MODULE 3

Verification and Validation

Objectives:

- To introduce software verification and validation and to discuss the distinction between them.
- To describe the program inspection process and its role in V & V.
- To explain static analysis as a verification technique.
- To describe the Cleanroom software development process.

Verification vs validation:

- 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 V & V process:

- Is a whole life-cycle process - V & V must be applied at each stage in the software process.
- Within the V & V process, there are two complementary approaches to system checking and analysis:
    - ➢ Software inspections or peer reviews: The discovery of defects in a system;
    - ➢ The assessment of whether or not the system is useful and useable in an Operational situation

V& V goals:

- Verification and validation should establish confidence that the software is fit for purpose.
- This does NOT mean completely free of defects.
- Rather, it must be good enough for its intended use and the type of use will determine the degree of confidence that is needed.

V & V confidence:

- Depends on system's purpose, user expectations and marketing environment
    - ➢ Software function: The level of confidence depends on how critical the software is to an organisation.
    - ➢ **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.
- Within the V & V process, there are two complementary approaches to system checking and analysis:
- **Software inspections:** Concerned with analysis of the static system representation to discover problems (static verification)
    - ➢ May be supplement by tool-based document and code analysis
- **Software testing:** Concerned with exercising and observing product behaviour (dynamic verification)
    - ➢ The system is executed with test data and its operational behaviour is observed

Static and dynamic V&V:

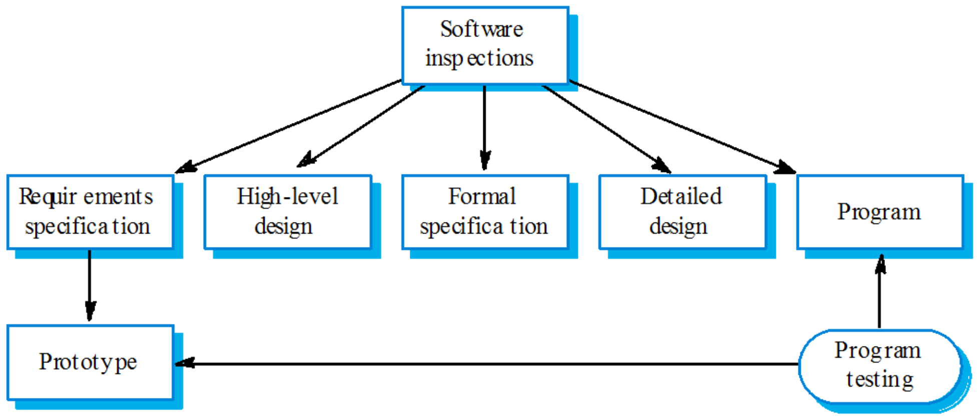


Figure: Static and dynamic verification and validation

- There are two distinct types of testing that may be used at different stages in the software process:
- **Defect testing**
    - ➢ Tests designed to discover system defects.
    - ➢ A successful defect test is one which reveals the presence of defects in a system.

- **Validation testing**
  - ➢ Intended to show that the software meets its requirements.
  - ➢ A successful test is one that shows that a requirements has been properly implemented.
- The processes of V & V and debugging are normally interleaved. As you discover faults in the program that you are testing, you have to change the program to correct these faults.
- **Testing and debugging goals are:**
  - ➢ Verification and validation processes are intended to establish the existence of defects in a software system.
  - ➢ Debugging is a process that locates and corrects these defects, shown in Figure

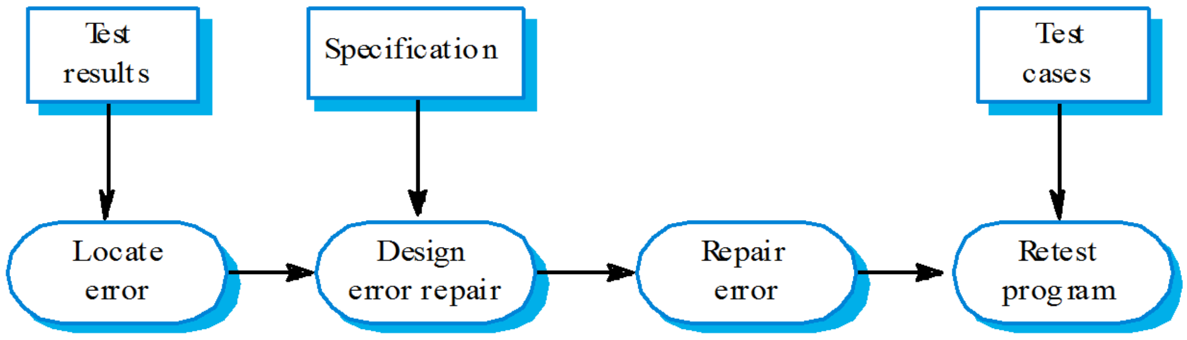


Figure The debugging process

Planning verification and validation

- Careful planning is required to get the most out of testing and inspection processes.
- Planning should start early in the development process.
- The plan should identify the balance between static verification and testing.
- Test planning is about defining standards for the testing process rather than describing product tests.
- The software development process model shown in Figure 22.3 is sometimes called the V-model

