Syllabus:
Software testing, Development testing, Test-driven development, Release testing, User testing.
Dependability properties, Availability and reliability, Safety Security.

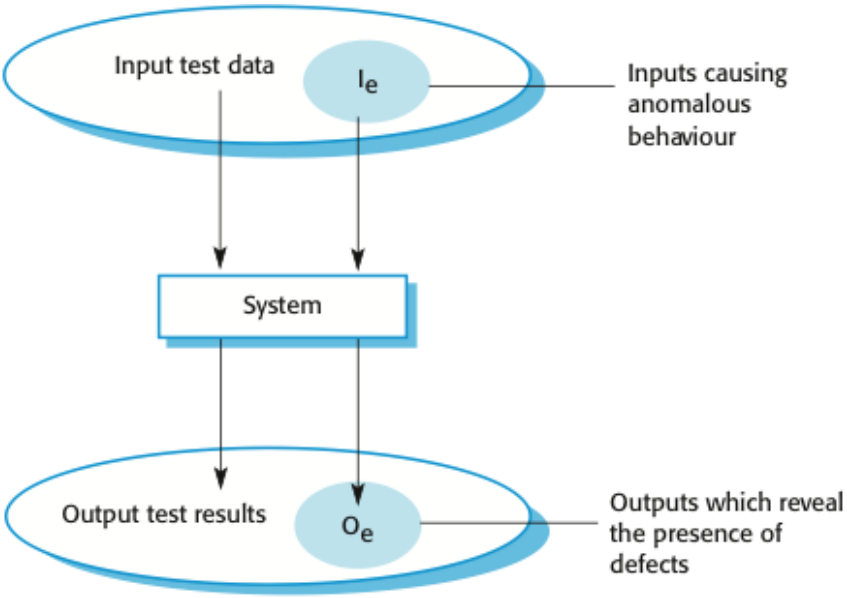
(Reference: Sommerville, Software Engineering, 10 ed., Chapter 7, 10, 11, 12)

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies or information about the program's non-functional attributes. Testing can reveal the presence of errors, but NOT their absence. Testing is part of a more general verification and validation process, which also includes static validation techniques.

Goals of software testing:

- To demonstrate to the developer and the customer that the software meets its requirements.
  - Leads to validation testing: you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  - A successful test shows that the system operates as intended.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Leads to defect testing: the test cases are designed to expose defects; the test cases can be deliberately obscure and need not reflect how the system is normally used.
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Testing can be viewed as an input-output process:



Verification and validation

Testing is part of a broader process of software verification and validation (V & V).

- Verification: Are we building the product right?
  The software should conform to its specification.
- Validation: Are we building the right product?
  The software should do what the user really requires.

The goal of V & V is to establish confidence that the system is good enough for its intended use, which depends on:

- Software purpose: the level of confidence depends on how critical the software is to an organization.
- User expectations: users may have low expectations of certain kinds of software.
- Marketing environment: getting a product to market early may be more important than finding defects in the program.

Inspections and testing

Software inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections not require execution of a system so may be used before implementation. They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.). They have been shown to be an effective technique for discovering program errors.

Advantages of inspections include:

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the V & V process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc.

Typically, a commercial software system has to go through **three stages of testing**:

- Development testing: the system is tested during development to discover bugs and defects.
- Release testing: a separate testing team test a complete version of the system before it is released to users.
- User testing: users or potential users of a system test the system in their own environment.

Development testing

Development testing includes all testing activities that are carried out by the team developing the system:

- Unit testing: individual program units or object classes are tested; should focus on testing the functionality of objects or methods.
- Component testing: several individual units are integrated to create composite components; should focus on testing component interfaces.
- System testing: some or all of the components in a system are integrated and the system is tested as a whole; should focus on testing component interactions.

Unit testing

Unit testing is the process of testing individual components in isolation. It is a defect testing process. Units may be:

- Individual functions or methods within an object;
- Object classes with several attributes and methods;
- Composite components with defined interfaces used to access their functionality.

