Unit 7

Verification and validation

Chapter 22

Verification and validation

Contents

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.
- Has two principal objectives
- 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

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. For example, the level of confidence required for software that is used to control a safety-critical system is very much higher than that required for a prototype software system that has been developed to demonstrate some new ideas.

2. User expectations It is a sad reflection on the software industry that many users have low expectations of their software and are not surprised when it fails during use. They are willing to accept these 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 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.

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. You can use inspections at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyses are static V & V techniques, as you don't need to run the software on a computer.
2. ***Software testing*** involves running an implementation of the software with test data. You examine the outputs of the software and its operational behaviour to check that it is performing as required. Testing is a dynamic technique of verification and validation.

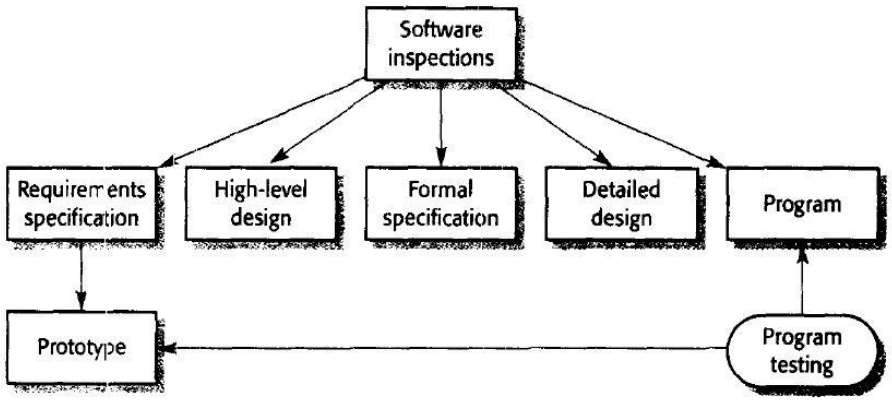


Figure 22.1 Static and dynamic verification and validation

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. Therefore, you can use software inspections at all stages of the software process. Starting with the requirements, any readable representations of the software can be inspected. As I have discussed, requirements and design reviews are the main techniques used for error detection in the specification and design.

Program testing:
Testing involves exercising the program using data like the real data processed by the program. You discover program defects or inadequacies by examining the outputs of the program and looking for anomalies.

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. As part of validation testing, you may use statistical testing to test the program s performance and reliability, and to check how it works under operational conditions.

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.

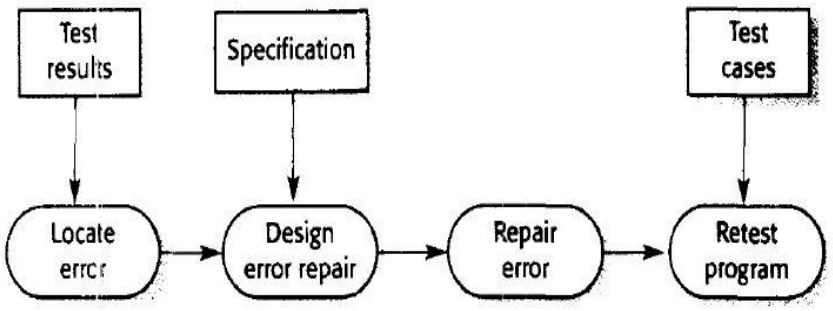
Testing and debugging

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. However, testing (or, more generally verification and validation) and debugging has different goals:

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.

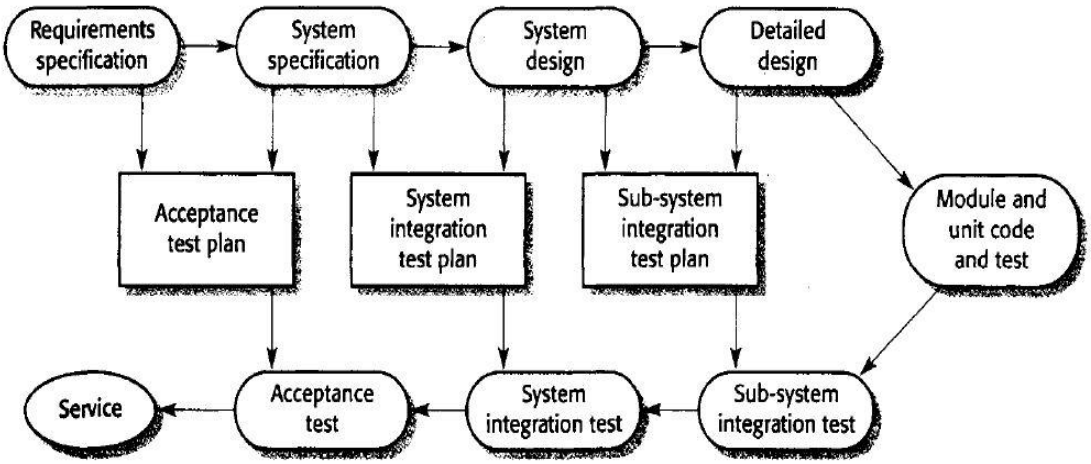
Verification and validation is concerned with establishing the existence of defects in a program. Debugging is concerned with locating and repairing these errors. Debugging involves formulating a hypothesis about program behaviour then testing these hypotheses to find the system error.