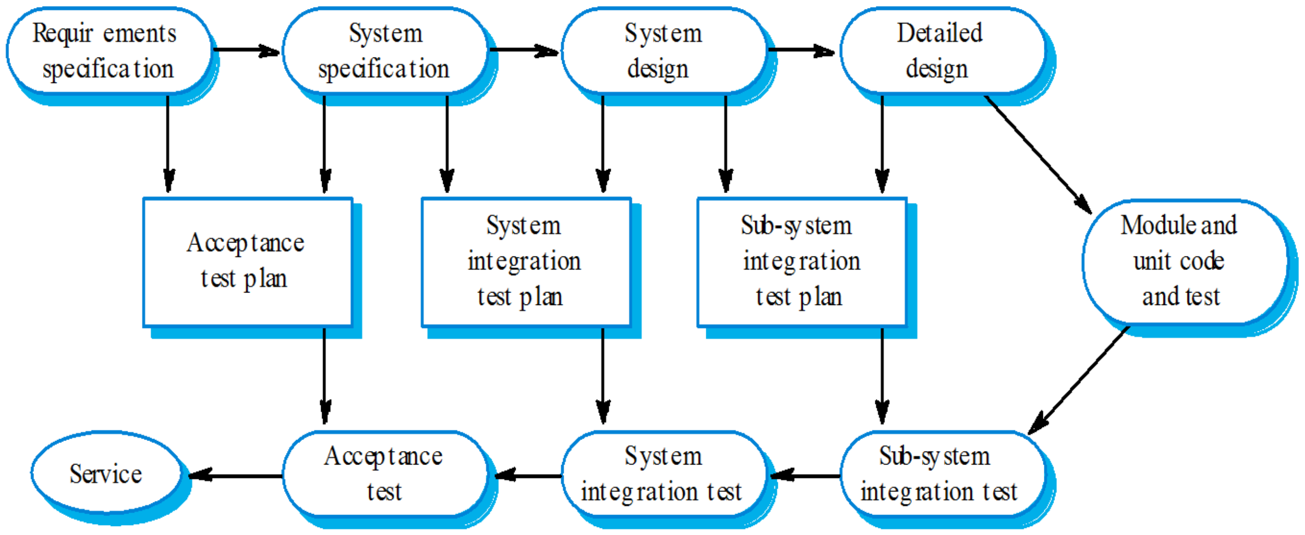
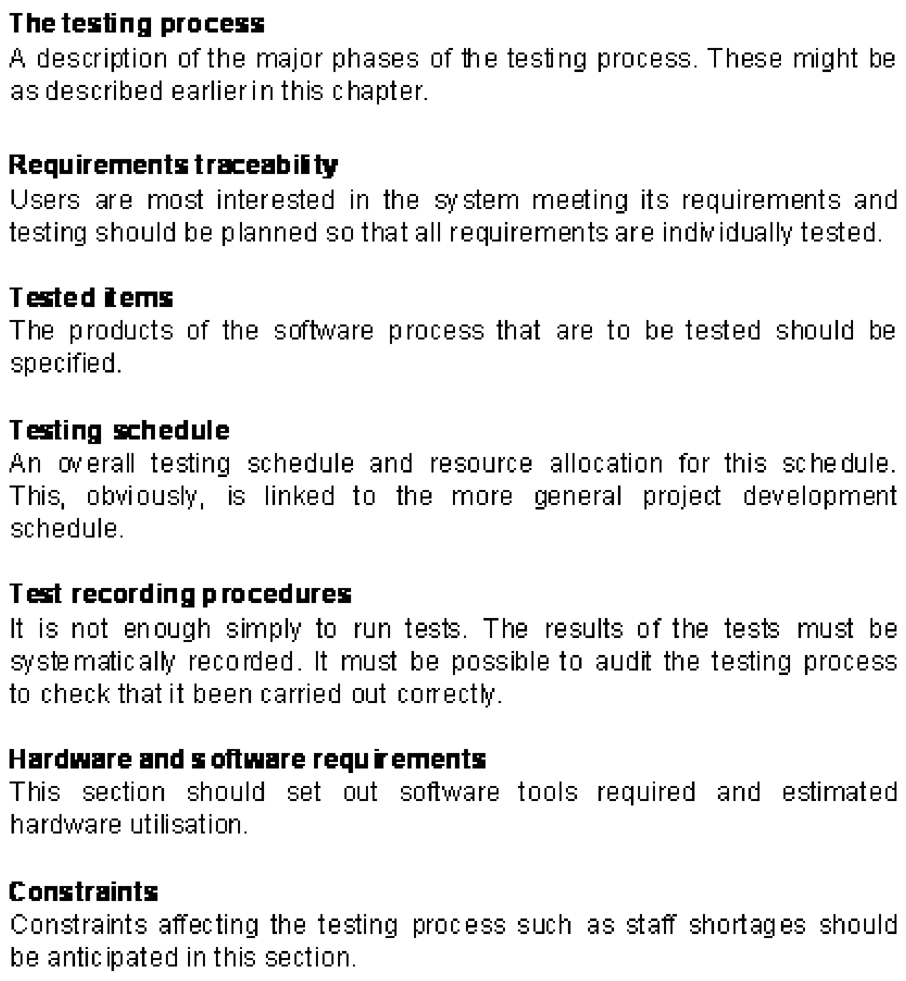

Requir ements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

Figure Test plans as a link between development and testing

- Test planning is concerned with establishing standards for the testing process, not just with describing product tests. As well as helping managers allocate resources and estimate testing schedules, test plans are intended for software engineers involved in designing and carrying out system tests.
- The major components of a test plan for a large and complex system are shown in Figure
- Test plans should normally include significant amounts of contingency so that slippages in design and implementation can be accommodated and staff redeployed to other activities.

The structure of a software test plan:

- The testing process.
- Requirements traceability.
- Tested items.
- Testing schedule.
- Test recording procedures.
- Hardware and software requirements.
- Constraints.



The testing process
A description of the major phases of the testing process. These might be as described earlier in this chapter.

Requirements traceability
Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

Tested items
The products of the software process that are to be tested should be specified.

Testing schedule
An overall testing schedule and resource allocation for this schedule. This, obviously, is linked to the more general project development schedule.

Test recording procedures
It is not enough simply to run tests. The results of the tests must be systematically recorded. It must be possible to audit the testing process to check that it been carried out correctly.

Hardware and software requirements
This section should set out software tools required and estimated hardware utilisation.

Constraints
Constraints affecting the testing process such as staff shortages should be anticipated in this section.

Figure Structure of a software test plan

- Test plans are not static documents but evolve during the development process.
- Test plans change because of delays at other stages in the development process.
- If part of a system is incomplete, the system as a whole cannot be tested. You then have to revise the test plan to redeploy the testers to some other activity and bring them back when the software is once again available.

Software inspections

- These 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.
- There are three major advantages of inspection over testing:

➢ During testing, errors can mask (hide) other errors. Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

➢ 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. This obviously adds to the system development costs.

➢ 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. You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

Inspections and testing:

- 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.

