7th unit

22. Verification and validation

Contents

Validation: Are we building the right product? The aim of validation is to ensure that the software system meets the customer's expectations. It goes beyond checking that the system conforms to its specification to showing that the software does what the customer expects it to do

Verification: means Are we building the product right?', the role of verification involves checking that the software conforms to its specification and meets its specified functional and non-functional requirements.

The ultimate goal of the verification and validation process is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use.

The level of required confidence depends on the system's purpose, the expectations of the system users and the current marketing environment for the system:

1. ***Software function*** The level of confidence required depends on how critical the software is to an organisation.
   Example, the level of confidence required for software that is used to control a safety-critical system is very much higher than other type of software systems.

2. ***User expectations*** many users have low expectations of their software . They are willing to accept the system failures when the benefits of use outweigh the disadvantages. However, user tolerance of system failures has been decreasing since the 1990s. It is now less acceptable to deliver unreliable systems, so software companies must devote more effort to verification and validation.

3. ***Marketing environment*** When a system is marketed, the sellers of the system must take into account competing programs, the price those customers are willing to pay for a system and the required schedule for delivering that system. Where a company has few competitors, it may decide to release a program at lower prices before it has been fully tested and debugged because they want to be the first into the market.

Where customers are not willing to pay high prices for software, they may be willing to tolerate more software faults. All of these factors must be considered when deciding how much effort should be spent on the V & V process.

Within the V & V process, there are two complementary approaches to system checking and analysis:

1. ***Software inspections or peer reviews*** analyse and check system representations such as the requirements document, design diagrams and the program source code. Inspections can be used at all stages of the V & V process.

Software inspections are static V & V techniques, where no need to run the software on a computer.

2. ***Software testing:*** it *involves* running an implementation of the software with test data. The outputs of the software and its operational behaviour are examined to check that it is performing as required.

Testing is a dynamic technique of verification and validation.

Figure 22.1 shows that software inspections and testing play complementary roles in the software process. The arrows indicate the stages in the process where the techniques may be used.

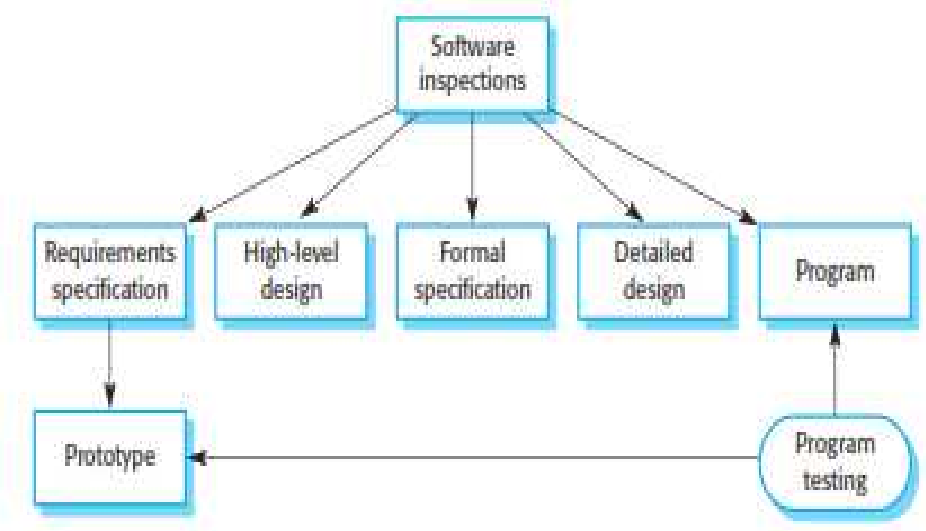


Figure 22.1 Static and dynamic verification and validation

. There are two distinct types of testing that may be used at different stages in the software process:

1. *Validation testing* is intended to show that the software is what the customer wants—that it meets its requirements.

2. *Defect testing* is intended to reveal defects in the system rather than to simulate its operational use. The goal of defect testing is to find inconsistencies between a program and its specification.

Verification & Validation Vs Debugging

1. Verification and validation processes are intended to establish the existence of defects in a software system.

2. Debugging is a process (Figure 22.2) that locates and corrects these defects.

Skilled debuggers look for patterns in the test output where the defect is exhibited and use their knowledge to locate the defect.

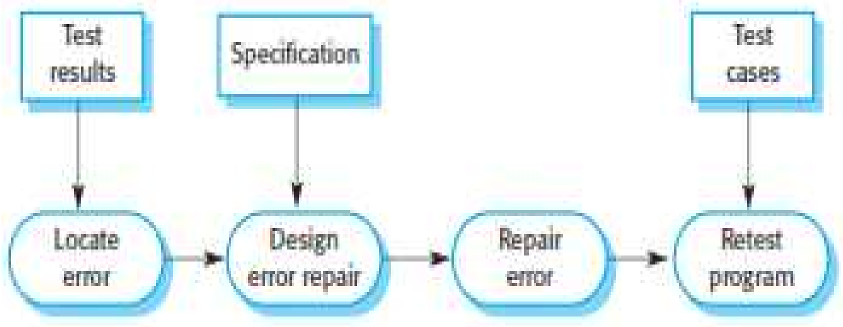


Figure 22.2 The debugging process

22.1 Planning verification and validation

1. Verification and validation is an expensive process. Careful planning is needed to get the Most out of inspections and testing and to control the costs of the verification and validation process.

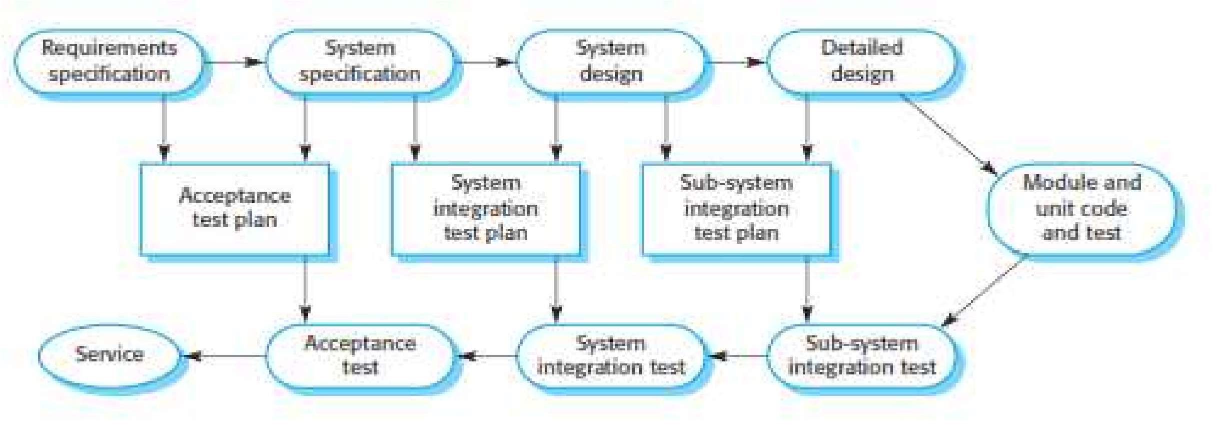

Requirements specification
System specification
System design
Detailed design
Acceptance test plan
System integration test plan
Sub-system integration test plan
Module and unit code and test
Service
Acceptance test
System integration test
Sub-system integration test

Figure 22.3 Test plans as a link between development and testing

2. The software development process model is shown in Figure 22.3 is sometimes called the **V-model** (turn Figure 22.3 on end to see the V).

Shows that test plans should be derived from the system specification and design. This model also breaks down system V & V into a number of stages. Each stage is driven by tests that have been defined to check the conformance of the program with its design and specification.

3. Test planning is concerned with establishing standards for the testing process, not just with describing product tests. As well as helping managers to allocate resources and estimate testing schedules.

4. Test planning help technical staff to get an overall picture of the system tests and place their own work in this context.

5. The major components of a test plan for a large and complex system are shown in Figure 22.4. As well as setting out the testing schedule and procedures.

- The test plan defines the hardware and software resources that are required. This is useful for system managers who are responsible for ensuring that these resources are available to the testing team.

- 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.