When testing object classes, tests should be designed to provide coverage of all of the features of the object:

- Test all operations associated with the object;
- Set and check the value of all attributes associated with the object;
- Put the object into all possible states, i.e. simulate all events that cause a state change.

Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

An automated test has three parts:

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- A call part, where you call the object or method to be tested.
- An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do. If there are defects in the component, these should be revealed by test cases.

This leads to two types of unit test cases:

- The first of these should reflect normal operation of a program and should show that the component works as expected.
- The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Component testing

Software components are often composite components that are made up of several interacting objects. You access the functionality of these objects through the defined component interface. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.

Interface types include:

- Parameter interfaces: data passed from one method or procedure to another.
- Shared memory interfaces: block of memory is shared between procedures or functions.
- Procedural interfaces: sub-system encapsulates a set of procedures to be called by other sub-systems.
- Message passing interfaces: sub-systems request services from other sub-systems.

Interface errors:

- Interface misuse: a calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding: a calling component embeds assumptions about the behavior of the called component which are incorrect.
- Timing errors: the called and the calling component operate at different speeds and out-of-date information is accessed.

General guidelines for interface testing:

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

System testing

System testing during development involves integrating components to create a version of the system and then testing the integrated system. The focus in system testing is testing the interactions between components. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. System testing tests the emergent behavior of a system.

During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested. Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.

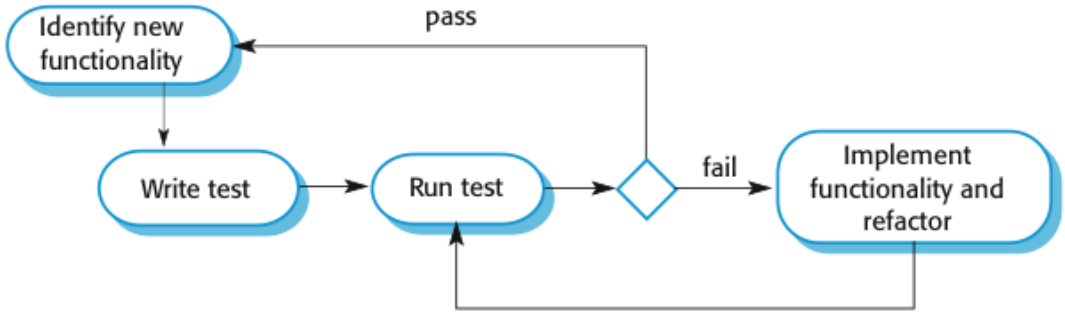
The use cases developed to identify system interactions can be used as a basis for system testing. Each use case usually involves several system components so testing the use case forces these interactions to occur. The sequence diagrams associated with the use case document the components and their interactions that are being tested.

Test-driven development

Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development. Tests are written before code and 'passing' the tests is the critical driver of development. This is a differentiating feature of TDD versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code. The code is developed incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test. TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

TDD example - a string calculator.

The goal of TDD isn't to ensure we write tests by writing them first, but to produce working software that achieves a targeted set of requirements using simple, maintainable solutions. To achieve this goal, TDD provides strategies for keeping code working, simple, relevant, and free of duplication.



TDD process includes the following activities:

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development:

- Code coverage: every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing: a regression test suite is developed incrementally as a program is developed.
- Simplified debugging: when a test fails, it should be obvious where the problem lies; the newly written code needs to be checked and modified.
- System documentation: the tests themselves are a form of documentation that describe what the code should be doing.

Regression testing is testing the system to check that changes have not 'broken' previously working code. In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program. Tests must run 'successfully' before the change is committed.

Release testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. The primary goal of the release testing process is to convince the customer of the system that it is good enough for use. Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use. Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release testing is a form of system testing. Important differences:

- A separate team that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Requirements-based testing involves examining each requirement and developing a test or tests for it. It is validation rather than defect testing: you are trying to demonstrate that the system has properly implemented its requirements.

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as part of the requirements engineering process, then you may be able to reuse these as testing scenarios.