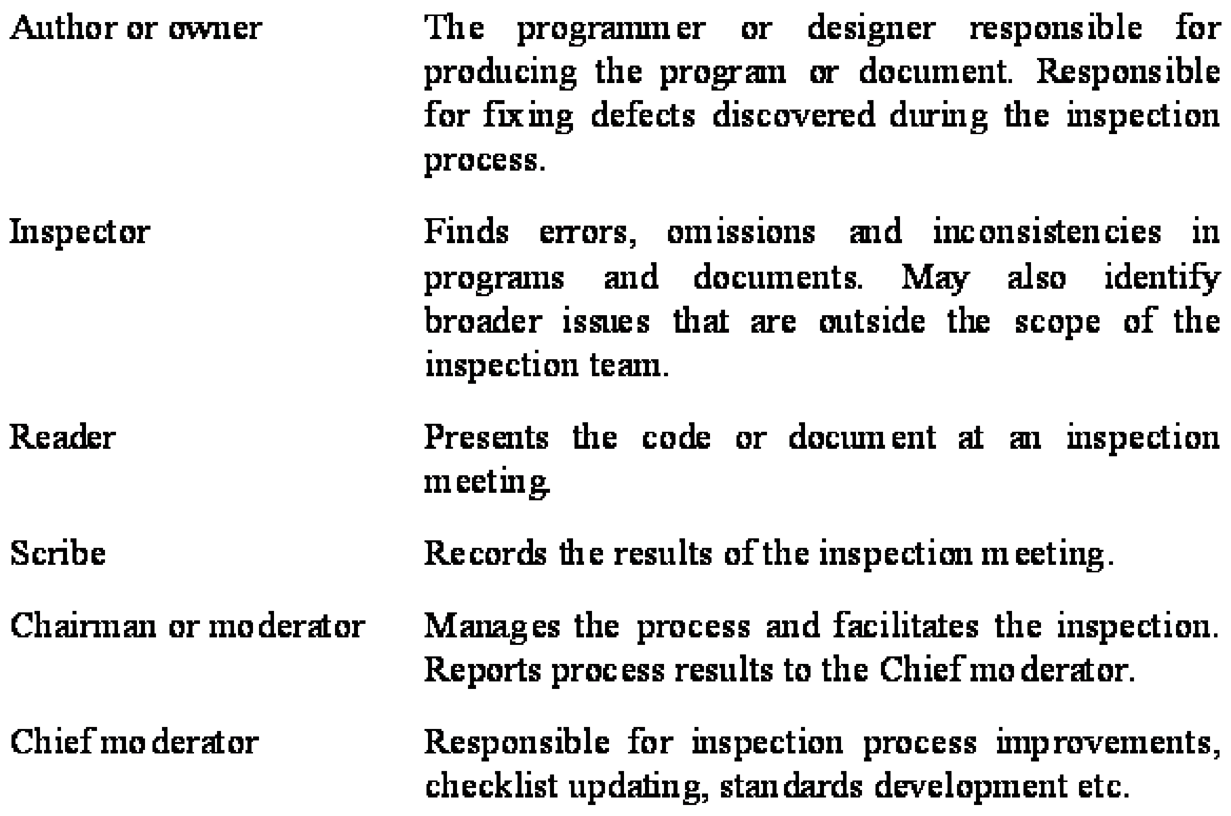
Program inspections

- Formalised approach to document reviews
- Intended explicitly for defect detection (not correction).
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards.

- The key difference between program inspections and other types of quality review is that the specific goal of inspections is to find program defects rather than to consider broader design issues.

Inspection pre-conditions:

- A precise specification must be available.
- Team members must be familiar with the organisation standards.
- Syntactically correct code or other system representations must be available.
- An error checklist should be prepared.
- Management must accept that inspection will increase costs early in the software process.
- Management should not use inspections for staff appraisal ie finding out who makes mistakes
- In a discussion of how inspection was successfully introduced in Hewlett-Packard's development process, Grady and Van Slack (Grady and Van Slack, 1994) suggest six roles, as shown below



Author or owner	The programmer or designer responsible for producing the program or document. Responsible for fixing defects discovered during the inspection process.
Inspector	Finds errors, omissions and inconsistencies in programs and documents. May also identify broader issues that are outside the scope of the inspection team.
Reader	Presents the code or document at an inspection meeting.
Scribe	Records the results of the inspection meeting.
Chairman or moderator	Manages the process and facilitates the inspection. Reports process results to the Chief moderator.
Chief moderator	Responsible for inspection process improvements, checklist updating, standards development etc.

Figure Roles in the inspection process

- The activities in the inspection process are shown in Figure 22.6.

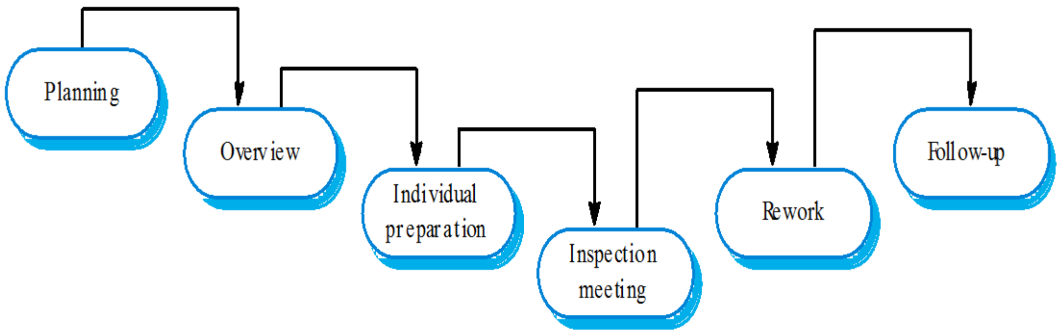
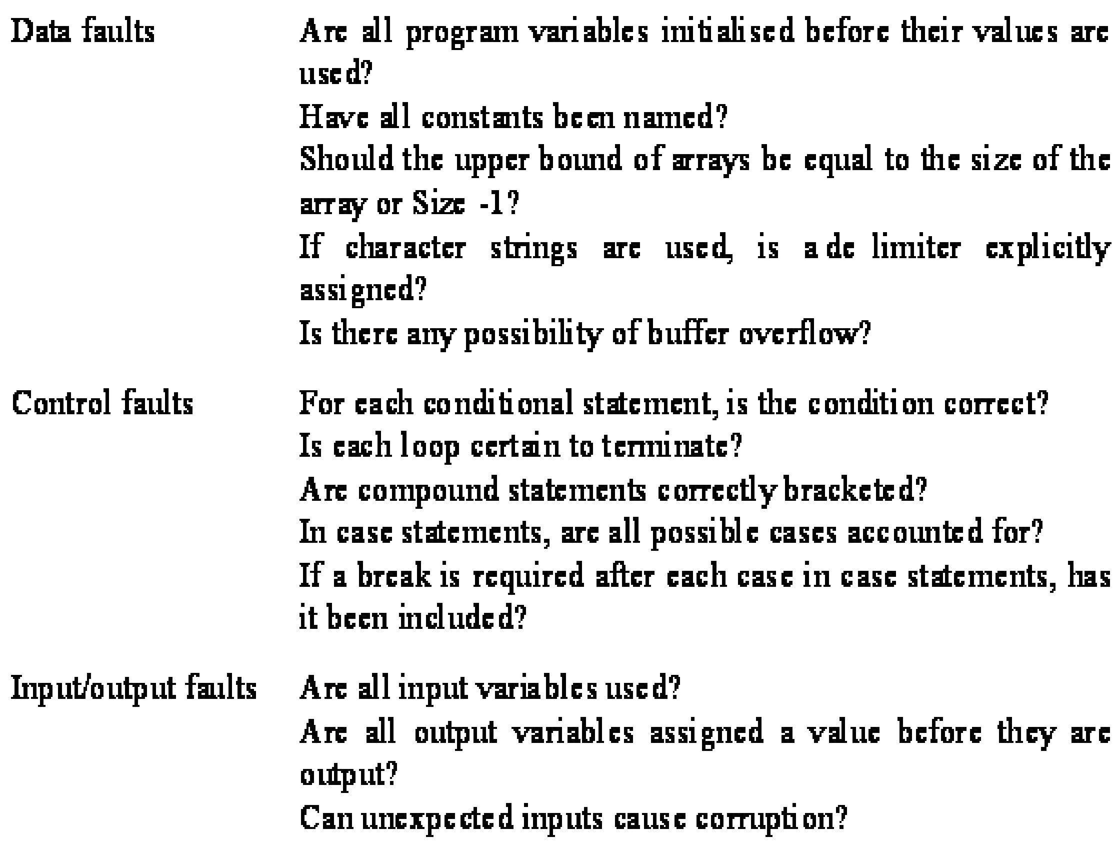
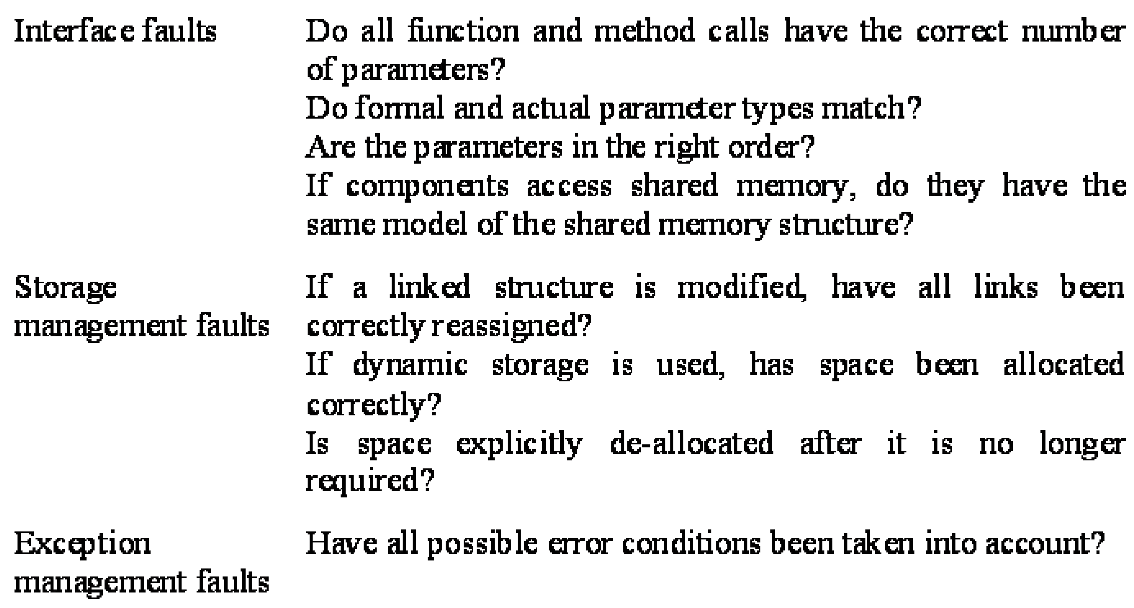