Figure 22.2 The debugging process



22. 1 Planning verification and validation

- Verification and validation is an expensive process. For some systems, such as real-time systems with complex non-functional constraints, more than half the system development budget may be spent on V & V.
- Careful planning is needed to get the most out of inspections and testing and to control the costs of the verification and validation process.
- You should start planning system validation and verification early in the development process. The software development process model shown in Figure 22.3 is sometimes called the V-model (turn Figure 22.3 on end to see the V).
- It is an instantiation of the generic waterfall model (see Chapter 4) and 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.
- As part of the V & V planning process, you should decide on the balance between static and dynamic approaches to verification and validation.
- 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 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.

The structure of a software test plan

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

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.

22.2 Software inspections

Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies. Generally, inspections focus on source code. But any readable representation of the software such as its requirements or a design model can be inspected.

There are three major advantages of inspection over testing:
1. 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.
2. 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.
3. 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.

Inspection success

Many different defects may be discovered in a single inspection. In testing, one defect, may mask another so several executions are required. The reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise.

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.

22.2.1 Program inspections

- 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 (Fagan 1976; Fagan, 1986). It is now a fairly widely used method of program verification, especially in critical systems engineering.
- The specific goal of inspections is to find program defects rather than to consider broader design issues. Defects may be logical errors, anomalies in the code that might indicate an erroneous condition or noncompliance with organisational or project standards.
- 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. 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 and the moderator organises the process.

Roles in inspection

Grady and Van Slack (Grady and Van Slack, 1994) suggest six roles, as shown in Figure 22.5.

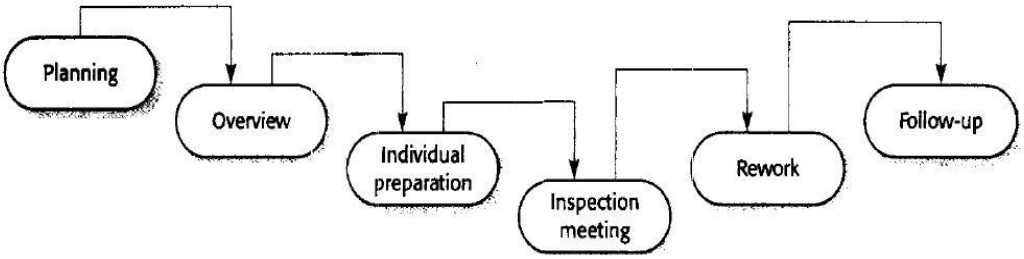
Figure 22.5 Roles in the inspection process

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.

Inspection pre- conditions

- 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.

The inspection process



- 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.
- 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.
- This is followed by a period of individual preparation. Each inspection team member studies the specification and the program and looks for defects in the code.
- The inspection itself should be fairly short (no more than two hours) and should focus on defect detection, standards conformance and poor-quality programming. The inspection team should not suggest how these defects should be corrected nor should it recommend changes to other components.
- Following the inspection, the program s author should make changes to it to correct the identified problems.
- 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.

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.

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?

Inspection rate

- About 500 source code statements per hour can be presented during the overview stage.
- During individual preparation, about 125 source code statements per hour can be examined.
- From 90 to 125 statements per hour can be inspected during the inspection meeting.

22.3 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.

Automated static analysis checks

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

Stages of static analysis

- *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.
- *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.
- *Interface analysis* This analysis checks the consistency of routine and procedure declarations and their use. It is unnecessary if a strongly typed language such as Java is used for implementation as the compiler carries out these checks. Interface analysis can detect type errors in weakly typed languages like FORTRAN and C. Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.
- *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. Information flow analysis can also show the conditions that affect a variable's value.

- *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.

LINT static analysis

Unix and Linux systems include a static analyser called LINT for C programs. LINT provides static checking, which is equivalent to that provided by the compiler in a strongly typed language such as Java. An example of the output produced by LINT is shown in below. In this transcript of a Unix terminal session, commands are shown in italics. The first command (line 138) lists the (nonsensical) program. It defines a function with one parameter, called printarray, and then calls this function with three parameters. Variables i and c are declared but are never assigned values. The value returned by the function IS never used.

The line numbered 139 shows the C compilation of this program with no errors reported by the C compiler. This is followed by a call of the LINT static analyser, which detects and reports program errors. The staatic analyser shows that the variables;: and i have been used but not initialised. And that printarray has been called with a difterent number of arguments than are ceclared. It also identifies the inconsistent use of the first argument in printarray and the fact that the functIOn value is never used.

