operation, the maximum dose computed will not exceed maxDose

Even if the software fails, the maximum dose computed will not exceed maxDose.

Software safety arguments

- ✧ Show that the system cannot reach in unsafe state.
- ✧ These are weaker than correctness arguments which must show that the system code conforms to its specification.
- ✧ They are generally based on proof by contradiction
  - Assume that an unsafe state can be reached;
  - Show that this is contradicted by the program code.

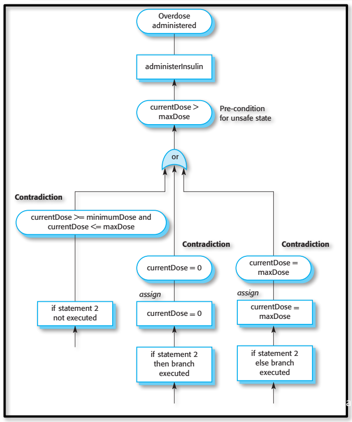
Insulin dose computation with safety checks

```
-- The insulin dose to be delivered is a function of blood sugar level,
-- the previous dose delivered and the time of delivery of the previous dose

currentDose = computeInsulin () ;

// Safety check—adjust currentDose if necessary.
// if statement 1
if (previousDose == 0)
{
        if (currentDose > maxDose/2)
        currentDose = maxDose/2 ;
}
else
        if (currentDose > (previousDose * 2) )
                currentDose = previousDose * 2 ;
// if statement 2
if ( currentDose < minimumDose )
        currentDose = 0 ;
else if ( currentDose > maxDose )
        currentDose = maxDose ;
administerInsulin (currentDose) ;
```

Informal safety argument based on demonstrating contradictions



```
...

/* if statement 2 */
if ( currentDose < minimumDose ) {
    currentDose = 0 ;
}
else if ( currentDose > maxDose ) {
    currentDose = maxDose ;
}

administerInsulin (currentDose) ;
...
```

Program paths

```
if ( currentDose < minimumDose ) {
    currentDose = 0 ;
}
else if ( currentDose > maxDose ) {
    currentDose = maxDose ;
}
administerInsulin (currentDose) ;
```

- ✧ Neither branch of if-statement 2 is executed
  - Can only happen if CurrentDose is >= minimumDose and <= maxDose.
- ✧ then branch of if-statement 2 is executed
  - currentDose = 0.
- ✧ else branch of if-statement 2 is executed
  - currentDose = maxDose.
- ✧ In all cases, the post conditions contradict the unsafe condition that the dose administered is greater than maxDose.

Key points

- ✧ Safety-critical systems are systems whose failure can lead to human injury or death.
- ✧ A hazard-driven approach is used to understand the safety requirements for safety-critical systems. You identify potential hazards and decompose these (using methods such as fault tree analysis) to discover their root causes. You then specify requirements to avoid or recover from these problems.
- ✧ It is important to have a well-defined, certified process for safety-critical systems development. This should include the identification and monitoring of potential hazards.

Key points

- ✧ Static analysis is an approach to V & V that examines the source code of a system, looking for errors and anomalies. It allows all parts of a program to be checked, not just those parts that are exercised by system tests.
- ✧ Model checking is a formal approach to static analysis that exhaustively checks all states in a system for potential errors.
- ✧ Safety and dependability cases collect the evidence that demonstrates a system is safe and dependable. Safety cases are required when an external regulator must certify the system before it is used.