Figure Inspection process

- **Before a program Inspection process begins, it is essential that:**
  - You have a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
  - The inspection team members are familiar with the organisational standards.
  - An up-to-date, compliable version of the code has been distributed to all team members. There is no point in inspecting code that is 'almost complete' even if a delay causes schedule disruption.
- **Inspection checklists:**
  - Checklist of common errors should be used to drive the inspection.
  - Error checklists are programming language dependent and reflect the characteristic errors that are likely to arise in the language.
  - In general, the 'weaker' the type checking, the larger the checklist.
  - Examples: Initialisation, Constant naming, loop termination, array bounds, etc.
- Figure explains about the inspection checks



Data faults	Are all program variables initialised before their values are used? Have all constants been named? Should the upper bound of arrays be equal to the size of the array or Size -1? If character strings are used, is a de limiter explicitly assigned? Is there any possibility of buffer overflow?
Control faults	For each conditional statement, is the condition correct? Is each loop certain to terminate? Are compound statements correctly bracketed? In case statements, are all possible cases accounted for? If a break is required after each case in case statements, has it been included?
Input/output faults	Are all input variables used? Are all output variables assigned a value before they are output? Can unexpected inputs cause corruption?

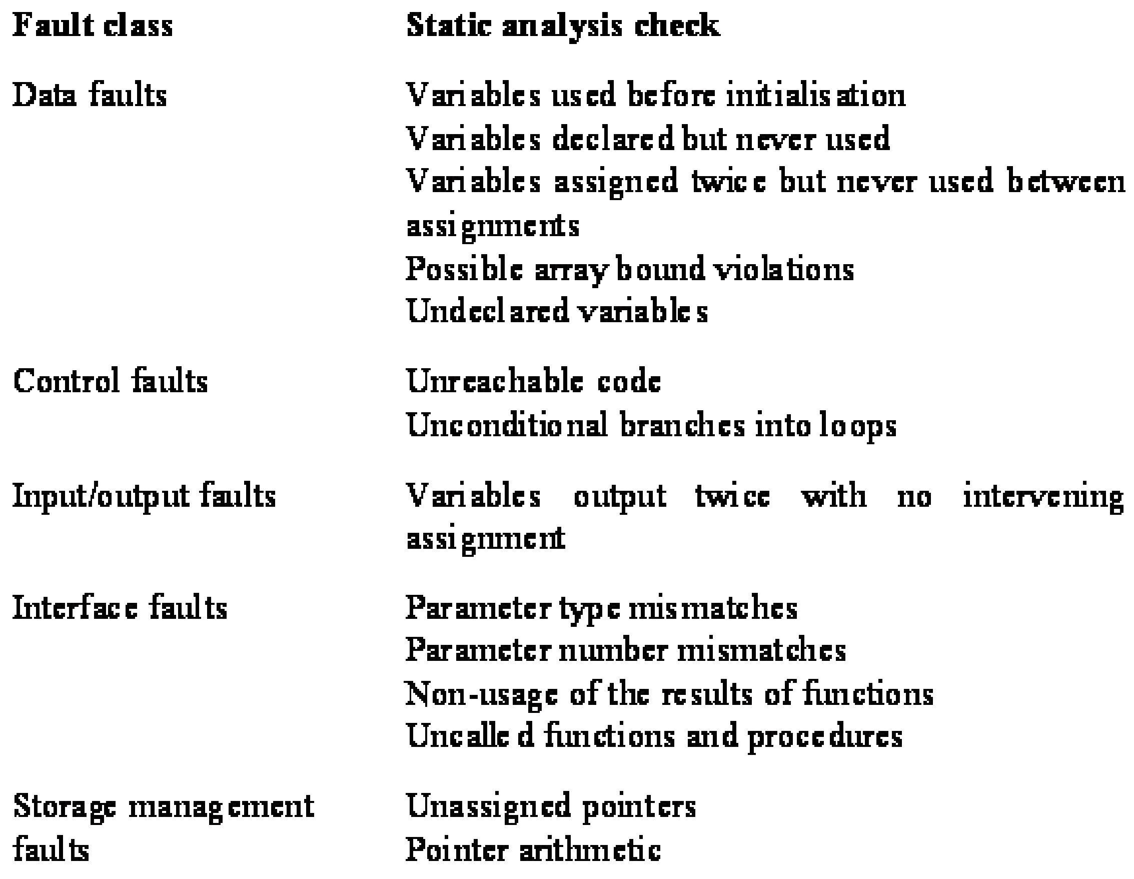


Interface faults	Do all function and method calls have the correct number of parameters? Do formal and actual parameter types match? Are the parameters in the right order? If components access shared memory, do they have the same model of the shared memory structure?
Storage management faults	If a linked structure is modified, have all links been correctly reassigned? If dynamic storage is used, has space been allocated correctly? Is space explicitly de-allocated after it is no longer required?
Exception management faults	Have all possible error conditions been taken into account?