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray) int
Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}
139% cc lint_ex.c
140% lint lint_ex.c
lint_ex.c(10): warning: c may be used before set lint_ex.c(10):
warning: i may be used before set printarray: variable # of
args. lint_ex.c(4) :: lint_ex.c(10) printarray, arg. 1 used
inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) ::
LINT static analysis
lint_ex.c(11)
printf returns value which is always ignored
```

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.

22.4 Verification and formal methods

- Formal methods can be used when a mathematical specification of the system is produced.
- They are the ultimate static verification technique.
- They involve detailed mathematical analysis of the specification and may develop formal arguments that a program conforms to its mathematical specification.

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

- A formal specification of the system may be developed and mathematically analysed for inconsistency. This technique is effective in discovering specification errors and omissions.
- You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. 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.

Arguments for formal methods

- Producing a mathematical specification requires a detailed analysis of the requirements and this is likely to uncover errors.
- They can detect implementation errors before testing when the program is analyzed alongside the specification.

Arguments against formal methods

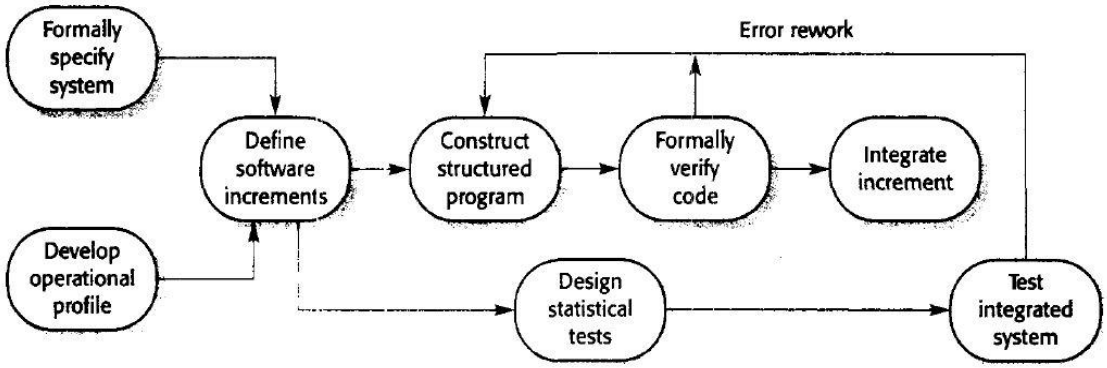
- Require specialized notations that cannot be understood by domain experts.
- Very expensive to develop a specification and even more expensive to show that a program meets that specification.
- It may be possible to reach the same level of confidence in a program more cheaply using other V & V techniques.

22.4.1 Cleanroom software development

A model of the Cleanroom process is shown in Figure 22.10. The objective of this approach to software development is zero-defect software. The name 'Cleanroom was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere. Cleanroom development is particularly relevant to this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.

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

1. *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.
2. *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.



3. *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. A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.
4. *Static verification* The developed software is statically verified using rigorous software inspections. There is no unit or module testing process for code components.
5. *Statistical testing of the system* The integrated software increment is tested statistically, as discussed in Chapter 24, Ito determine its reliability. These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown m Figure 22.10.

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

1. The specification team This group is responsible for developing and maintaining the system specification. This team produces customer-oriented specifications and mathematical specifications for verification. In some cases, when the specification is complete, the specification team also takes responsibility for development.

2. The development team This tearn has the responsibility of developing and verifying the software. The software is not executed during the development process. A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.

3. 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. Test case development is carried out in parallel with software development. The test cases are used to certify the software reliability.

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.

Chapter 23
Software Testing

Contents

The two fundamental testing activities are **component testing**-testing the parts of the system-and **system testing**-testing the system as a whole.

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, such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

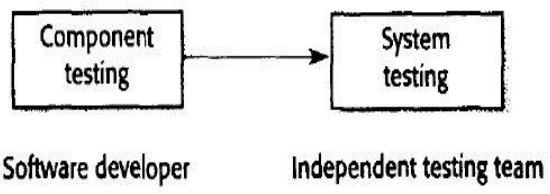
Component testing

- Testing of individual components.
- Usually the responsibility of the component developer.
