Chapter - 11

Software Testing Techniques and Strategies

11.1 Introduction

Software has infiltrated almost all areas in the industry and has over the years become more and more wide spread as a crucial component of many systems. System failure in any industry can be very costly and in the case of critical systems (fight control, nuclear reactor monitoring, medical applications, etc.) it can mean lost human lives. These "cost" factors call for some kind of system failure prevention. One way to ensure system's reliability is to extensively test the system. Since software is a system component, it requires a testing process also.

Software testing is the process of executing a program with the intention of finding errors in the code. It is the process of exercising or evaluating a system or system component by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results (Shooman 1983; Conte 1986; Sommerville 1998; Pressman 1997).

The objective of testing is to show incorrectness and testing is considered to succeed when an error is detected (Myers 1979). An error is a conceptual mistake made by either the programmer or the designer or a discrepancy between a computed value and a theoretically correct value. A fault is a specific manifestation of an error. An error may be the cause of several faults. A failure is the inability of a system or component to perform its required function within the specified limits. A failure may be produced when a fault is executed or exercised.

Testing should not be a distinct phase in system development but should be applicable throughout the design, development and maintenance phases.

Researchers are also making efforts in the direction of generating some good automated software testing tools and have also been successful to some extent.

Following sections in this chapter presents all the related concepts about software testing.

11.2 Software Testing

There are many published definitions of software testing, however; all of these definitions boil down to essentially the same thing:

*"Software testing is the process of executing software in a controlled manner, in order to answer the question: **Does the software behave as specified?**"*

Software testing is often used in association with the terms verification and validation (Sommerville 1998; Pressman 1997).

- *Verification* is the checking or testing of items, including software, for conformance and consistency with an associated specification. Software testing is just one kind of verification, which also uses techniques such as reviews, analysis, inspections and walkthroughs.
- *Validation* is the process of checking that what has been specified is what the user actually wanted.

> ***Validation:-Are we doing the right job?***
> ***Verification:-Are we doing the job right?***

The term bug is often used to refer to a problem or fault in a computer. There are software bugs and hardware bugs. The term originated in the United States, at the time when computers were built out of valves, when a series of previously inexplicable faults were eventually traced to moths flying about inside the computer.

Software testing should not be confused with debugging.

Debugging is the process of analysing and locating bugs when software does not behave as expected. Although the identification of some bugs will be obvious from playing with the software, a methodical approach to software testing is a much more thorough means of identifying bugs. Debugging is therefore an activity that supports testing, but cannot replace testing. However, no amount of testing can be guaranteed to discover all bugs.

Other activities that are often associated with software testing are static analysis and dynamic analysis. Static analysis investigates the source code of software, looking for problems and gathering metrics without actually executing the code. Dynamic analysis looks at the behaviour of software while it is executing, to provide information such as execution traces, timing profiles, and test coverage information.

11.3 Testing and the Software Lifecycle

Testing should be thought of as an integral part of the software process and an activity that must be carried out throughout the life cycle.

Each phase in the software lifecycle has a distinct end product such as the software requirements specification (SRS) documentation, program unit design and program unit code. Each end product can be checked for conformance with a previous phase and against the original requirements. Thus, errors can be detected at each phase of development.

Validation and Verification should occur throughout the software lifecycle.

- *Verification* is the process of evaluating each phase end product to ensure consistency with the end product of the previous phase.
- *Validation* is the process of testing software, or a specification, to ensure that it matches user requirements.

Software testing is that part of validation and verification associated with evaluating and analysing program code. It is one of the two most expensive stages within the software lifecycle, the other being maintenance. Software testing of a product begins after the development of the program units and continues until the product is obsolete.

Testing and fixing can be done at any stage in the life cycle. However, the cost of finding and fixing errors increases dramatically as development progresses.

Changing a Requirements document during the first review is inexpensive. It costs more when requirements change after the code has been written: the code must be rewritten.