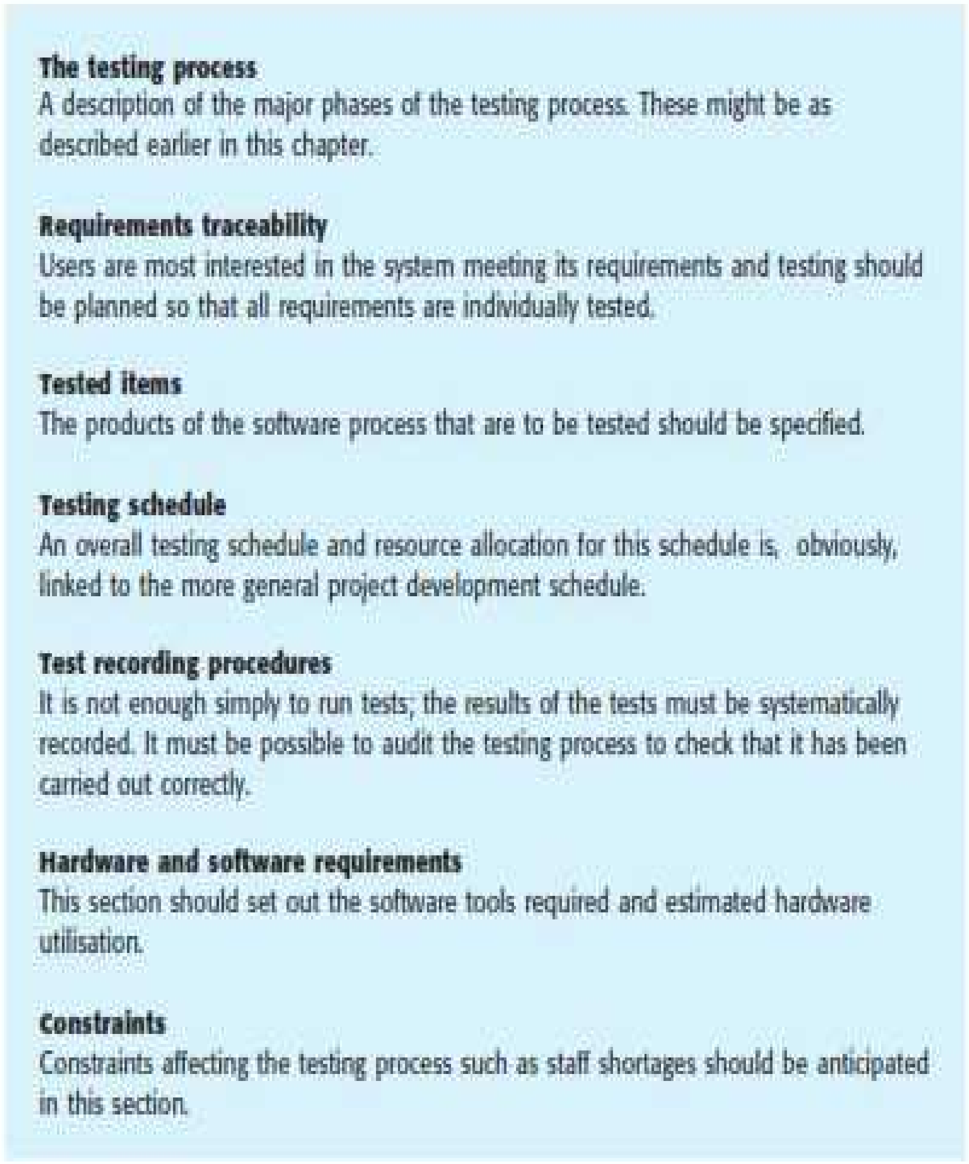


Figure 22.4 The structure of a software test plan

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 is, obviously, 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 has been carried out correctly.

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

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

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

22.2 Software inspections

1. Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies. Inspecting a system, use knowledge of the system,its application domain and the programming language or design model to discover errors.

2. There are three major advantages of inspection over testing:

a). During testing, errors can mask (hide) other errors. Once one error is discovered, it can be a new error or are side effects of the original error.

Because inspection is a static process, it is not concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

b). Incomplete versions of a system can be inspected without additional costs.

If a program is incomplete, then you need to develop specialised test harnesses to test the parts that are available. This obviously adds to the system development costs in testing.

c). As well as searching for program defects, an inspection can consider broader quality attributes of a program such as compliance with standards, portability and maintainability.

3. Start system V & V with inspections early in the development process, but once a system is integrated, testing is needed to check its emergent properties and system's functionality is what the owner of the system really wants.

4. **Disadvantage**

- Difficult to introduce formal inspections into many software development organisations.
- Software engineers with experience of program testing are sometimes reluctant to accept that inspections can be more effective for defect detection than testing.
- Managers may be suspicious because inspections require additional costs during design and development. They may not wish to take the risk that there will be no corresponding savings during program testing.
- Inspections take time to arrange and appear. It is difficult to convince a hard-pressed manager that this time can be made up later because less time will be spent on program debugging.

22.2.1 The program inspection process

1. Program inspections are reviews whose objective is program defect detection. The notion of a formalised inspection process was first developed at IBM in the 1970s.

2.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.

By contrast, other types of review may be more concerned with schedule,costs, progress against defined milestones or assessing whether the software is likely to meet organisational goals.

3. The program inspection is a formal process that is carried out by a team of at least four people. Team members systematically analyse the code and point out possible defects.

4. In Fagan's original proposals, he suggested roles such as author, reader, tester and moderator.

The reader reads the code aloud to the inspection team

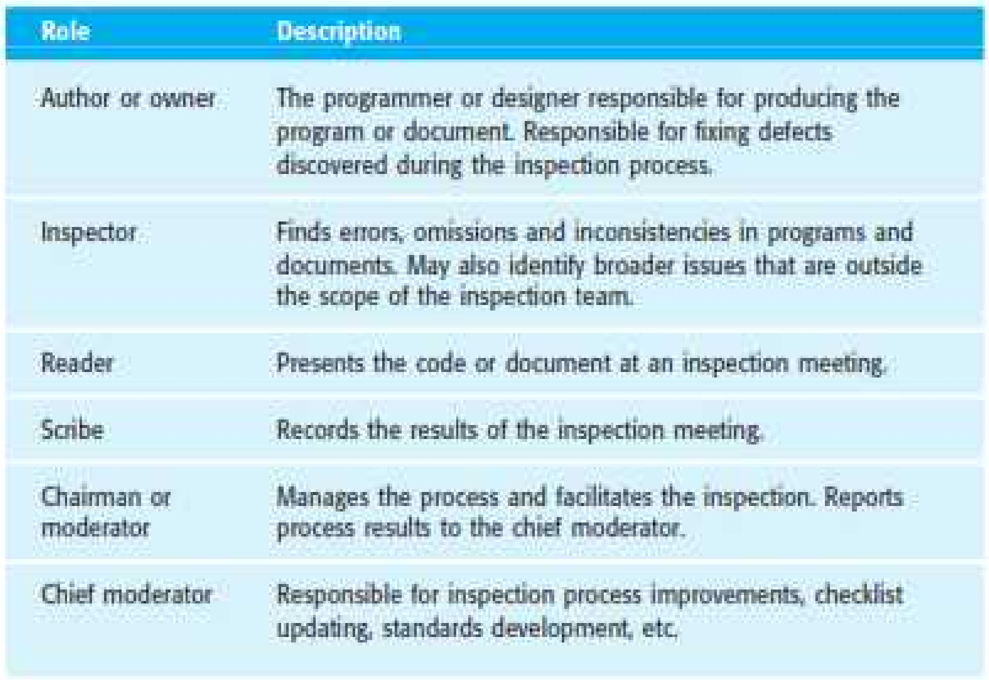
The tester inspects the code from a testing perspective

The moderator organises the process.

5. Grady and Van Slack suggest six roles, as shown in Figure 22.5. Here the same person can take more than one role so the team size may vary from one inspection to another.

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

Figure 22.5 Roles in the inspection process



Role	Description
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.

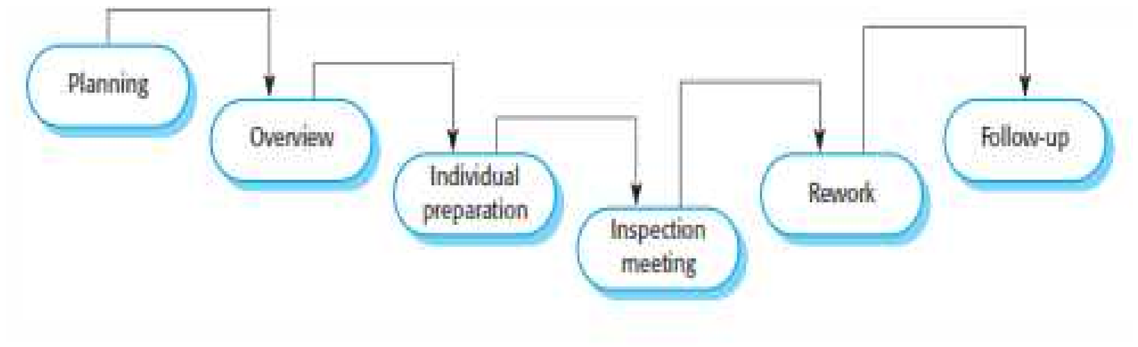


Figure 22.6 The inspection process

6. Inspection pre-condition

Before a program inspection process begins, it is essential that:

a). to 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.

b). The inspection team members should be familiar with the organisational standards.

c). An up-to-date, compilable version of the code has been distributed to all team members for inspecting .