- Tests are derived from developer's experience.

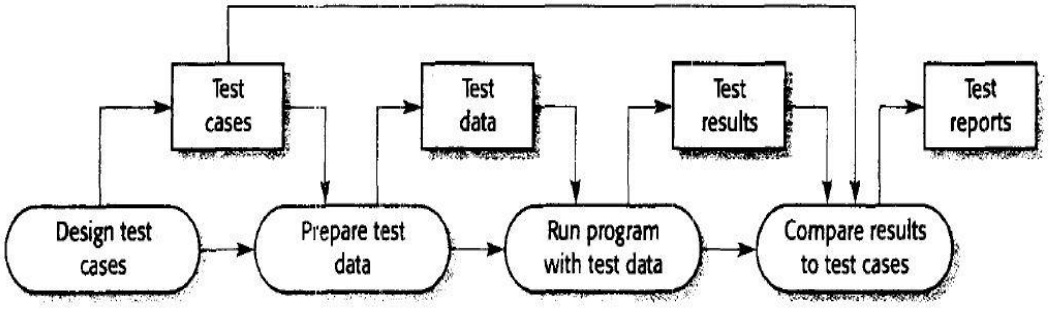
System testing

- Testing of groups of components integrated to create system or sub-system.
- The responsibility of independent testing team.
- Tests are based on system specification.

Figure 23.1 Testing phases



Software testing process



Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. Overall, therefore, the goal of software testing is to convince system developers and customers that the software is good enough for operational use.

Testing is a process intended to build confidence in the software. A general model of the testing process is shown in Figure. 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. The output of the tests can only be predicted by people who understand what the system should do.

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.

23.1 System testing

System testing involves integrating two or more components that implement system functions or features and then testing this integrated system. In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer; in a waterfall process, system testing is concerned with testing the entire system.

There are two distinct phases to system testing:

1. ***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.
2. ***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 and with ensuring that the system is dependable. 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.

23.1.1 Integration testing

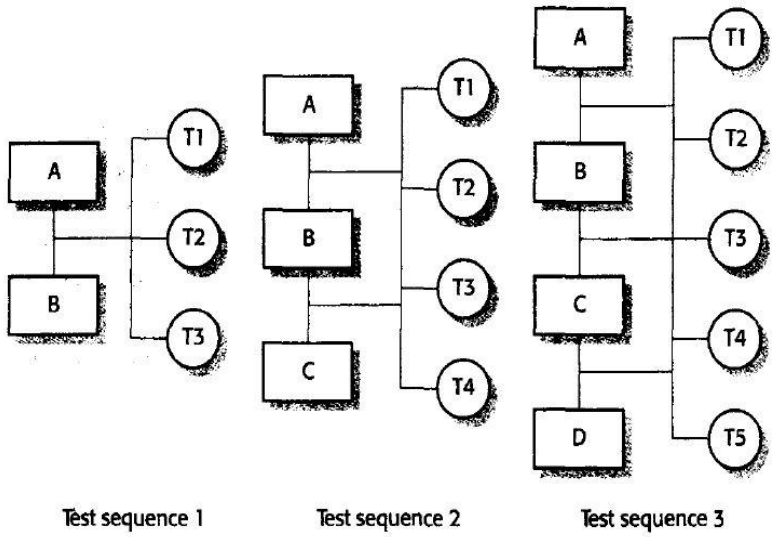
The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system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

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

1. **Top-down integration**
   Sometimes, the overall skeleton of the system is developed first, and components are added to it. This is called top-down integration.
2. **Bottom-up integration** Alternatively, first integrate infrastructure components that provide common services, such as network and database access, then add the functional components.

A major problem that arises during integration testing is localising errors. There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred. To make it easier to locate errors, you should always use an incremental approach to system integration and testing.

Figure 23.3
Incremental
integration testing



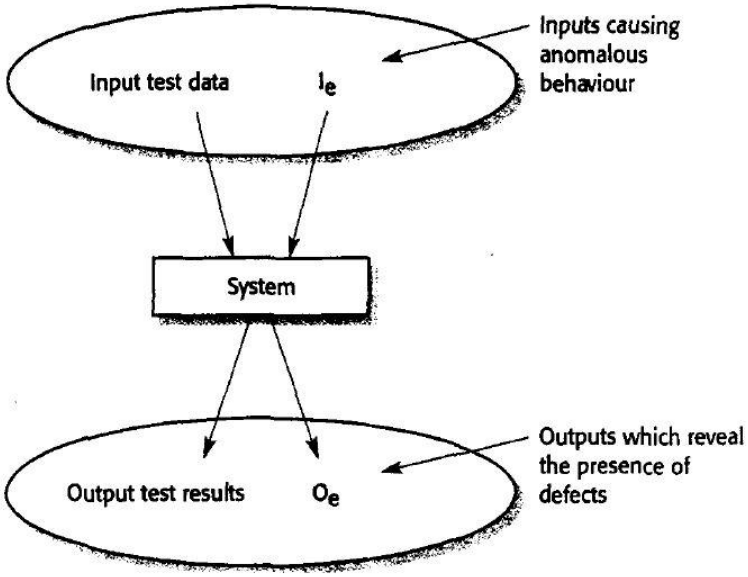
In the example shown in Figure, A, B, C and D are components and n to T5 are related sets of tests of the features incorporated in the system. n, T2 and T3 are first run on a system composed of component A and component B (the minimal system). If these reveal defects, they are corrected. Component C is integrated and n, 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. The source of the problem is localised, thus simplifying defect location and repair. Test set T4 is also run on the system. Finally, component D is integrated and tested using existing and new tests (T5).

23.1.2 Release testing

- Release testing is the process of testing a release of the system that will be distributed to customers.
- The primary goal of this process is to increase the supplier's confidence that the system meets its requirements. If so, it can be released as a product or delivered to the customer.
- To demonstrate that the system meets its requirements, you have to show that it delivers the specified functionality, performance and dependability, and that it does not fail during normal use.

Release testing is usually a black-box testing process where the tests are derived from the system specification. The system is treated as a black box whose behaviour can only be determined by studying its inputs and the related outputs. Another name for this is *functional testing* because the tester is only concerned with the functionality and not the implementation of the software. Figure 23.4 illustrates the model of a system that is assumed in black-box testing. 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 0,) then the test has detected a problem with the software

Figure 23.4 Black box testing



Testing guidelines are:

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

LIBSYS Testing:

A student in Scotland studying American history has been asked to write a paper on 'Frontier mentality in the American West from 1840 to 1880'. 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 whether she can access 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 library's server and printed for her. She receives a message from LIBS YS telling her that she will receive an e-mail message when the 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:

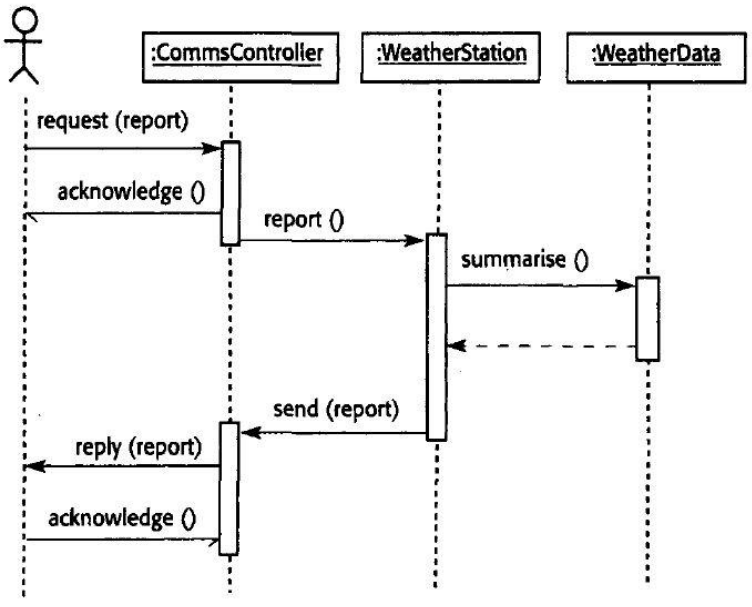
1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility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

Use case

Use cases can be a basis for deriving the tests for a system. They help identify operations to be tested and help design the required test cases. From an associated sequence diagram, the inputs and outputs to be created for the tests can be identified.

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. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the test. Therefore issuing a request for a report will result in the execution of the following thread of methods:
Com msController:request --> WeatherStation:report --> WeatherData:summarise

Figure 23.5 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:

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

23.1.3 Performance testing

- Once a system has been completely integrated, it is possible to test the system for emergent properties such as performance and reliability Performance tests have to be designed to ensure that the system can process its intended load.
- As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system.
- To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A, 5% of type B and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A.

Stress testing

In performance testing, this means stressing the system 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.

This type of testing has two functions:

- 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.
- 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.

23.2 Compon1ent testing

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. As I discussed in the introduction, for most systems, the developers of components are responsible for component testing.

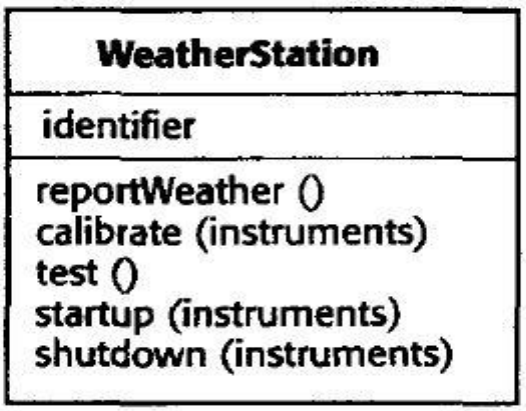
There are different types of component that may be tested at this stage:

- Individual functions or methods within an object
- Object classes that have several attributes and methods
- Composite components made up of several different objects or functions. These composite components have a defined interface that is used to access their functionality.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. Therefore, object class testing should include:

- The testing in isolation of all operations associated with the object
- The setting and interrogation of all attributes associated with the object
- 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.

Weather station object interface



Consider, for example, the weather station from Chapter 14 whose interface is shown in above Figure. It has only a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks whether it has been set up. You need to define test cases for reportWeather, calibrate, test, startup and shutdown. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test shutdown you need to have executed the startup method.

Weather station testing

Examples of state sequences that should be tested in the weather station include:

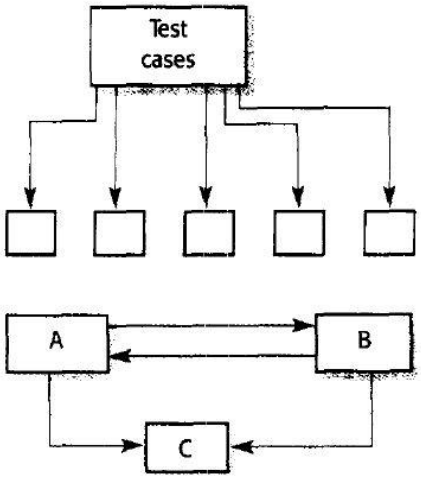
Shutdown -> Waiting -> Shutdown

Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Waiting -> Collecting -> Waiting -> Summarising --> Transmitting -> Waiting

23.2.1 Interface Testing

Figure 23.7 Interface testing



Many components in a system are not simple functions or objects but are composite components that are made up of several interacting objects. Testing these composite components then is primarily concerned with testing that the component interface behaves according to its specification.

Figure 23.7 illustrates this process of interface testing. Assume that components A, Band 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 comp051te component created by combining these components. Interface testing is particularly important for object-oriented and component-based development. Objects and components are defined by their interfaces and may be reused in combination with other components in different systems. Interface errors in the composite component cannot be detected by testing the individual objects or components. Errors in the composite component may arise because of interactions between its parts.