Part of release testing may involve testing the emergent properties of a system, such as performance and reliability. Tests should reflect the profile of use of the system. Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

User testing

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. User testing is essential, even when comprehensive system and release testing have been carried out.

Types of user testing include:

- Alpha testing: users of the software work with the development team to test the software at the developer's site.
- Beta testing: a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

- Acceptance testing: customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system. Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made. Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

Dependability & Dependable Systems

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.

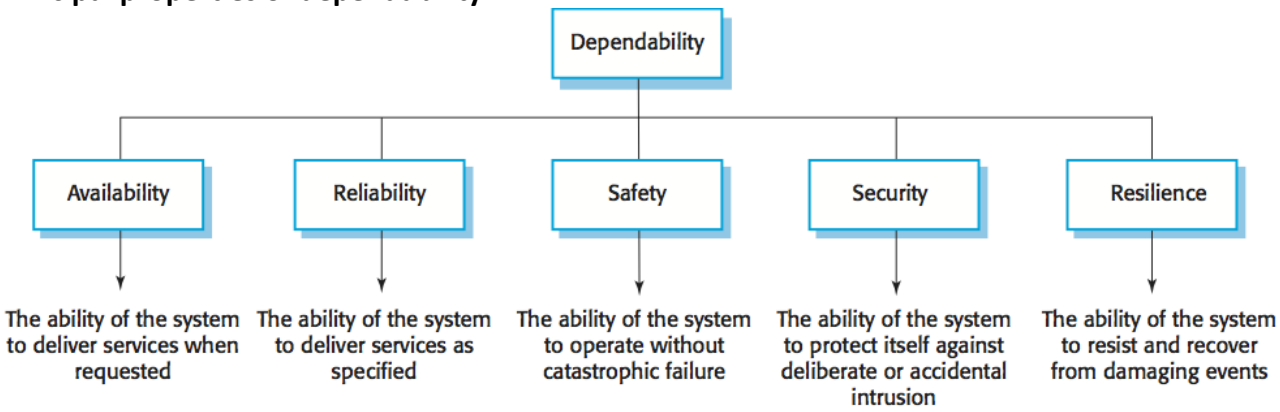
System failures may have widespread effects with large numbers of people affected by the failure. Systems that are not dependable and are unreliable, unsafe or insecure may be rejected by their users. The costs of system failure may be very high if the failure leads to economic losses or physical damage.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

Causes of failure:

- Hardware failure
  Hardware fails because of design and manufacturing errors or because components have reached the end of their natural life.
- Software failure
  Software fails due to errors in its specification, design or implementation.
- Operational failure
  Human operators make mistakes. Now perhaps the largest single cause of system failures in socio-technical systems.

Dependability properties

Principal properties of dependability:



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.

Other properties of software dependability:

- 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;
- Survivability reflects the extent to which the system can deliver services whilst under hostile attack;
- Error tolerance reflects the extent to which user input errors can be avoided and tolerated.

Many dependability attributes depend on one another. Safe system operation depends on the system being available and operating reliably. A system may be unreliable because its data has been corrupted by an external attack. Denial of service attacks on a system are intended to make it unavailable. If a system is infected with a virus, you cannot be confident in its reliability or safety.

How to achieve dependability?

- 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 recognize and resist cyberattacks.
- Include recovery mechanisms to help restore normal system service after a failure.