7. The inspection process

a) The inspection team moderator is responsible for inspection planning. This involves selecting an inspection team, organising a meeting room and ensuring that the material to be inspected and its specifications are complete.

b) The program to be inspected is presented to the inspection team during the overview stage when the author of the code describes what the program is intended to do.

c) This is followed by a period of individual preparation. Each inspection team member studies the specification and the program.

d) The inspection carried out itself should be fairly short. Each inspection team member looks for defects in the code .The inspection team should not suggest how these defects should be corrected nor should it recommend changes to other components.

e) Following the inspection, the program's author should make changes to it to correct the identified problems.

f) In the follow-up stage, the moderator should decide whether a reinspection of the code is required. He or she may decide that a complete reinspection is not required and that the defects have been successfully fixed.

The program is then approved by the moderator for release.

8. Checklists

a) checklist is a list of common programmer errors.

b). During an inspection, a checklist of common programmer errors is often used to focus the discussion.

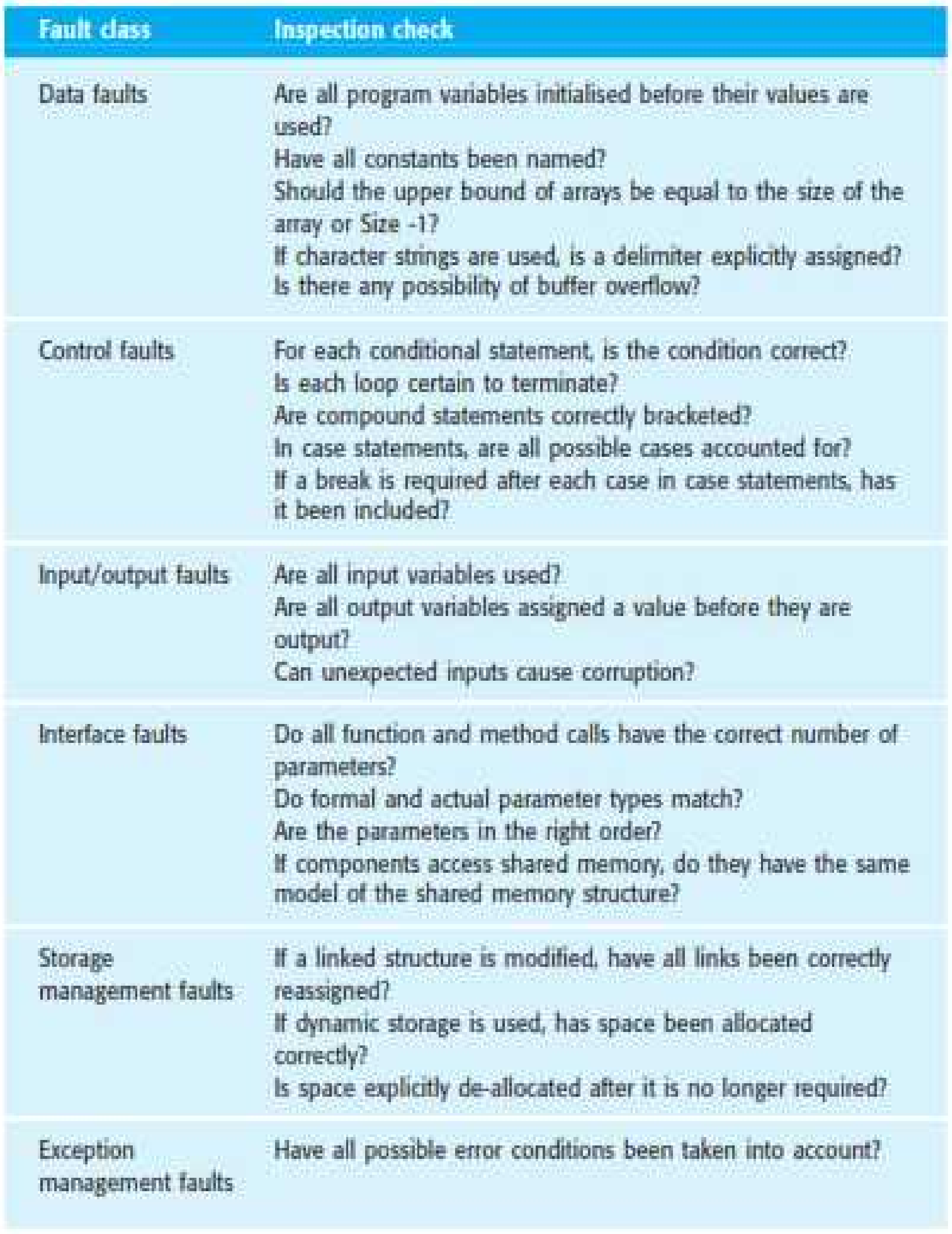
c) This checklist varies according to programming language because of the different levels of checking provided by the language compiler.

d) Possible checks that might be made during the inspection process are shown In Figure 22.7 for different fault class.

e) Each organization should develop its own inspection checklist based on local standards and practices. Checklists should be regularly updated as new types of defects are found.

f) The time needed for an inspection and the amount of code that can be covered depends on the experience of the inspection team, the programming language and the application domain.

Figure 22.7
Inspection checks



Fault class	Inspection check
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 delimiter 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?

22.3 Automated static analysis

1. Inspections are one form of static analysis—you examine the program without executing it.

2. For some errors and heuristics, it is possible to automate the process of checking programs against the check list, which has resulted in the development of automated static analysersfor different programming languages.

3. Static analysers are software tools that scan the source text of a program and detect possible faults and anomalies. They parse the program text and thus recognise the types of statements in the program.

They can then detect whether statements are well formed, make inferences about the control flow in the program and, in many cases, compute the set of all possible values for program data.

They can be used as part of the inspection process or as a separate V & V process activity.

4. The intention of automatic static analysis is to draw an inspector's attention to anomalies in the program, such as variables that are used without initialisation, variables that are unused or data whose value could go out of range.

5. Some of the checks that can be detected by static analysis are shown in Figure 22.8.

6. The stages involved in static analysis include:

a). *Control flow analysis* This stage identifies and highlights loops with multiple exit or entry points and unreachable code. Unreachable code is code that is surrounded by unconditional goto statements or that is in a branch of a conditional statement where the guarding condition can never be true.

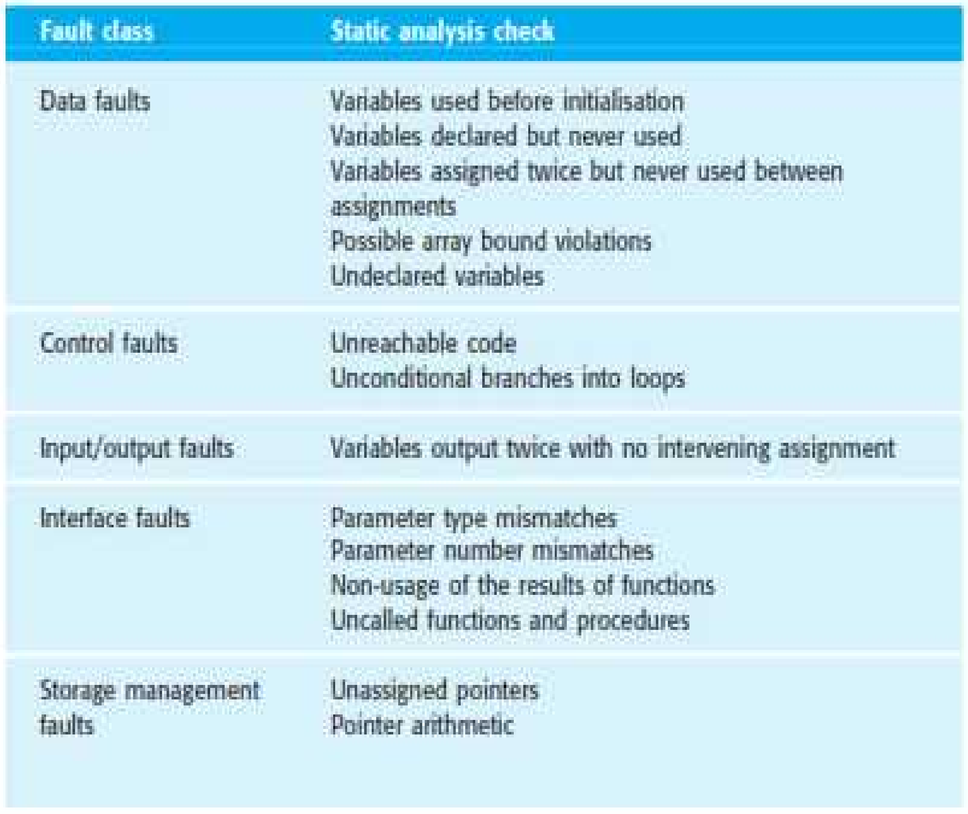
b). *Data use analysis* This stage highlights how variables in the program are used. It detects variables that are used without previous initialisation, variables that are written twice without an intervening assignment and variables that are declared but never used. Data use analysis also discovers ineffective tests where the test condition is redundant. Redundant conditions are conditions that are either always true or always false.

c). *Interface analysis* This analysis checks the consistency of routine and procedure declarations and their use. Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.

d). *Information flow analysis* This phase of the analysis identifies the dependencies between input and output variables. It shows how the value of each program variable is derived from other variable values. With this information, a code inspection should be able to find values that have been wrongly computed. Information flow analysis can also show the conditions that affect a variable's value.

d). *Path analysis* This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path. It essentially unravels the program's control and allows each possible predicate to be analysed individually.