Interface Types:

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

2. ***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.

3. ***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.

4. ***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 fall into three classes:

1. ***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.

2. ***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. For example, a binary search routine may be called with an unordered array to be searched. The search would then fail.

3. ***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 interface information.

General guidelines for interface testing are:

1. 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.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.
3. 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.
4. Use stress testing, as discussed in the previous section, in message-passing system Design tests that generate many more messages than are likely to occur in practice. Timing problems may be revealed in this way.
5. 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.

23.3 Test case design

- 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.
- 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.

Approaches to test case design:

1. 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, you identify test cases that can demonstrate that the system meets that requirement.

2. Partition testing where you 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.

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

23.3.1 Requirements-based testing

Requirements-based testing, therefore, is a systematic approach to test case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing.

For example, consider the requirements for the LIBSYS system

1. The user shall be able to search either all of the initial set of databases or select a subset from it.
2. The system shall provide appropriate viewers for the user to read documents in the document store.
3. 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.

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.

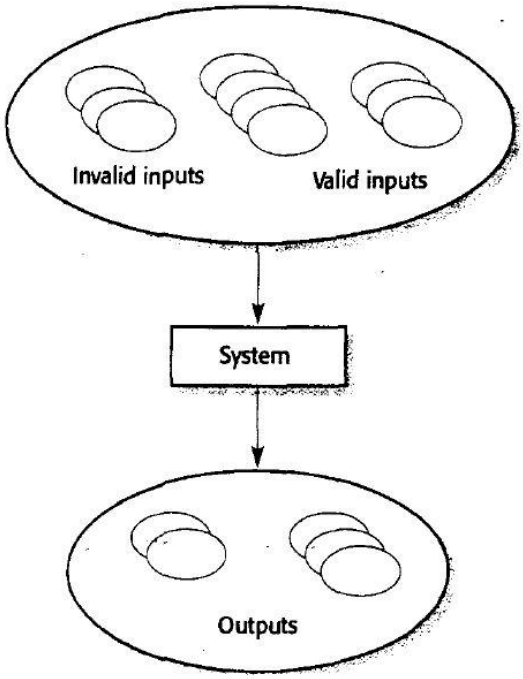
23.3.2 Partition testing

- 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 and menu selections.
- Because of this equivalent behaviour, these classes are sometimes called *equivalence partitions* or *domains* (Bezier, 1990). One systematic approach to test case design is 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.

In Figure, 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. You also identify partitions where the inputs are outside the other partitions that you have chosen.

These test whether the program handles invalid input correctly. Valid and invalid inputs also form equivalence partitions. Once you have identified a set of partitions, you can choose test cases from each of these partitions.

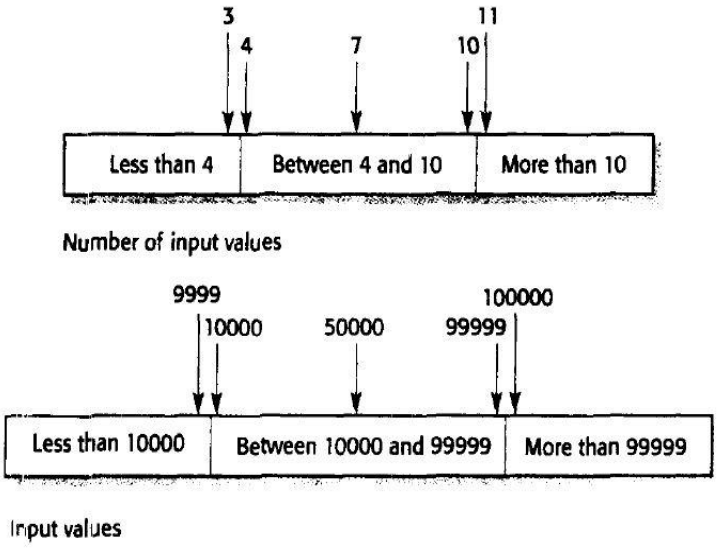
Figure 23.8 Equivalence partitioning



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 23.9 shows the partitions for this situation and possible test input values.

To illustrate the derivation of test cases, I use the specification of a search component, shown in Figure 23.10. This component searches a sequence of elements for a given element (the key). It returns the position of that element in the sequence. I have specified this in an abstract way by defining pre-conditions, which are true before the component is called, and post-conditions, which are true after execution.

Figure 23.9 Equivalence partitions



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, you can see **two equivalence partitions**:

1. Inputs where the key element is a member of the sequence (Found =true)
2. Inputs where the key element is not a sequence member (Found =false)

When you are testing programs with sequences, arrays or lists, there are a number of guidelines that are often useful in designing test cases:

Figure 23.10 The specification of a search routine