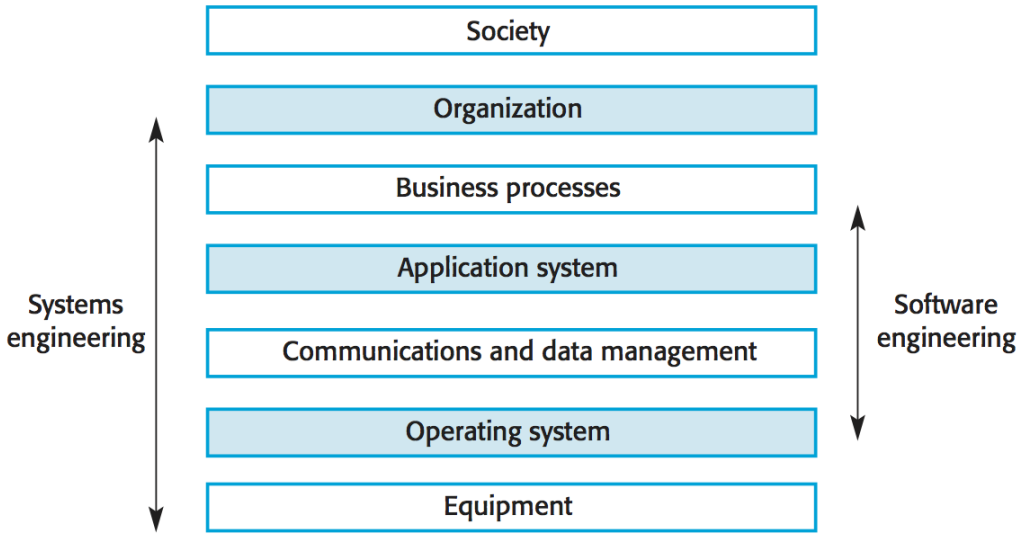
Dependability costs tend to increase exponentially as increasing levels of dependability are required because of two reasons. The use of more expensive development techniques and hardware that are required to achieve the higher levels of dependability. The increased testing and system validation that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

Socio-technical systems

Software engineering is not an isolated activity but is part of a broader systems engineering process. Software systems are therefore not isolated systems but are essential components of broader systems that have a human, social or organizational purpose.

- Equipment: hardware devices, some of which may be computers; most devices will include an embedded system of some kind.
- Operating system: provides a set of common facilities for higher levels in the system.
- Communications and data management: middleware that provides access to remote systems and databases.
- Application systems: specific functionality to meet some organization requirements.
- Business processes: a set of processes involving people and computer systems that support the activities of the business.
- Organizations: higher level strategic business activities that affect the operation of the system.
- Society: laws, regulation and culture that affect the operation of the system.

There are interactions and dependencies between the layers in a system and changes at one level ripple through the other levels. For dependability, a systems perspective is essential.



Emergent properties

Emergent properties are properties of the system as a whole rather than properties that can be derived from the properties of components of a system. Emergent properties are a consequence of the relationships between system components. They can therefore only be assessed and measured once the components have been integrated into a system.

Two types of emergent properties:

- Functional properties
  These appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.
- Non-functional emergent properties
  Examples are reliability, performance, safety, and security. These relate to the behavior of the system in its operational environment. They are often critical for computer-based systems as failure to achieve some minimal defined level in these properties may make the system unusable.

Some examples of emergent properties:

Property	**Description**
Volume	The volume of a system (the total space occupied) varies depending on how the component assemblies are arranged and connected.
Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failures and therefore affect the reliability of the system.
Security	The security of the system (its ability to resist attack) is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguards.
Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem, access the components that are faulty, and modify or replace these components.
Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators, and its operating environment.

Regulation and compliance

Many critical systems are regulated systems, which means that their use must be approved by an external regulator before the systems go into service. Examples: 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.

Regulation and compliance (following the rules) applies to the socio-technical system as a whole and not simply the software element of that system. 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 develop the documentation for certification as it is to develop the system itself.

Redundancy and diversity

- Redundancy: Keep more than a single version of critical components so that if one fails then a backup is available.
- Diversity: Provide the same functionality in different ways in different components so that they will not fail in the same way.
- Redundant and diverse components should be independent so that they will not suffer from 'common-mode' failures.

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.

Dependable processes

To ensure a minimal number of software faults, it is important to have a well-defined, repeatable software process. A well-defined repeatable process is one that does not depend entirely on individual skills; rather can be enacted by different people. Regulators use information about the process to check if good software engineering practice has been used. For fault detection, it is clear that the process activities should include significant effort devoted to verification and validation.