Figure 22.8 Automated static analysis checks



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

22.4 Verification and formal methods

1. 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.

2. formal methods are the ultimate static verification technique. They require very detailed analyses of the system specification and the program, and their use is often time consuming and expensive. Consequently, the use of formal methods is mostly confined to safety- and security-critical software development processes.

3. Formal methods may be used at different stages in the V & V process:

a). A formal specification of the system may be developed and mathematically analysed for checking inconsistency. This technique is effective in discovering specification errors and omissions.

b) Is the code of a software system is consistent with its specification can be formally verified using mathematical statements and formal specification is effective in discovering programming and some design errors.

4. The argument against the use of formal specification is that :

a) It requires specialized notations. These can only be used by specially trained staff and cannot be understood by domain experts.

b) formal verification is not cost-effective. Verifying a nontrivial software system takes a great deal of time and requires specialised tools such as theorem provers and mathematical expertise.

5. Formal specification do not guarantee that the software will be reliable in **practical use**. The reasons for this are:

a) ***The specification may not reflect the real requirements of system users***. many failures experienced by users were a consequence of specification errors and omissions that could not be detected by formal system specification.

Furthermore, system users rarely understand formal notations so they cannot read the formal specification directly to find errors and omissions.

b). ***The proof may contain errors***. Program proofs are large and complex, so, like large and complex programs, they usually contain errors.

c). ***The proof may assume a usage pattern which is incorrect***. If the system is not used as anticipated, the proof may be invalid.

22.4.1 Cleanroom software development

1. Cleanroom software development is a software development philosophy that uses formal methods to support rigorous software inspection.

2. A model of the Cleanroom process is shown in Figure 22.10. The objective of this approach to software development is zero-defect software.

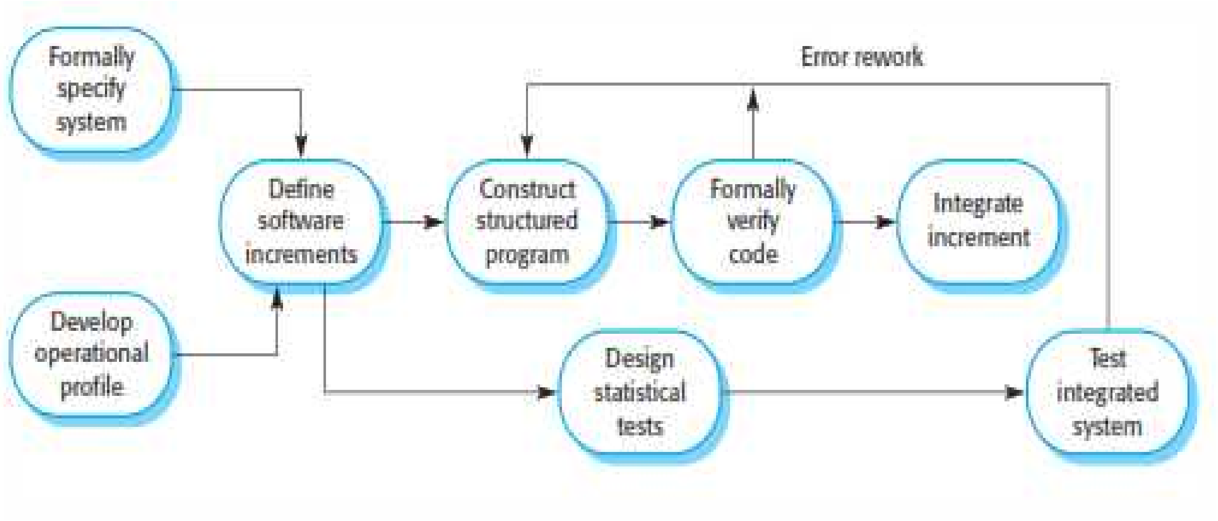


Figure 22.10 The Cleanroom development process

3. The Cleanroom approach to software development is based on five key strategies:

a). *Formal specification* The software to be developed is formally specified.

Example: statetransition model that shows system responses to stimuli is used to express the

specification.

b). *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.

c). ***Structured programming*** The program development process is a process of stepwise refinement of the specification. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.

d). *Static verification* The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.

e). *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 in Figure 22.10.

4. There are three teams involved when the Cleanroom process is used for large system development:

a). *The specification team* This group is responsible for developing and maintaining the system specification. This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification.

b). *The development team* This team has the responsibility of developing and verifying the software. The software is not executed during the development process.

c). *The certification team* This team is responsible for developing a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification. The test cases are used to certify the software reliability.

5. Advantages:

a) Use of the Cleanroom approach has generally led to software with very few errors.
b) leads to Rigorous program inspection.
c) Inspection and formal analysis has been found to be very effective in the Cleanroom process.
d) cost-effective because less effort is required to test and repair the developed software.

2nd chapter,7th unit
23 Software testing

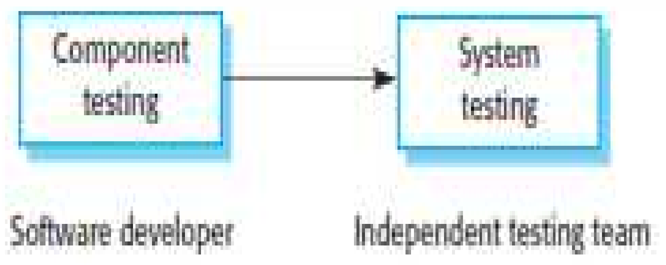
Contents

A more abstract view of software testing is shown in Figure 23.1. The two fundamental testing activities are

Component testing— testing the parts of the system. The aim of the component testing stage is to discover defects by testing individual program components. These components may be functions, objects or reusable components .

System testing— testing the system as a whole. During system testing, these components/sub-systems are integrated to form the complete system. At this stage, system testing should focus on establishing that the system meets its functional and non-functional requirements, and does not behave in unexpected ways.

Figure 23.1 Testing phases



The software testing process has two distinct goals:

1. ***To demonstrate to the developer and the customer that the software meets its requirements*.**

For custom software, this means that there should be at least one test for every requirement in the user and system requirements documents.

For generic software products, it means that there should be tests for all of the system features that will be incorporated in the product release.

2. ***To discover faults or defects in the software where the behaviour of the software***

***is incorrect, undesirable or does not conform to its specification*.** Defect testing is concerned with rooting out all kinds of undesirable system behaviour,

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.

The second goal leads to defect testing, where the test cases are designed to expose defects.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. m'Testing can only show the presence of errors, not their absence.' Testing is a process intended to build confidence in the software.

A general model of the testing process is shown in Figure 23.2.

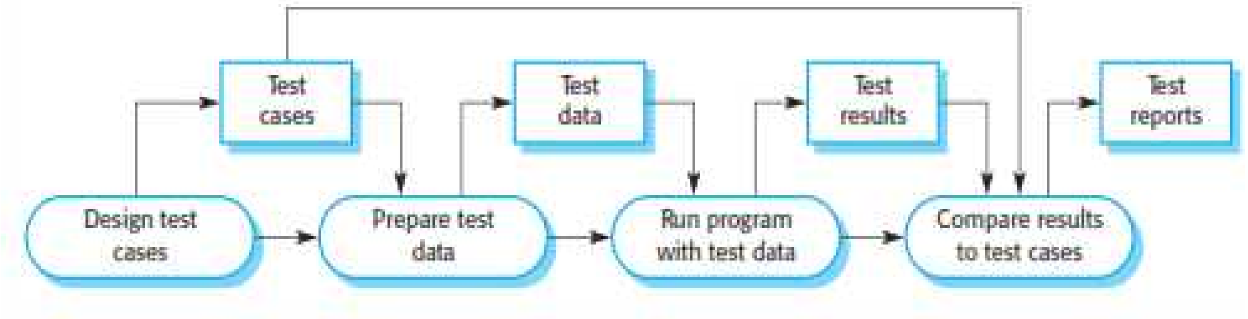


Figure 23.2 A model of the software testing process

-Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested.