Figure Inspection checks

Automated static analysis

- Static analysers are 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.
- Some of the checks that can Je detected by static analysis are shown in Figure 22.8.



Fault class	Static analysis check
Data faults	Variables used before initialisation 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

Figure Static analysis check automated static analysis checks

- The stages involved in static analysis include:
  - **Control flow analysis:** Checks for loops with multiple exit or entry points, finds unreachable code, etc.
  - **Data use analysis:** Detects uninitialized variables, variables written twice without an intervening assignment, variables which are declared but never used, etc.

- **Interface analysis:** Checks the consistency of routine and procedure declarations and their use.
- **Information flow analysis**: Identifies the dependencies of output variables. Does not detect anomalies itself but highlights information for code inspection or review
- **Path analysis:** Identifies paths through the program and sets out the statements executed in that path. Again, potentially useful in the review process

- **Use of static analysis:**
  - Particularly valuable when a language such as C is used which has weak typing and hence many errors are undetected by the compiler,
  - Less cost-effective for languages like Java that have strong type checking and can therefore detect many errors during compilation.

<u>Verification and formal methods</u>

- Formal methods of software development are based on mathematical representations of the software, usually as a formal specification.
- These formal methods are mainly concerned with a mathematical analysis of the specification; with transforming the specification to a more detailed, semantically equivalent representation; or with formally verifying that one representation of the system is semantically equivalent to another representation.
- Formal methods may be used at different stages in the V & V process:
  - A formal specification of the system may be developed and mathematically analyzed for inconsistency.
  - A transformational development process where a formal specification is transformed through a series of more detailed representations or a Cleanroom process may be used to support the formal verification process.

<u>Cleanroom software development</u>

- The name is derived from the 'Cleanroom' process in semiconductor fabrication. The philosophy is defect avoidance rather than defect removal.
- The Cleanroom approach to software development is based on five key strategies:

- **Incremental development:** The software is partitioned into increments that are developed and validated separately using the Cleanroom process. These increments are specified, with customer input, at an early stage in the process.
- **Formal specification:** The software to be developed is formally specified. A state-transition model that shows system responses to stimuli is used to express the specification.
- **Structured programming:** Only a limited number of control and data abstraction constructs are used. The program development process is a process of stepwise refinement of the specification.
- **Static verification:** The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
- **Statistical testing of the system:** The integrated software increment is tested statistically, to determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown:

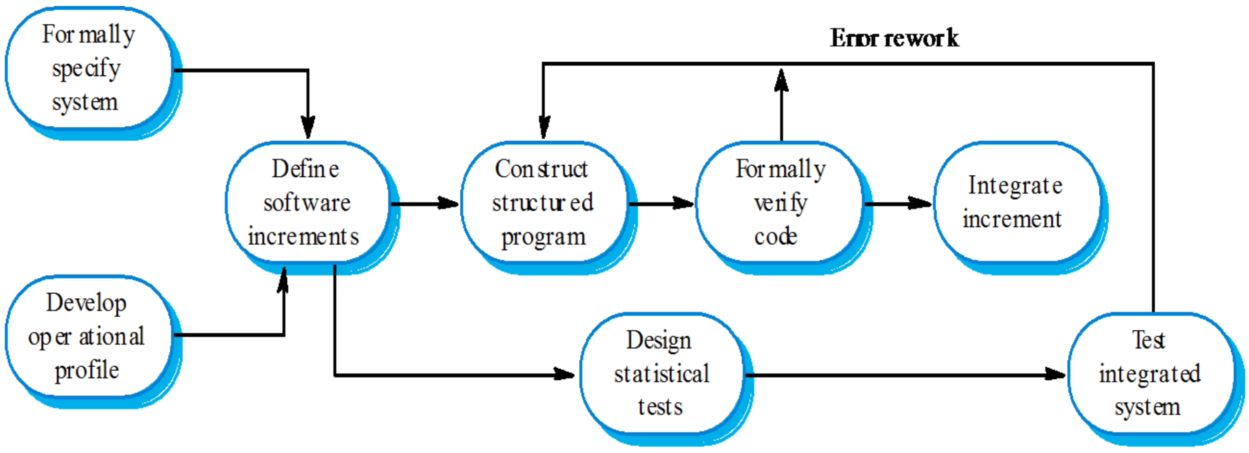


Figure The Cleanroom development process

Cleanroom process characteristics:

- Formal specification using a state transition model.
- Incremental development where the customer prioritises increments.
- Structured programming - limited control and abstraction constructs are used in the program.
- Static verification using rigorous inspections.
- Statistical testing of the system

Formal specification and inspections:

- The state based model is a system specification and the inspection process checks the program against this model.
- The programming approach is defined so that the correspondence between the model and the system is clear.
- Mathematical arguments (not proofs) are used to increase confidence in the inspection process.

Cleanroom process teams:

- **Specification team:** Responsible for developing and maintaining the system specification.
- **Development team:** Responsible for developing and verifying the software. The software is NOT executed or even compiled during this process.
- **Certification team**: Responsible for developing a set of statistical tests to exercise the software after development. Reliability growth models used to determine when reliability is acceptable.

Cleanroom process evaluation

- The results of using the Cleanroom process have been very impressive with few discovered faults in delivered systems.
- Independent assessment shows that the process is no more expensive than other approaches.
- There were fewer errors than in a 'traditional' development process.
- However, the process is not widely used. It is not clear how this approach can be transferred to an environment with less skilled or less motivated software engineers.

Software Testing

Objectives:

- To discuss the distinctions between validation testing and defect testing.
- To describe the principles of system and component testing.
- To describe strategies for generating system test cases.
- To understand the essential characteristics of tool used for test automation.

There are two testing activities:

- **Component testing**
  - ➢ Testing of individual program components;
  - ➢ Usually the responsibility of the component developer (except sometimes for critical systems);
  - ➢ Tests are derived from the developer's experience.
- **System testing**
  - ➢ Testing of groups of components integrated to create a system or sub-system;
  - ➢ The responsibility of an independent testing team;
  - ➢ Tests are based on a system specification.
- A more abstract view of software testing is shown in Figure

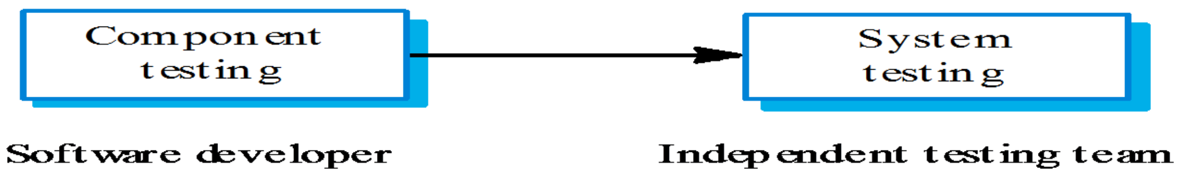


Figure Testing phases

Defect testing:

- The goal of defect testing is to discover defects in programs.
- A successful defect test is a test which causes a program to behave in an anomalous way.
- Tests show the presence not the absence of defects.