Dependable process characteristics:

- Explicitly defined
  A process that has a defined process model that is used to drive the software production process. Data must be collected during the process that proves that the development team has followed the process as defined in the process model.
- Repeatable
  A process that does not rely on individual interpretation and judgment. The process can be repeated across projects and with different team members, irrespective of who is involved in the development.

Dependable process activities

- Requirements reviews to check that the requirements are, as far as possible, complete and consistent.
- Requirements management to ensure that changes to the requirements are controlled and that the impact of proposed requirements changes is understood.
- Formal specification, where a mathematical model of the software is created and analysed.
- System modelling, where the software design is explicitly documented as a set of graphical models, and the links between the requirements and these models are documented.
- Design and program inspections, where the different descriptions of the system are inspected and checked by different people.

- Static analysis, where automated checks are carried out on the source code of the program.
- Test planning and management, where a comprehensive set of system tests is designed.

Dependable software often requires certification so both process and product documentation has to be produced. Up-front requirements analysis is also essential to discover requirements and requirements conflicts that may compromise the safety and security of the system. These conflict with the general approach in agile development of co-development of the requirements and the system and minimizing documentation.

An agile process may be defined that incorporates techniques such as iterative development, test-first development and user involvement in the development team. So long as the team follows that process and documents their actions, agile methods can be used. However, additional documentation and planning is essential so 'pure agile' is impractical for dependable systems engineering.

Reliability Engineering

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.

Reliability terminology

Term	Description
Human error or mistake	Human behavior that results in the introduction of faults into a system.
System fault	A characteristic of a software system that can lead to a system error.
System error	An erroneous system state that can lead to system behavior that is unexpected by system users.
System failure	An event that occurs at some point in time when the system does not deliver a service as expected by its users.

Failures are a usually a result of system errors that are derived from faults in the system. However, faults do not necessarily result in system errors if the erroneous system state is transient and can be 'corrected' before an error arises. Errors do not necessarily lead to system failures if the error is corrected by built-in error detection and recovery mechanism.

Fault management strategies to achieve reliability:

- Fault avoidance
  Development techniques are used that either minimize the possibility of mistakes or trap mistakes before they result in the introduction of system faults.
- Fault detection and removal
  Verification and validation techniques that increase the probability of detecting and correcting errors before the system goes into service are used.
- Fault tolerance
  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.

Availability and reliability

Reliability is the probability of failure-free system operation over a specified time in a given environment for a given purpose. Availability is 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.

The formal definition of reliability does not always reflect the user's perception of a system's reliability. Reliability can only be defined formally with respect to a system specification i.e. a failure is a deviation from a specification. Users don't read specifications and don't know how the system is supposed to behave; therefore, perceived reliability is more important in practice.

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.

Removing X% of the faults in a system will not necessarily improve the reliability by X%. Program defects may be in rarely executed sections of the code so may never be encountered by users. Removing these does not affect the perceived reliability. Users adapt their behavior to avoid system features that may fail for them. A program with known faults may therefore still be perceived as reliable by its users.

Reliability requirements

Functional reliability requirements define system and software functions that avoid, detect or tolerate faults in the software and so ensure that these faults do not lead to system failure.

Reliability is a measurable system attribute so non-functional reliability requirements may be specified quantitatively. These define the number of failures that are acceptable during normal use of the system or the time in which the system must be available. Functional reliability requirements define system and software functions that avoid, detect or tolerate faults in the software and so ensure that these faults do not lead to system failure. Software reliability requirements may also be included to cope with hardware failure or operator error.

Reliability metrics are units of measurement of system reliability. System reliability is measured by counting the number of operational failures and, where appropriate, relating these to the demands made on the system and the time that the system has been operational. Metrics include:

- Probability of failure on demand (POFOD). The probability that the system will fail when a service request is made. Useful when demands for service are intermittent and relatively infrequent.