- Test data are the inputs that have been devised to test the system. Test data can sometimes be generated automatically. Automatic test case generation is impossible.

23.1 System testing

1. System testing involves integrating two or more components that implement system functions or features and then testing this integrated system.

2. For most complex systems, there are two distinct phases to system testing:

a). *Integration testing,* where the test team have access to the source code of the system. When a problem is discovered, the integration team tries to find the source of the problem and identify the components that have to be debugged. Integration testing is mostly concerned with finding defects in the system.

b). *Release testing,* where a version of the system that could be released to users is tested. Here, the test team is concerned with validating that the system meets its requirements.. Release testing is usually 'black-box' testing where the test team is simply concerned with demonstrating that the system does or does not work properly.

Problems are reported to the development team whose job is to debug the program. Where

Customers are involved in release testing, this is sometimes called *acceptance testing*.

23.1.1 Integration testing

1. The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. The components that are integrated may be reusable components that have been adapted for a particular system or newly developed components.

2. Integration testing checks that these components actually work together correctly and transfer the right data at the right time across their interfaces.

3. System integration involves identifying clusters of components that deliver some system functionality and integrating these by adding code that makes them work together.

4. Types of integration

Top-down integration: here the overall skeleton of the system is developed first, and components are added to it

Bottom-up integration: here first the infrastructure components that provide common services, such as network and database access is integrated, then add the functional components.

5.Incremental approach to system integration

a) A major problem that arises during integration testing is localising errors.

b) It is difficult to identify localized errors because of complex interactions between the system components .

c) To make it easier to locate errors, an incremental approach to system integration and testing is used. Divide the integration process into increments.

-Initially, integrate a minimal system configuration and test this increment.

-Then a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required to verify the new system functionality.

-Rerunning an existing set of tests is called *regression testing*. If regression testing exposes problems, check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added increment of functionality.

d) In the example shown in Figure 23.3, A, B, C and D are components and T1 to T5 are related sets of tests of the features incorporated in the system.

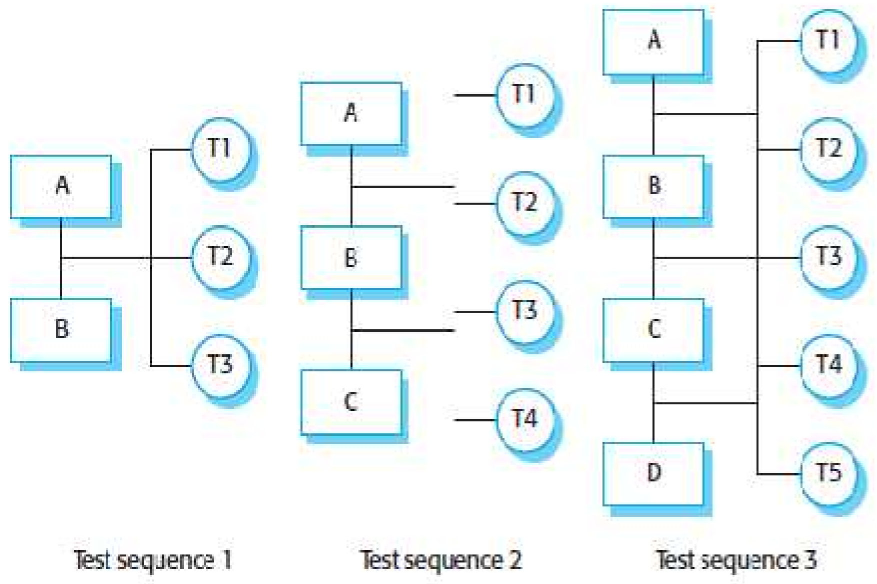
e) T1, T2 and T3 are first run on a system composed of component A and component B (the minimal system). If these reveal defects, they are corrected. Component C is integrated and T1, T2 and T3 are repeated to ensure that there have not been unexpected interactions with A and B.

If problems arise in these tests, this probably means that they are due to interactions with the new component.

f)The source of the problem is localised, thus simplifying defect location and repair. Test set T4 is also run on the system.

g) Finally, component D is integrated and tested using existing and new tests (T5).

Figure 23.3
Incremental
integration testing



6. When planning integration, first decide the order of integration of components. second involve the customer in the development process, this helps in deciding which functionality should be included in each system increment.

Example library system, LIBSYS,

First start by integrating the search facility so that, in a minimal system, users can search for documents that they need, then add the functionality to allow users to download a document, then progressively add the components that implement other system features.

23.1.2 Release testing (*functional testing)*

1. Release testing is the process of testing a release of the system that will be distributed to customers.

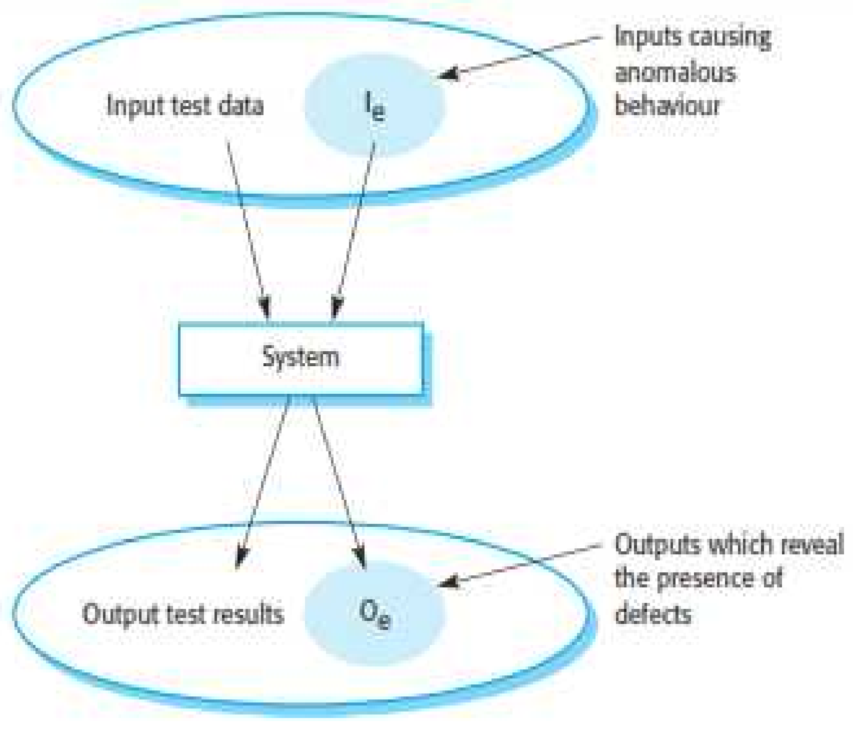
2. The primary goal of this process is to increase the supplier’s confidence that the system delivers the specified functionality, performance and dependability, and that it does not fail during normal use. If so, it can be released as a product or delivered to the customer.

3. Release testing is usually a black-box testing process where the tests are derived from the system specification, by studying its inputs and the related outputs.

4. In functional testing the tester is only concerned with the functionality and not the implementation of the software.

5. Figure 23.4 illustrates the model of a system .

Figure 23.4 Black-box testing


Inputs causing anomalous behaviour
Input test data
I_e
System
Outputs which reveal the presence of defects
Output test results
O_e

-The tester presents inputs to the component or the system and examines the corresponding outputs.

-If the outputs are not those predicted (i.e., if the outputs are in set Oe) then the test has detected a problem with the software.

-When testing system releases, try to 'break' the software by choosing test cases that are in the set Ie in Figure 23.4.

6. A set of guidelines that increase the probability that the defect tests will be successful.

1. Choose inputs that force the system to generate all error messages.
2. Design inputs that cause input buffers to overflow.
3. Repeat the same input or series of inputs numerous times.
4. Force invalid outputs to be generated.
5. Force computation results to be too large or too small.

7. Scenario-based testing

a).To validate that the system meets its requirements, the best approach to use is scenario-based testing, where number of scenarios can be devised and then develop test cases from these scenarios.

b) For example, consider searching and downloading of document by using the facility in LIBSYS, that can later request permission and registers request by filling copyright form. If granted, the document will be downloaded to the registered library's server and printed for user.

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

1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.

2. Test the search facility using queries against known sources to check that the search mechanism is actually finding documents.