```
procedure Search (Key : ELEM ; T: SEQ of ELEM ;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
    – the sequence has at least one element
    T'FIRST <= T'LAST
Post-condition
    – the element is found and is referenced by L
    ( Found and T (L) = Key)
or
    – the element is not in the sequence
    ( not Found and
    not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key ))
```

Figure 23.11 Equivalence partitions for search routine

Sequence	Element
Single value	In sequ nce
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, ??

Testing guidelines:

1. 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.
2. 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.
3. Derive tests so that the first, middle and last elements of the sequence are accessed.

This approach reveals problems at partition boundaries.

From these guidelines, **two more equivalence partitions** can be identified:

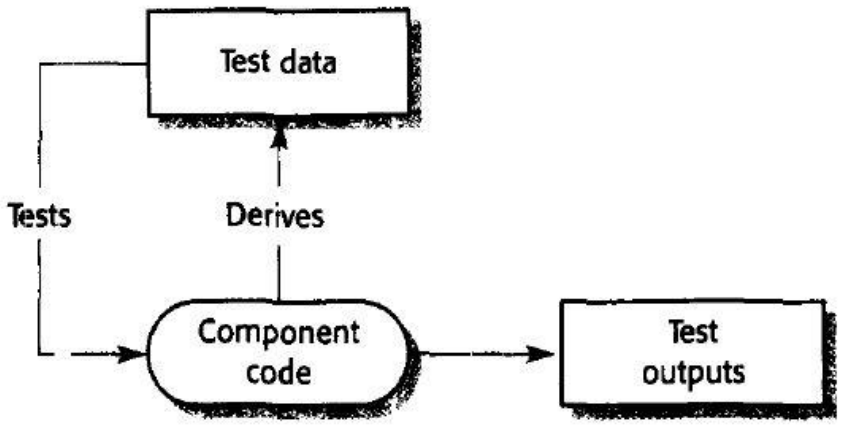
1. The input sequence has a single value.
2. The number of elements in the input sequence is greater than 1.

You then identify further partitions by combining these partitions-for example, the partition where the number of elements in the sequence is greater than 1 and the element is not in the sequence.

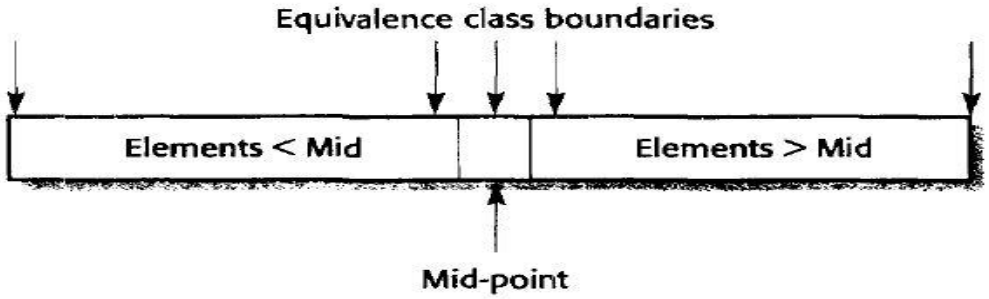
A set of possible test cases based on these partitions is also shown in Figure 23.11. If the key element is not in the sequence, the value of L is undefined *('??').* The guideline that different sequences of different sizes should be used has been applied in these test cases.

23.3.3 Structural testing

Structural testing (Figure below) 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 to distinguish it from black-box testing.



By examining the code of the search routine, you can see that binary searching involves splitting the search space into three parts. Each of these parts makes up an equivalence partition (Figure below). You then design test cases where the key lies at the boundaries of each of these partitions.