Testing process goals:

- **Validation testing**
  - ➢ To demonstrate to the developer and the system customer that the software meets its requirements.

- A successful test shows that the system operates as intended.

- **Defect testing**
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification;
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.
- A general model of the testing process is as shown below

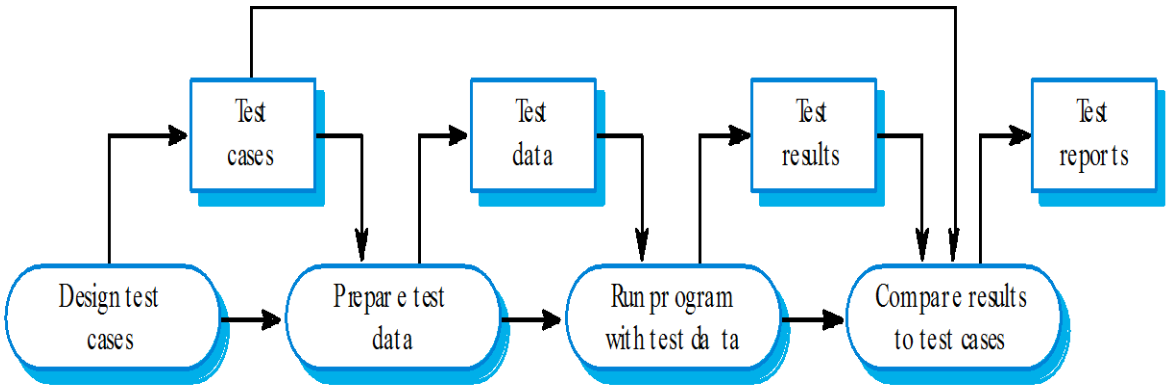


Figure A model of the software testing process

Testing policies:

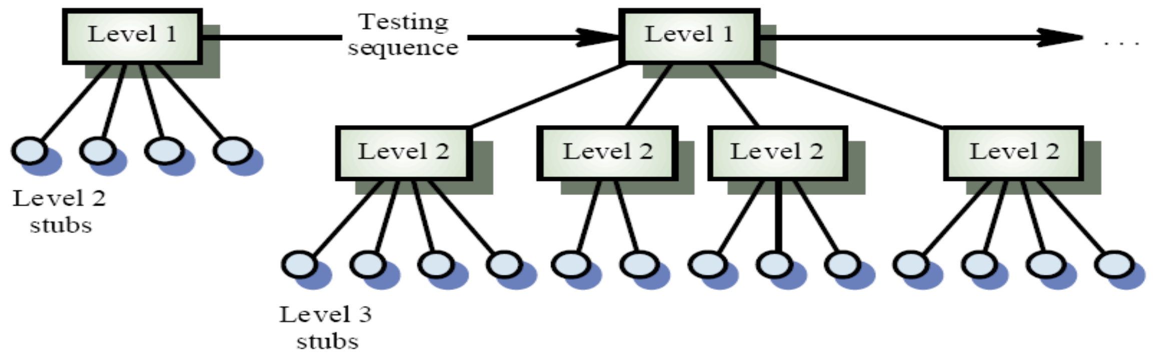
- Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible.
- Testing policies define the approach to be used in selecting system tests:
  - All functions accessed through menus should be tested;
  - Combinations of functions accessed through the same menu should be tested;
  - Where user input is required, all functions must be tested with correct and incorrect input.

System testing

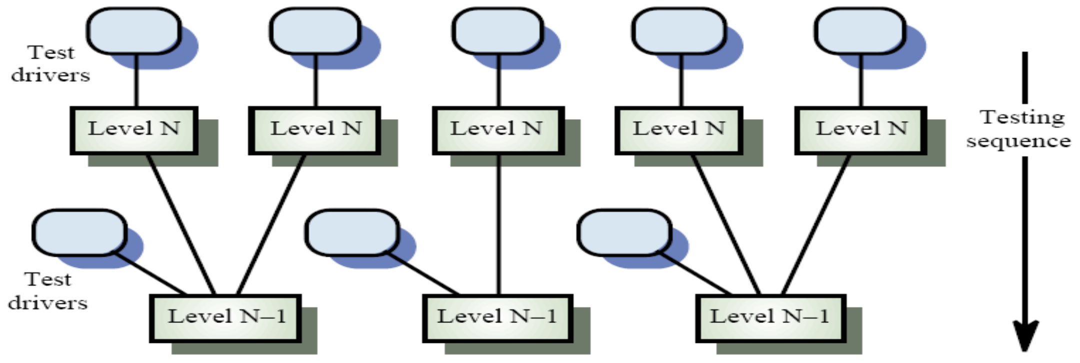
- Involves integrating components to create a system or sub-system.
- May involve testing an increment to be delivered to the customer.
- For most complex systems, there are two distinct phases to system testing:
  - **Integration testing:** The test team has access to the system source code. The system is tested as components are integrated.
  - **Release testing:** The test team test the complete system to be delivered as a lack-box.

Integration testing

- Involves building a system from its components and testing it for problems that arise from component interactions.
- There are two ways to do integration:
    - ➢ Top-down integration: Develop the skeleton of the system and populate it with components.



    - ➢ Bottom-up integration: Integrate infrastructure components then add functional components**.**



- To simplify error localisation, systems should be incrementally integrated.
- To make it easier to locate errors, you should always use an incremental approach to system integration and testing. Initially, you should integrate a minimal system configuration and test this system.
- To make it easier to locate errors, you should always use an incremental approach to system integration and testing.