- Rate of occurrence of failures (ROCOF). Reflects the rate of occurrence of failure in the system. Relevant for systems where the system has to process a large number of similar requests in a short time. Mean time to failure (MTTF) is the reciprocal of ROCOF.
- Availability (AVAIL). Measure of the fraction of the time that the system is available for use. Takes repair and restart time into account. Relevant for non-stop, continuously running systems.

Non-functional reliability requirements are specifications of the required reliability and availability of a system using one of the reliability metrics (POFOD, ROCOF or AVAIL). Quantitative reliability and availability specification has been used for many years in safety-critical systems but is uncommon for business critical systems. However, as more and more companies demand 24/7 service from their systems, it makes sense for them to be precise about their reliability and availability expectations.

Functional reliability requirements specify the faults to be detected and the actions to be taken to ensure that these faults do not lead to system failures.

- Checking requirements that identify checks to ensure that incorrect data is detected before it leads to a failure.
- Recovery requirements that are geared to help the system recover after a failure has occurred.
- Redundancy requirements that specify redundant features of the system to be included.
- Process requirements for reliability which specify the development process to be used may also be included.

Fault tolerance

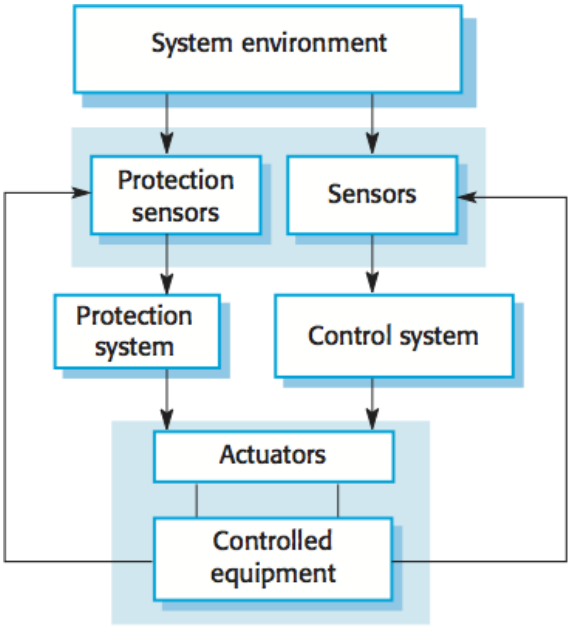
In critical situations, software systems must be fault tolerant. Fault tolerance is required where there are high availability requirements or where system failure costs are very high. Fault tolerance means that the system can continue in operation in spite of software failure. Even if the system has 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.

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.

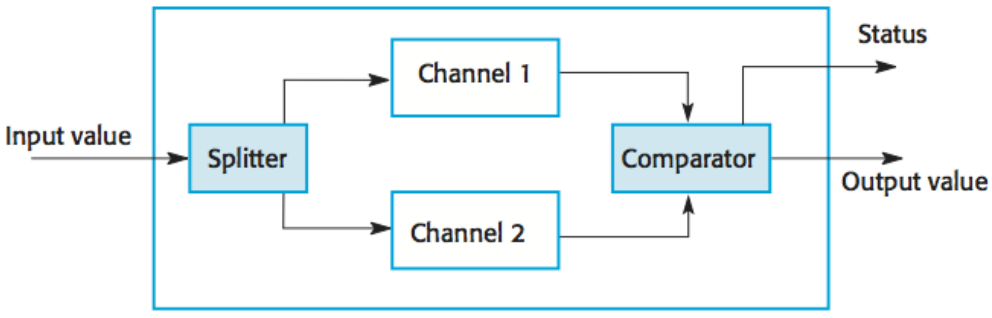
Protection system is a specialized system that is associated with some other control system, which can take emergency action if a failure occurs, e.g. a system to stop a train if it passes a red light, or a system to shut down a reactor if temperature/pressure are too high. Protection systems independently monitor the controlled system and the environment. If a problem is detected, it issues commands to take emergency action to shut down the system and avoid a catastrophe. Protection systems are redundant because they include monitoring and control

capabilities that replicate those in the control software. Protection systems should be diverse and use different technology from the control software. They are simpler than the control system so more effort can be expended in validation and dependability assurance. Aim is to ensure that there is a low probability of failure on demand for the protection system.