3. Test the system presentation facility to check that information about documents is displayed properly.

4. Test the mechanism to request permission for downloading.

8. Use-case based testing.

a) The use-cases and sequence charts can be used during both integration and release testing.

b)Example: Figure 23.5 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system.

c) The diagram identify operations that will be tested and to help design the test cases to execute the tests. Therefore issuing a request for a report will result in the execution of the following thread of methods:

CommsController:request → WeatherStation:report → WeatherData:summarise

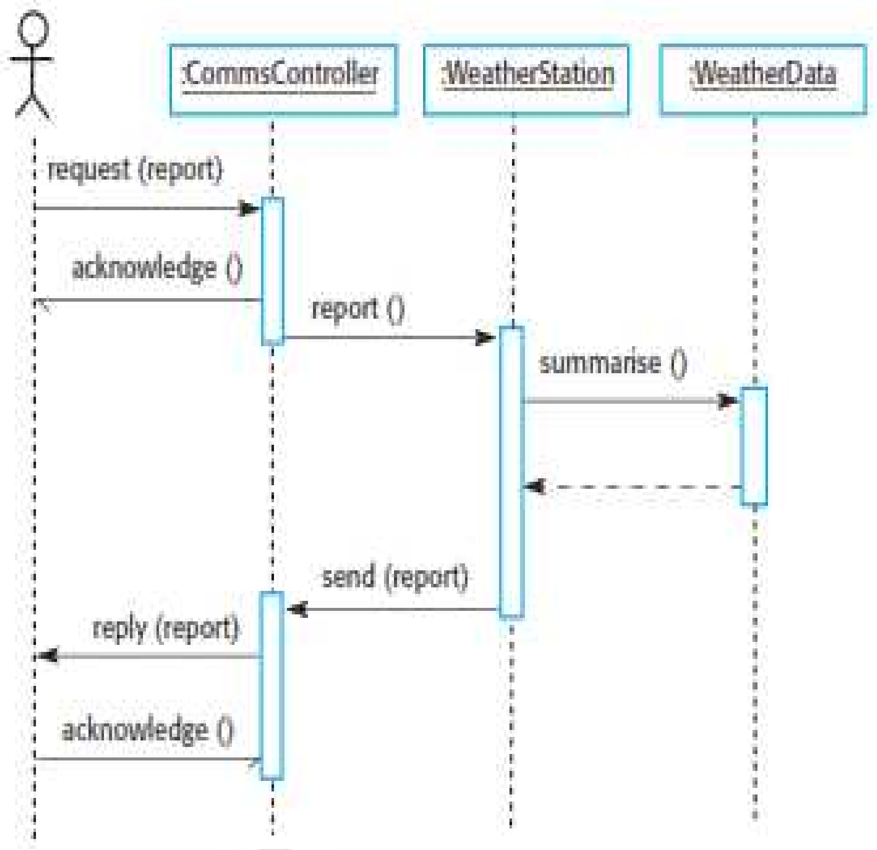
c) The sequence diagram can also be used to identify inputs and outputs that have to be created for the test:

1. 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 create summarised data that can be used to check that the report is correctly organised.

2. An input request for a report to WeatherStation results in a summarised report being generated. then test this in isolation by creating raw data corresponding to the summary prepared for the test of CommsController and checking that the WeatherStation object correctly produces this summary.

3. This raw data is also used to test the WeatherData object.

Figure 23.5 Collect weather data sequence chart



23.1.3 Performance testing/stress testing

1. Performance tests have to be designed to ensure that the system can process its intended load. **2.** This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

3. performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system.

4. Operational Profile- it is a set of tests that reflect the actual mix of work that will be handled by the system. An operational profile have been constructed to test whether performance requirements are being achieved

5. Effective way to discover defects is to design tests around the limits of the system. In performance testing, testing means stressing the system (hence the name *stress testing*) by making demands that are outside the design limits of the software.

For example, a transaction processing system may be designed to process up to 300 transactions per second; an operating system may be designed to handle up to 1,000 separate terminals. Stress testing continues these tests beyond the maximum design load of the system until the system fails.

6. This type of testing has two functions:

a). It tests the failure behaviour of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services.

Stress testing checks that overloading the system causes it to 'fail-soft' rather than collapse under its load.

b). It stresses the system and may cause defects to come to light that would not normally be discovered.

23.2 Component testing/unit testing

1. Component testing (sometimes called *unit testing*) is the process of testing individual components in the system. This is a defect testing process so its goal is to expose faults in these components

2. There are different types of component that may be tested at this stage:

a). Individual functions or methods within an object- Individual functions or methods are the simplest type of component and your tests are a set of calls to these routines with different input parameters.

b). Object classes that have several attributes and methods

- object class testing provide coverage of all the features of the object it includes:

A). The testing in isolation of all operations associated with the object

B). The setting and interrogation of all attributes associated with the object

C). The exercise of the object in all possible states. This means that all events that cause a state change in the object should be simulated.

c). Composite components made up of several different objects or functions.-These composite components have a defined interface that is used to access their functionality.

3. Inheritance makes it more difficult to design object class tests. Where a superclass provides operations that are inherited by a number of subclasses, all of these subclasses should be tested with all inherited operations. The reason for this is that the inherited operation may make assumptions about other operations and attributes, which these may have been changed when inherited. Equally, when a superclass operation is overridden then the overwriting operation must be tested. Tests that fall into the same equivalence class might be those that use the same attributes of the objects. Therefore, equivalence classes should be identified that initialise, access and update all object class attributes.

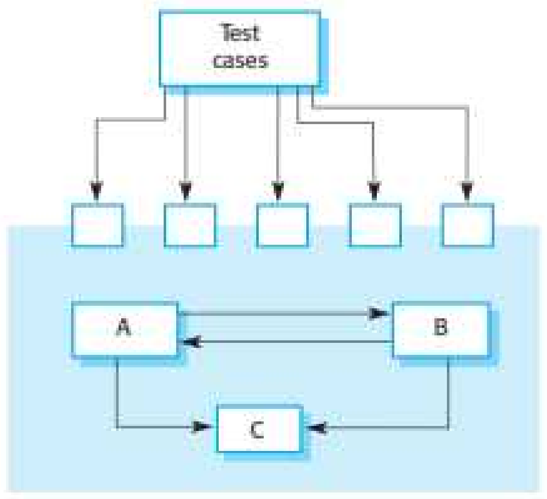
23.2.1 Interface testing

1. Many components in a system are not simple functions or objects but are composite components that are made up of several interacting objects. The functionality of these components can be accessed through their defined interface.

2. Testing these composite components then is primarily concerned with testing that, is the component interface behaves according to its specification.

3. Figure 23.7 illustrates this process of interface testing. Assume that components A, B and C have been integrated to create a larger component or sub-system. The test cases are not applied to the individual components but to the interface of the composite component created by combining these components.

Figure 23.7 Interface testing



4. Errors in the composite component may arise because of interactions between its parts.

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

a). ***Parameter interfaces*** These are interfaces where data or sometimes function references are passed from one component to another.

b). *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.

c). *Procedural interfaces* These are interfaces where one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

d). *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.

6. Interface errors are one of the most common forms of error in complex systems

These errors fall into three classes:

a). *Interface misuse* A calling component calls some other component and makes an error in the use of its interface. This type of error is particularly common with parameter interfaces where parameters may be of the wrong type, may be passed in the wrong order or the wrong number of parameters may be passed.

b). *Interface misunderstanding* A calling component misunderstands the specification of the interface of the called component and makes assumptions about the behaviour of the called component. The called component does not behave as expected and this causes unexpected behaviour in the calling component.

Example, a binary search routine may be called with an unordered array to be searched. The search would then fail.

c). *Timing errors* These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared

7. Some general guidelines for interface testing are:

a). Examine the code to be tested and explicitly list each call to an external component. Design a set of tests where the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.

b). Where pointers are passed across an interface, always test the interface with null pointer parameters.

c). Where a component is called through a procedural interface, design tests that should cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.

d). Use stress testing, as discussed in the previous section, in message-passing systems. Design tests that generate many more messages than are likely to occur in practice. Timing problems may be revealed in this way.

e). Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

interface information.

23.3 Test case design

1. Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system.

2. The goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements.

3. There are various approaches that can be taken to design test case:

a). ***Requirements-based testing*** where test cases are designed to test the system requirements. This is mostly used at the system-testing stage as system requirements are usually implemented by several components. For each requirement, identify test cases that can demonstrate that the system meets that requirement.

b). *Partition testing* where identify input and output partitions and design tests so that the system executes inputs from all partitions and generates outputs in all partitions. Partitions are groups of data that have common characteristics.

such as all negative numbers, all names less than 30 characters, all events arising from choosing items on a menu, and so on.

c). *Structural testing* where use knowledge of the program's structure to design tests that exercise all parts of the program. Essentially, when testing a program, try to execute each statement at least once. Structural testing helps identify test cases that can make this possible.