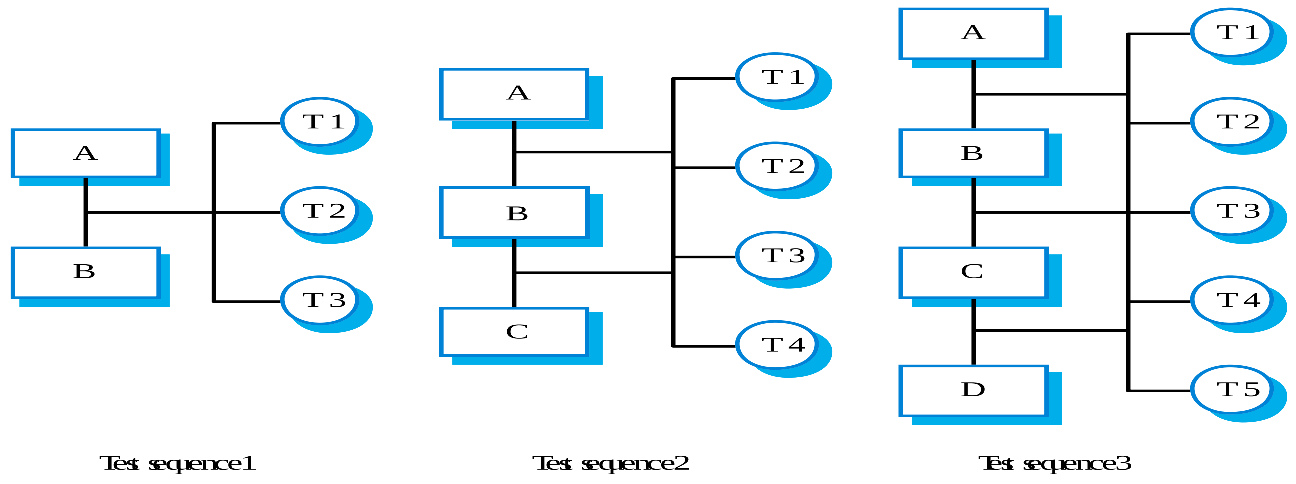


Figure Incremental integration testing

- **Architectural validation**
  - ➢ Top-down integration testing is better at discovering errors in the system architecture.
- **System demonstration**
  - ➢ Top-down integration testing allows a limited demonstration at an early stage in the development.
- **Test implementation**
  - ➢ Often easier with bottom-up integration testing.
- **Test observation**
  - ➢ Problems with both approaches. Extra code may be required to observe tests.

Release testing

- The process of testing a release of a system that will be distributed to customers.
- Primary goal is to increase the supplier's confidence that the system meets its requirements.
- Release testing is usually black-box or functional testing:
  - ➢ Based on the system specification only.
  - ➢ Testers do not have knowledge of the system implementation.
- Figure illustrates the model of a system that is assumed in black-box testing.

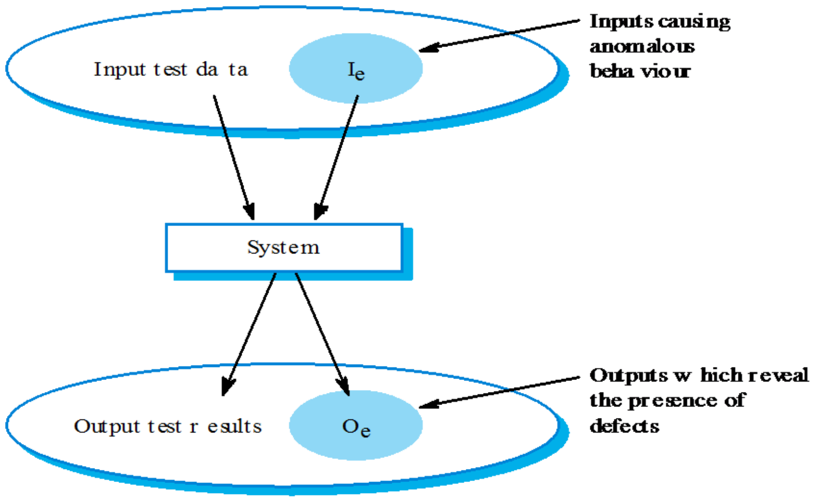
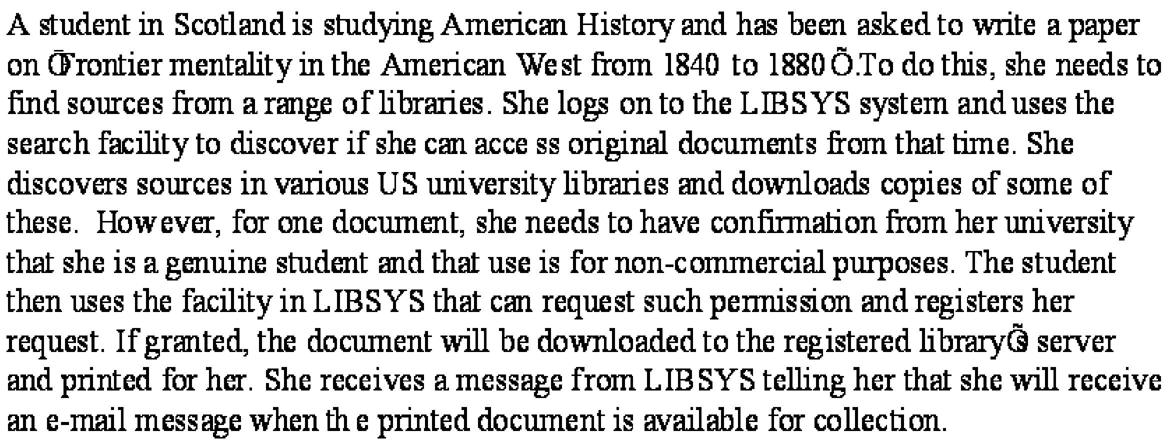

Input test da ta
Ie
Inputs causing anomalous beha viour
System
Output test r esults
Oe
Outputs w hich reveal the presence of defects

Figure Black box testing

- Authors such as Whittaker (Whittaker, 2(02) have encapsulated their testing experience in a set of guidelines that increase the probability that the defect tests will be successful. Some examples of these guidelines are:
  - ➢ Choose inputs that force the system to generate all error messages.
  - ➢ Design inputs that cause buffers to overflow.
  - ➢ Repeat the same input or input series several times.
  - ➢ Force invalid outputs to be generated.
  - ➢ Force computation results to be too large or too small.

<u>Testing scenario:</u>



A student in Scotland is studying American History and has been asked to write a paper on ŒFrontier mentality in the American West from 1840 to 1880Õ.To do this, she needs to find sources from a range of libraries. She logs on to the LIBSYS system and uses the search facility to discover if she can acce ss original documents from that time. She discovers sources in various US university libraries and downloads copies of some of these. However, for one document, she needs to have confirmation from her university that she is a genuine student and that use is for non-commercial purposes. The student then uses the facility in LIBSYS that can request such permission and registers her request. If granted, the document will be downloaded to the registered libraryÕs server and printed for her. She receives a message from LIBSYS telling her that she will receive an e-mail message when th e printed document is available for collection.

- From this scenario, it is possible to device a number of tests that can be applied to the proposed release of LIBSYS:

- ➢ Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
- ➢ Test the search facility using queries against known sources to check that the search mechanism is actually finding documents.
- ➢ Test the system presentation facility to check that information about documents is displayed properly.
- ➢ Test the mechanism to request permission for downloading.
- ➢ Test the e-mail response indicating that the downloaded document is available.