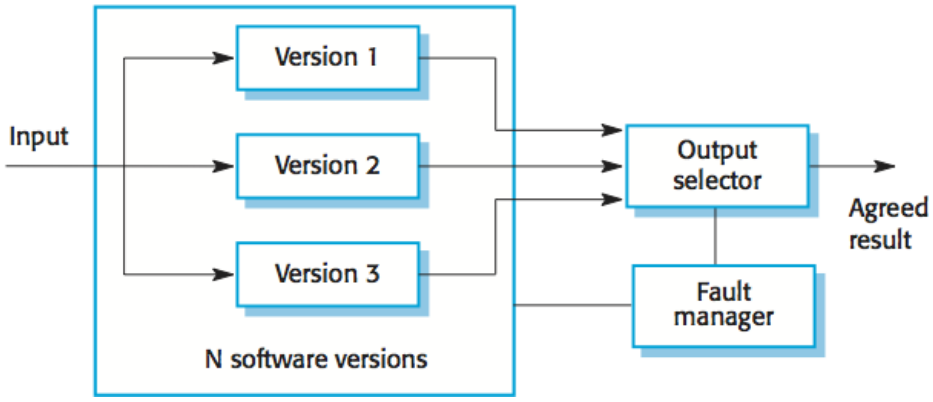
Self-monitoring architecture is a multi-channel architectures where the system monitors its own operations and takes action if inconsistencies are detected. The same computation is carried out on each channel and the results are compared. If the results are identical and are produced at the same time, 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.

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.



N-version programming involves multiple versions of a software system to carry out computations at the same time. There should be an odd number of computers involved, typically 3.

The results are compared using a voting system and the majority result is taken to be the correct result. Approach derived from the notion of triple-modular redundancy, as used in hardware systems.



Hardware fault tolerance depends on triple-modular redundancy (TMR). There are three replicated identical components that receive the same input and whose outputs are compared. If one output is different, it is ignored and component failure is assumed. Based on most faults resulting from component failures rather than design faults and a low probability of simultaneous component failure.

Programming for reliability

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

- 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 ().
- 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.
- 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.
- 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.

Error-prone constructs:

- Unconditional branch (goto) statements
- Floating-point numbers (inherently imprecise, which may lead to invalid comparisons)
- Pointers
- Dynamic memory allocation
- Parallelism (can result in subtle timing errors because of unforeseen interaction between parallel processes)
- Recursion (can cause memory overflow as the program stack fills up)
- Interrupts (can cause a critical operation to be terminated and make a program difficult to understand)
- Inheritance (code is not localized, which may result in unexpected behavior when changes are made and problems of understanding the code)
- Aliasing (using more than 1 name to refer to the same state variable)
- Unbounded arrays (may result in buffer overflow)
- Default input processing (if the default action is to transfer control elsewhere in the program, incorrect or deliberately malicious input can then trigger a program failure)

- 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.
- 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.
- 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.
- 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.

Safety Engineering

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.

Safety and reliability are related but distinct. Reliability is concerned with conformance to a given specification and delivery of service. Safety is concerned with ensuring system cannot cause damage irrespective of whether or not it conforms to its specification. System reliability is essential for safety but is not enough.

Safety terminology

Term	Definition
Accident (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.
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.
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.
Hazard probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from 'probable' (e.g. 1/100 chance of a hazard occurring) to 'implausible' (no conceivable situations are likely in which the hazard could occur).
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.

Reliable systems can be unsafe:

- Dormant faults in a system can remain undetected for many years and only rarely arise.
- Specification errors: the system can behave as specified but still cause an accident.
- Hardware failures could generate spurious inputs that are hard to anticipate in the specification.
- Context-sensitive commands (the right command at the wrong time) are often the result of operator error.

Safety-critical systems

In safety-critical systems 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.

Two levels of safety criticality:

- Primary safety-critical systems: embedded software systems whose failure can cause the associated hardware to fail and directly threaten people.