In general, when designing test cases, start with the highest-level tests from the requirements then progressively add more detailed tests using partition and structural testing.

23.3.1 Requirements-based testing

1. Requirements-based testing, is a systematic approach to test case design where consider each requirement and derive a set of tests for it.

2. Requirements-based testing is validation rather than defect testing—trying to demonstrate that the system has properly implemented its requirements.

3. Example, consider the user requirements for the LIBSYS system

a). The user shall be able to search either all of the initial set of databases or selecta subset from it.

b). The system shall provide appropriate viewers for the user to read documents in the document store.

c). Every order shall be allocated a unique identifier (ORDER_ID) that the user shall be able to copy to the account's permanent storage area.

4. Possible tests for the first of these requirements, assuming that a search function has been tested, are:

- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes one database.
- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes two databases.
- Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than two databases.
- Select one database from the set of databases and initiate user searches for items that are known to be present and known not to be present.
- Select more than one database from the set of databases and initiate searches for items that are known to be present and known not to be present.

5. from the above, testing a requirement does not mean just writing a single test. But have to write several tests to ensure the coverage of the requirement.

23.3.2 Partition testing

1. The input data and output results of a program usually fall into a number of different classes that have common characteristics such as positive numbers, negative numbers etc

- Programs normally behave in a comparable way for all members of a class. That is, test a program that does some computation and requires two positive numbers, then the program should behave in the same way for all positive numbers.

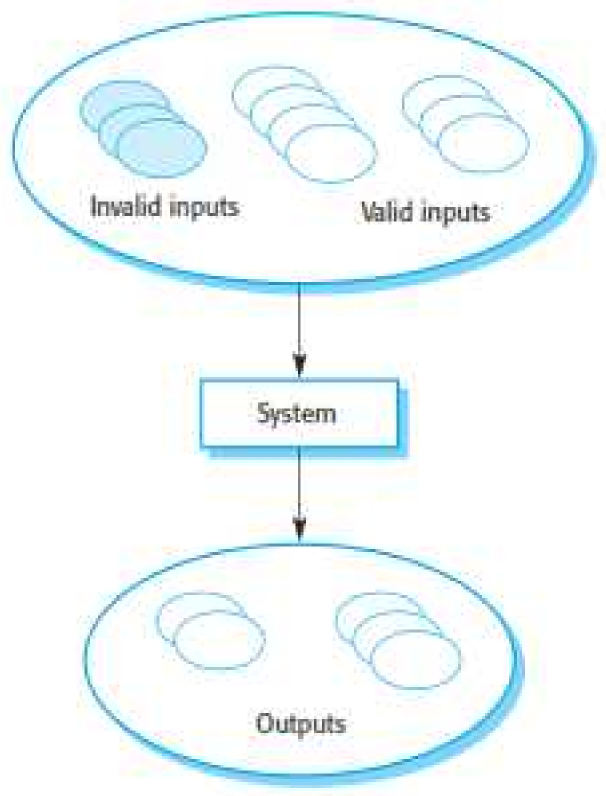
2. Because of this equivalent behaviour, these classes are sometimes called *equivalence partitions* or *domains*.

3. partition testing: It is asystematic approach to test case design, based on identifying all partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

4. In Figure 23.8, each equivalence partition is shown as an ellipse.

- Input equivalence partitions are sets of data where all of the set members should be processed in an equivalent way.
- Output equivalence partitions are program outputs that have common characteristics, so they can be considered as a distinct class.
- Identify partitions where the inputs are outside the other partitions that have been chosen.
- These test whether the program handles invalid input correctly. Valid and invalid Inputs also form equivalence partitions.
- Once a set of partitions have been identified, chose test cases from each of these partitions. choose test cases on the boundaries of the partitions plus cases close to the mid-point of the partition..
- Then test these by choosing the mid-point of the partition. Boundary values are often a typical (e.g., zero may behave differently from other nonnegative numbers) so are overlooked by developers.

Figure 23.8 Equivalence partitioning



Example (Note: If asked in exam include example otherwise no need)

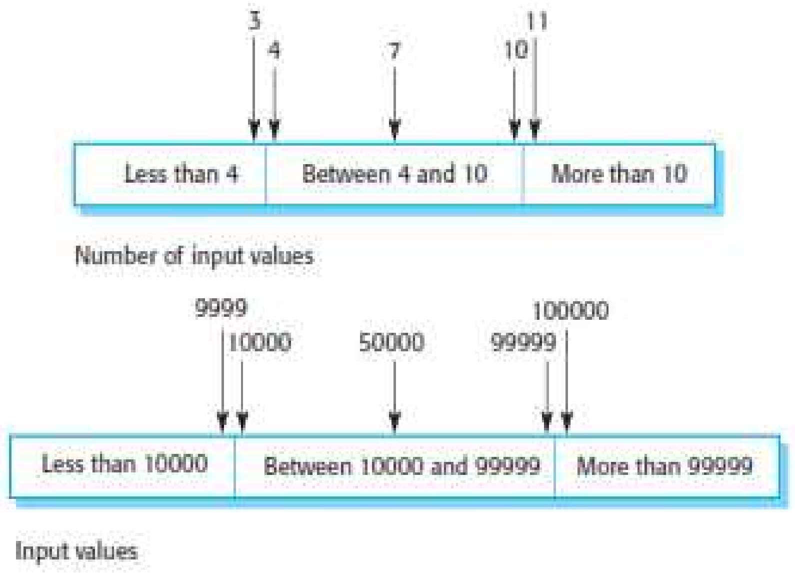
1) For example, say a program specification states that the program accepts 4 to 8 inputs that are five-digit integers greater than 10,000. Figure 23.9shows the partitions for this situation and possible test input values.
2) The pre-condition states that the search routine will only work with sequences that are not empty. The post-condition states that the variable Found is set if the key element is in the sequence. The position of the key element is the index L. The index value is undefined if the element is not in the sequence.

From this specification, see two equivalence partitions:

a. Inputs where the key element is a member of the sequence (Found = true)

b. Inputs where the key element is not a sequence member (Found = false)

Figure 23.9 Equivalence partitions



3). guidelines that are useful in designing test cases:

a). Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, the program may not work properly when presented with a single-value sequence.

b). Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.

c) Derive tests so that the first, middle and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

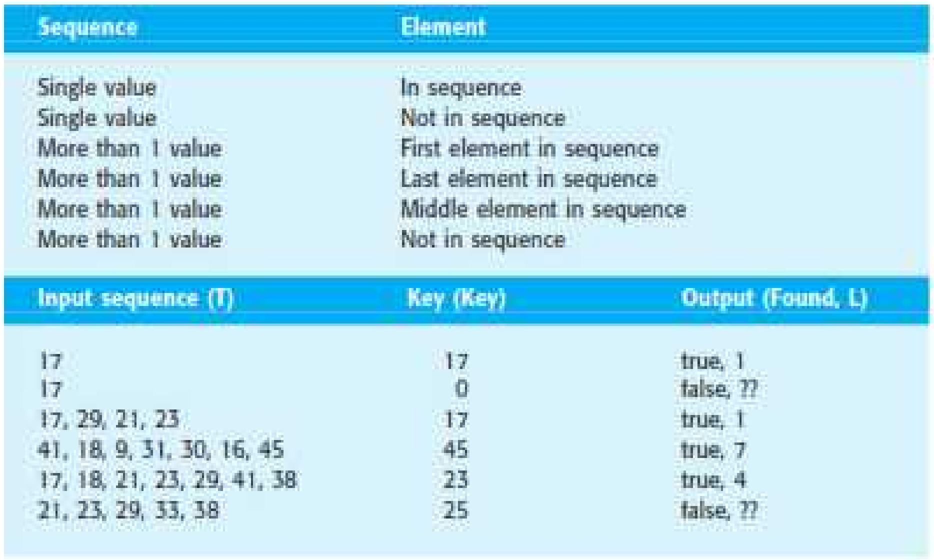
4). From these guidelines, two more equivalence partitions can be identified:

a. The input sequence has a single value.

b. The number of elements in the input sequence is greater than 1. then identify further partitions by combining these partitions

5). Figure 23.11 shows the partitions that have been identified to test the search component. A set of possible test cases based on these partitions is also shown in Figure 23.11. If the key element is not in the sequence, the value of L is undefined ('??'). The guideline that different sequences of different sizes should be used has been applied in these test cases. The set of input values used to test the search routine is not exhaustive. The routine may fail if the input sequence happens to include the elements 1, 2, 3 and 4. However, it is reasonable to assume that if the test fails to detect defects when one member of a class is processed, no other members of that class will identify defects.