```
        public static void search ( int key, int [] elemArray, Result r )
        {
1.          int bottom = 0 ;
2.          int top = elemArray.length - 1 ;
            int mid ;
3.          r.found = false ;
4.          r.index = -1 ;
5.          while ( bottom <= top )
            {
6               mid = (top + bottom) / 2 ;
7               if (elemArray [mid] == key)
                {
8                   r.index = mid ;
9                   r.found = true ;
10                   return ;
                } // if part
                else
                {
11                  if (elemArray [mid] < key)
12                      bottom = mid + 1 ;
                    else
13                      top = mid - 1 ;
                }
            } //while loop
14.     } // search
    } //BinSearch
```

This leads to a revised set of test cases for the search routine, as shown in Figure 23.15. Notice that I have modified the input array so that it is arranged in ascending order and have added further tests where the key element is adjacent to the midpoint of the array.

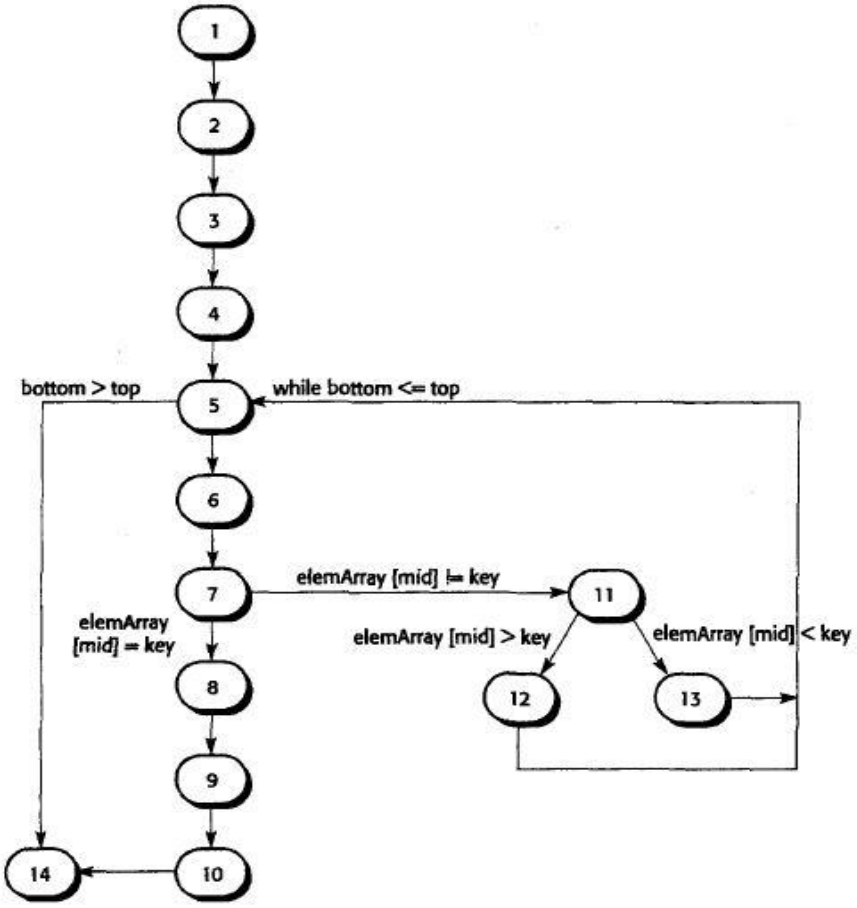
Figure 23.15 Test cases for search routine

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

- 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. In an object-oriented development process, path testing may be used when testing methods associated with objects.
- 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. If there are no goto statements in a program, it is a simple process to derive its flow graph.
- The objective of path testing is to ensure that each independent path through the program is executed at least once.

Figure 23.16 Flow graph for a binary search routine



The flow graph for the binary search procedure is shown in Figure 23.16 where each node represents a line in the program with an executable statement. By tracing the flow, therefore, you can see that the paths through the binary search flow graph are:

1, 2, 3,4,5,6,7,8,9, 10, 14

1, 2, 3, 4, 5, 14

1,2,3,4,5,6,7, 11, 12, 5,....

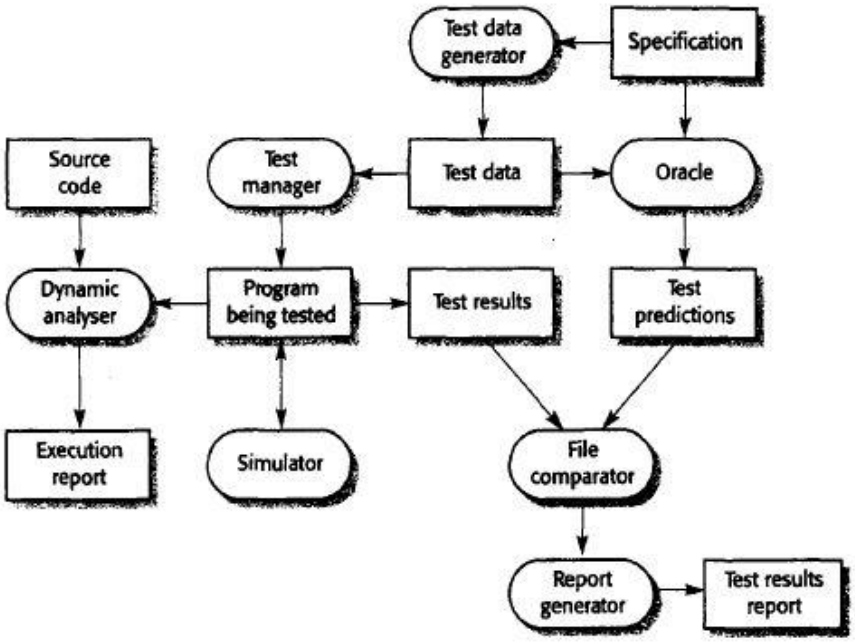
1,2,3,4,6,7,2,11,13,5, ...

- If all of these paths are executed, we can be sure that every statement in the method has been executed at least once and that every branch has been exercised for true and false conditions.
- We can find the number of independent paths in a program by computing the *cyclomatic complexity* (McCabe, 1976) 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.
- 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.

23.4 Test automation

Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. One approach to test automation (Mosley and Posey,2002) where a testing framework such as JUnit (Massol and Husted, 2003) is used for regression testing. JUnit is a set of Java classes that the user extends to create an automated testing environment. Each individual test is implemented as an object and a test runner runs all of the tests.

Software testing workbench:



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

1. *Test manager* manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.

2. 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.

3. Oracle Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-ta-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.

4. File comparator Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.

5. Report generator Provides report definition and generation facilities for test results.

6. 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.

7. Simulator Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute. User interface simulations are script-driven programs that simulate multiple simultaneous user interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.