- Secondary safety-critical systems: systems whose failure results in faults in other (socio-technical) systems, which can then have safety consequences.

Safety achievement strategies:

- 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 minimize the damage that may result from an accident.

Accidents in complex systems rarely have a single cause as these systems are designed to be resilient to a single point of failure. 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. However, accidents are inevitable.

Safety requirements

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, and features that provide protection against system failures and external attacks.

Hazard-driven analysis:

- Hazard identification
  Identify the hazards that may threaten the system. Hazard identification may be based on different types of hazard: physical, electrical, biological, service failure, etc.
- Hazard assessment
  - The process is concerned with understanding the likelihood that a risk will arise and the potential consequences if an accident or incident should occur. Risks may be categorized as: intolerable (must never arise or result in an accident), as low as reasonably practical - ALARP (must minimize the possibility of risk given cost and schedule constraints), and acceptable (the consequences of the risk are acceptable and no extra costs should be incurred to reduce hazard probability).
  - The acceptability of a risk is determined by human, social, and political considerations. In most societies, the boundaries between the regions are pushed upwards with time i.e. society is less willing to accept risk (e.g., the costs of cleaning up pollution may be less than the costs of preventing it but this may not be socially acceptable). Risk assessment is subjective.
  - Hazard assessment process: for each identified hazard, assess hazard probability, accident severity, estimated risk, acceptability.
- Hazard analysis
  Concerned with discovering the root causes of risks in a particular system. Techniques have been mostly derived from safety-critical systems and can be: inductive, bottom-up:

start with a proposed system failure and assess the hazards that could arise from that failure; and deductive, top-down: start with a hazard and deduce what the causes of this could be.
    - Fault-tree analysis is a deductive top-down technique.:
        - Put the risk or hazard at the root of the tree and identify the system states that could lead to that hazard.
        - Where appropriate, link these with 'and' or 'or' conditions.
        - A goal should be to minimize the number of single causes of system failure.
- Risk reduction
  The aim of this process is to identify dependability requirements that specify how the risks should be managed and ensure that accidents/incidents do not arise. Risk reduction strategies: hazard avoidance; hazard detection and removal; damage limitation.

Safety engineering processes

Safety engineering processes are based on reliability engineering processes. Regulators may require evidence that safety engineering processes have been used in system development.

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

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 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.

Safety-related process activities:

- Creation of a hazard logging and monitoring system;
- Appointment of project safety engineers who have explicit responsibility for system safety;
- Extensive use of safety reviews;
- Creation of a safety certification system where the safety of critical components is formally certified;
- Detailed configuration management.

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.

Advantages of formal methods

Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors. Concurrent systems can be analyzed 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.

Disadvantages of formal methods

Require specialized notations that cannot 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.

Model checking involves creating an extended finite state model of a system and, using a specialized system (a model checker), checking that model for errors. The model checker explores all possible paths through the model and checks that a user-specified property is valid for each path. Model checking is particularly valuable for verifying concurrent systems, which are hard to test. Although model checking is computationally very expensive, it is now practical to use it in the verification of small to medium sized critical systems.

Static program analysis uses software tools for source text processing. They parse the program text and try to discover potentially erroneous conditions and bring these to the attention of the V & V team. They are very effective as an aid to inspections - they are a supplement to but not a replacement for inspections.

Three levels of static analysis:

- Characteristic error checking
  The static analyzer can check for patterns in the code that are characteristic of errors made by programmers using a particular language.
- User-defined error checking
  Users of a programming language define error patterns, thus extending the types of error that can be detected. This allows specific rules that apply to a program to be checked.
- Assertion checking
  Developers include formal assertions in their program and relationships that must hold. The static analyzer symbolically executes the code and highlights potential problems.

Static analysis is particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler. Particularly valuable for security checking - the static analyzer can discover areas of vulnerability such as buffer overflows or unchecked inputs. Static analysis is now routinely used in the development of many safety and security critical systems.