Figure 23.11 Equivalence partitions for search routine



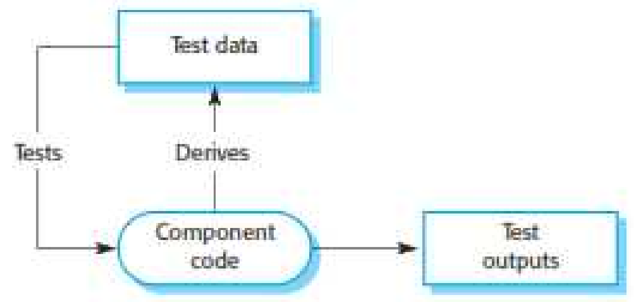
Sequence	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

23.3.3 Structural testing

1. Structural testing (Figure 23.12) is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation. This approach is sometimes called 'white-box', 'glass-box' testing, or ''clear-box' testing.

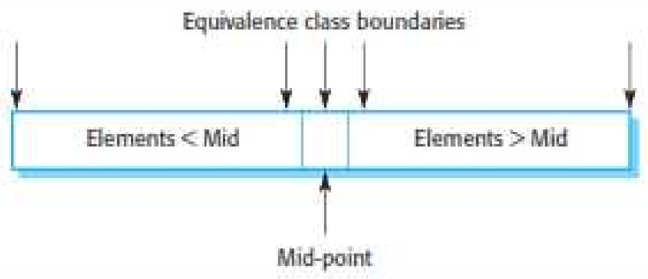
Figure 23.12 Structural testing



Example

1. Identifying partitions and test cases for a binary search routine (Figure 23.14).
2. pre-conditions: The sequence is implemented as an array that array must be ordered and the value of the lower bound of the array must be less than the value of the upper bound.
3. Binary searching involves splitting the search space into three parts. Each of these parts makes up an equivalence partition (Figure 23.13). then design test cases where the key lies at the boundaries of each of these partitions

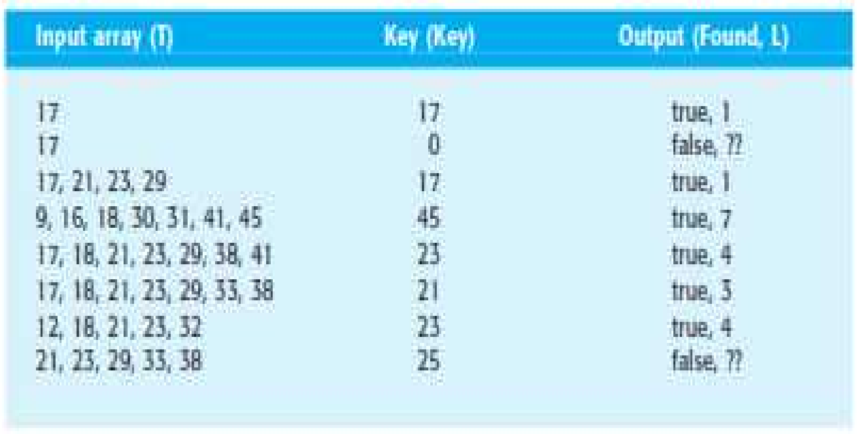
Figure 23.13 Binary search equivalence classes



.

4. This leads to a test cases for the search routine, as shown in Figure 23.15.

Figure 23.15 Test cases for search routine



Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

23.3.4 Path testing

1. Path testing is a structural testing strategy whose objective is to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. Furthermore, all conditional statements are tested for both true and false cases.

2. The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program.

-A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams.

Example :flow graph for the binary search method in Figure 23.16. shown each statement as a separate node where the node number corresponds to the line number in the program.

3. The objective of path testing is to ensure that each independent path through the program is executed at least once. An independent program path is one that traverses at least one new edge in the flow graph. Both the true and false branches of all conditions must be executed.

4. The number of independent paths in a program can be found by computing the *cyclomatic complexity* of the program flow graph. For programs without goto statements, the value of the cyclomatic complexity is one more than the number of conditions in the program. A simple condition is logical expression without ‘and’ or ‘or’ connectors.

5. If the program includes compound conditions, which are logical expressions including ‘and’ or ‘or’ connectors, then count the number of simple conditions in the compound conditions when calculating the cyclomatic complexity.

Example : if there are six if-statements and a while loop and all conditional expressions are simple, the cyclomatic complexity is 8. If one conditional expression is a compound expression such as ‘if A and B or C’, then count this as three simple conditions. The cyclomatic complexity is therefore 10. The cyclomatic complexity of the binary search algorithm (Figure 23.14) is 4 because there are three simple conditions at lines 5, 7 and 11.

6. After discovering the number of independent paths through the code by computing the cyclomatic complexity, next design test cases to execute each of these paths.

7. The minimum number of test cases that you need to test all program paths is equal to the cyclomatic complexity.

8. dynamic program analyser can be used to discover the program’s execution profile. Dynamic program analysers are testing tools , count the number of times each program statement has been executed.

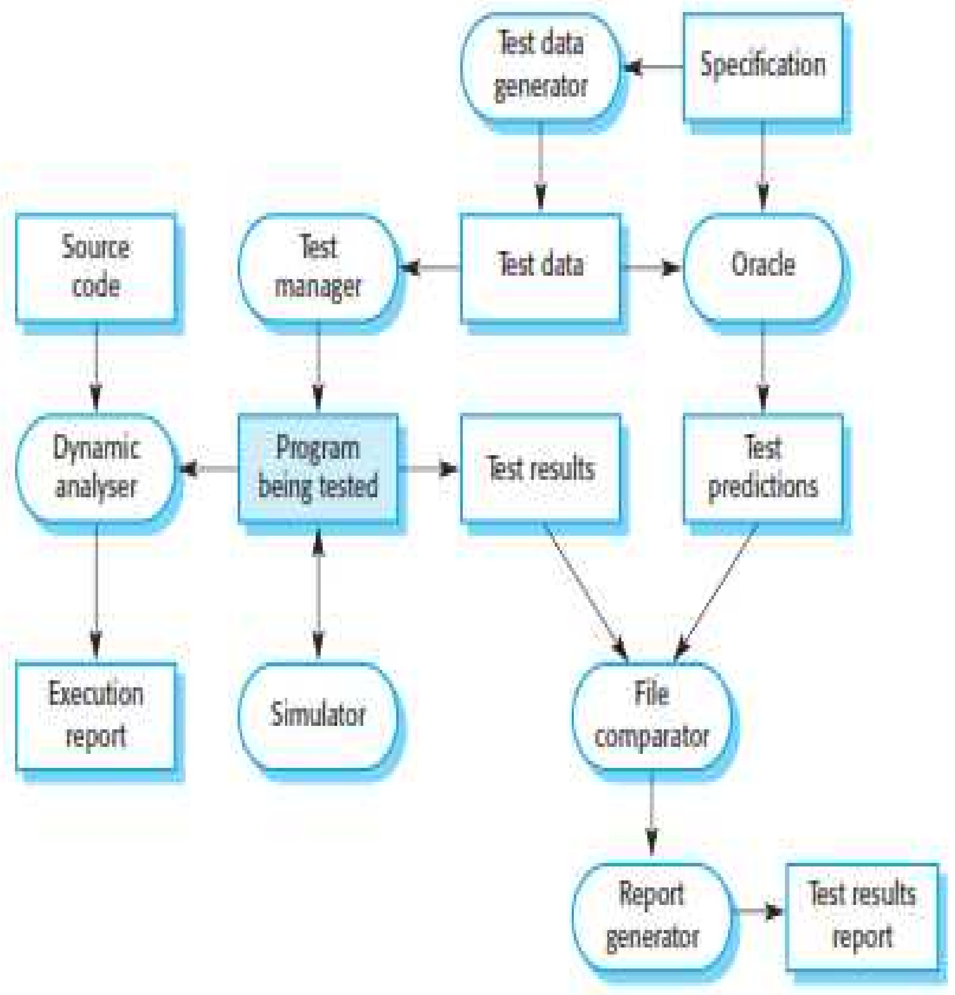
23.4 Test automation

1. Testing is an expensive and laborious phase of the software process. As a result, software tools have been developed to perform testing. These tools offer a range of facilities and their use can significantly reduce the costs of testing.

2. A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.

3. Figure 23.17 shows some of the tools that might be included in a testing workbench:

Figure 23.17 A testing workbench



a). *Test manager* Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested.

b). *Test data generator* Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.

c). *Oracle* Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems.

d). *File comparator* Compares the results of program tests with previous test results and reports differences between them.

e). *Report generator* Provides report definition and generation facilities for test results.

f). *Dynamic analyser* Adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.

g). *Simulator* Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulators are script-driven programs that simulate multiple simultaneous user interactions.