- For each of these tests, you should design a set of tests that include valid and invalid inputs and that generate valid and invalid outputs.
- If you have used use-cases to describe the system requirements, these use-cases and associated sequence diagrams can be a basis for system testing. The use-cases and sequence charts can be used during both integration and release testing.
- Figure shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system.

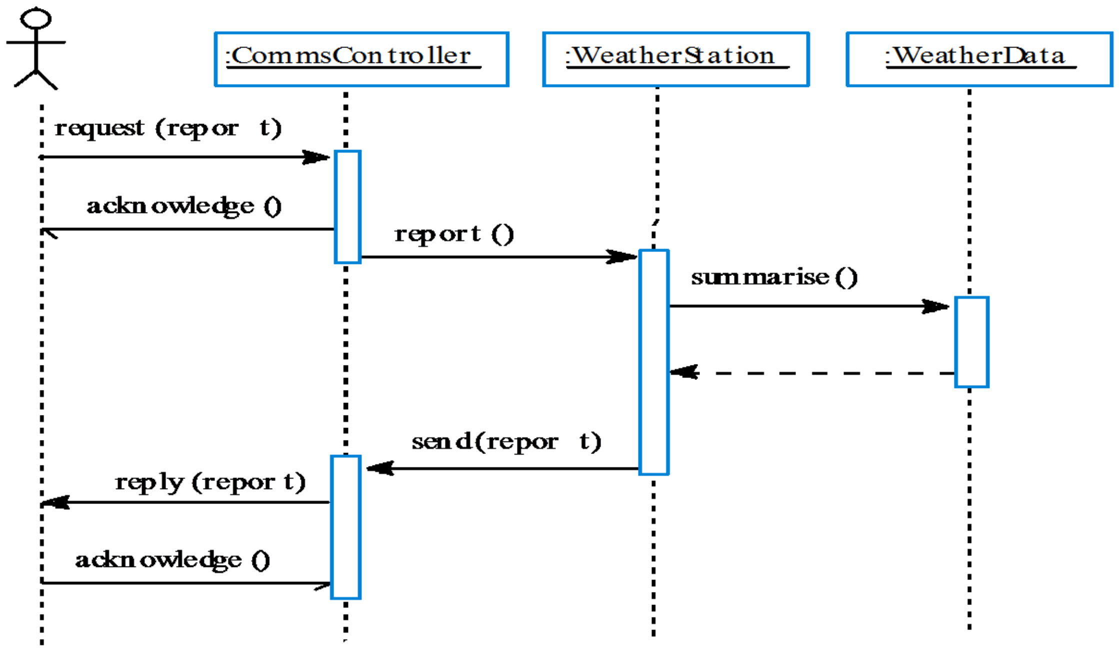


Figure Collect weather data sequence chart

- The sequence diagram can also be used to identify inputs and outputs that have to be created for the test:
  - An Input of a request for a report should have an associated acknowledgement and a report should ultimately be returned from the request.
  - During the testing you should create summarized data that can be used to check that the report is correctly organized.
  - An input request for a report to WeatherStation results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of CommsController and checking that the WeatherStation object correctly produces this summary.
  - This raw data is also used to test the WeatherData object.

Performance Testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- This type of testing has two functions:
  - It tests the failure behaviour of the system. Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.
  - It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates.

Component Testing

- Component or unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- There are different types of component that may be tested at this stage:
  - Individual functions or methods within an object;
  - Object classes with several attributes and methods;

- Composite components with defined interfaces used to access their functionality.

Object class testing:

- Complete test coverage of a class involves
  - Testing all operations associated with an object.
  - Setting and interrogating all object attributes.
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localized.
- Figure shows the Weather station object interface:

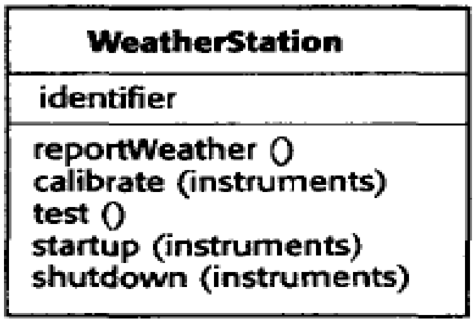


Figure Weather station object interface

Weather station testing:

- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions.
- For example:
  - Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Particularly important for object-oriented development as objects are defined by their interfaces.

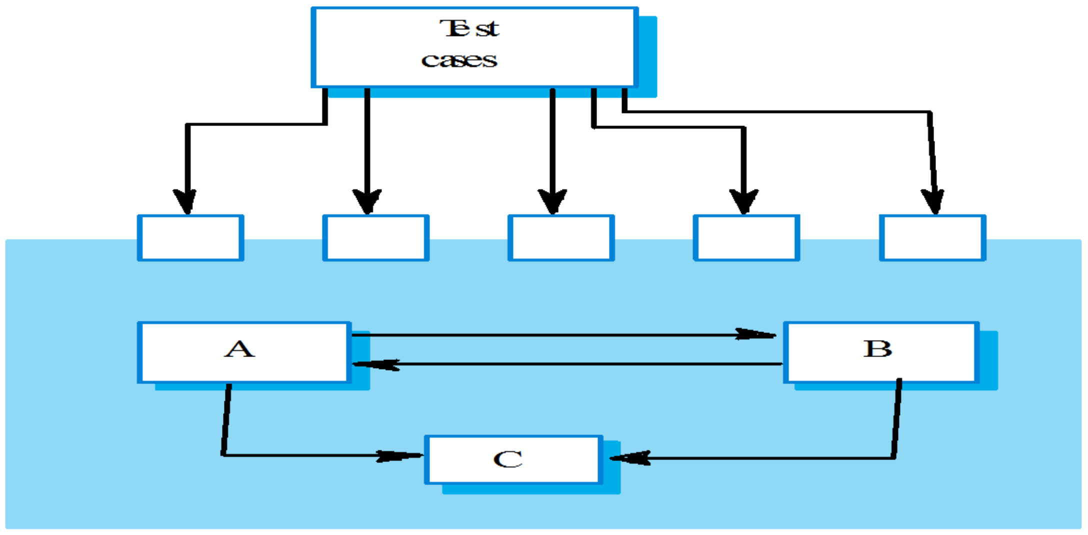


Figure Interface testing

- There are different types of interfaces between program components and, consequently, different types of interface errors that can occur:

  ➢ Parameter interfaces: These are interfaces where data or sometimes function references are passed from one component to another.

  ➢ Shared memory interfaces: These are interfaces where a block of memory is shared between components. Data is placed in the memory by one sub-system and retrieved from there by other sub-systems.

  ➢ Procedural interfaces: These are interfaces where one component encapsulates reusable components have this form of interface.

  ➢ Message passing interfaces: These are interfaces where one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client-server systems.

- Interface errors are one of the most common forms of error in complex systems, These errors fall into three classes:

  ➢ 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.

- **Interface testing guidelines:**
  - 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.