Bug fixes are much cheaper when programmers find their own errors. Fixing an error before releasing a program is much cheaper than sending new disks, or even a technician to each customer's site to fix it later. It is illustrated in **Fig. 11.1.**

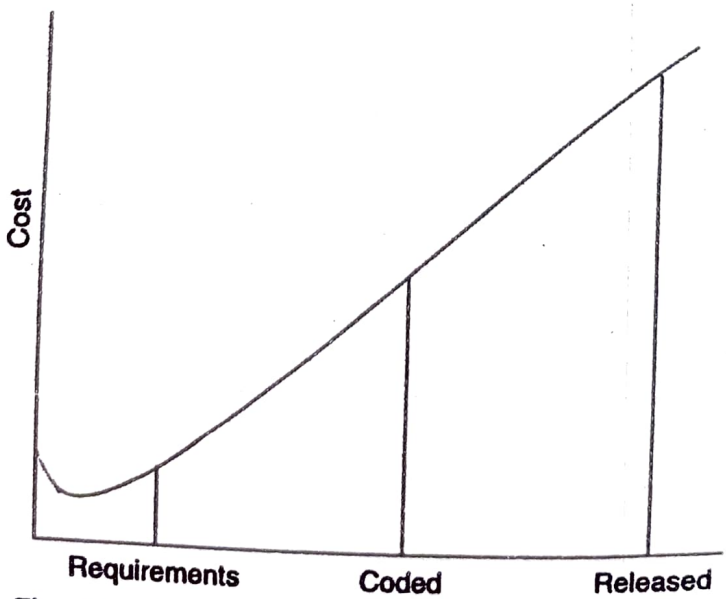

Fig. 11.1: *Cost of Finding and Fixing Software Errors*

The types of testing required during several phases of software lifecycle are described below:

Requirements

- Requirements must be reviewed with the client; rapid prototyping can refine requirements and accommodate changing requirements.

Specification

- The specifications document must be checked for *feasibility, traceability, completeness,* and absence of *contradictions* and *ambiguities.*
- Specification reviews (walkthroughs or inspections) are especially effective.

Design

- Design reviews are similar to specification reviews, but more technical.
- The design must be checked for *logic faults, interface* faults, lack of *exception handling,* and *non-conformance* to specifications.

Implementation

- Code modules are informally tested by the programmer while they are being implemented (*desk checking*).
- Thereafter, formal testing of modules is done methodically by a testing team.
- This formal testing can include *non-execution-based methods* (code inspections and walkthroughs) and *execution-based methods* (black-box testing, white-box testing).

Integration

- Integration testing is performed to ensure that the modules combine together correctly to achieve a product that meets its specifications. Particular care must be given to the interfaces between modules.
- The appropriate order of combination must be determined as *top-down, bottom-up,* or a combination thereof.

Product Testing

- The functionality of the product as a whole is checked against its specifications. Test cases are derived directly from the specifications document. The product is also tested for robustness (error-handling capabilities and stress tests).
- All source code and documentation are checked for completeness and consistency.

Acceptance Testing

- The software is delivered to the client, who tests the software on the actual hardware, using actual data instead of test data. A product cannot be considered to satisfy its specifications until it has passed an acceptance test.
- Commercial off-the-shelf (or *shrink-wrapped*) software usually undergoes *alpha* and *beta* testing as a form of acceptance test.

Maintenance

- Modified versions of the original product must be tested to ensure that changes have been correctly implemented.
- Also, the product must be tested against previous test cases to ensure that no inadvertent changes have been introduced. This latter consideration is termed *regression* testing.

Software Process Management

- The software process management plan must undergo scrutiny. It is especially important that cost and duration estimates be checked thoroughly.

If left unchecked, errors can propagate through the development lifecycle and amplify in number and cost. The cost of detecting and fixing an error is well documented and is known to be more costly as the system develops. An error found during the operation phase is the most costly to fix.

11.4 Objectives of Software Testing

Software Testing is usually performed for the following objectives:

- *Software Quality Improvement*
- *Verification and Validation*
- *Software Reliability Estimation*

All the above software testing objectives are discussed in the following sections.

11.4.1 Software Quality Improvement

As computers and software are used in critical applications, the outcome of a bug can be severe. Bugs can cause huge losses. Bugs in critical systems have:

- *Caused airplane crashes,*
- *Allowed space shuttle missions to go awry,*
- *Halted trading on the stock market, and*
- *Worse*

Bugs can kill and cause disasters. The so-called year 2000 (Y2K) bug could have resulted into more screeching halt on the first day of the century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Software quality means the conformance to the specified software design requirements. Being correct, the minimum requirement of quality, means performing as required under specified circumstances.

Debugging, a narrow view of software testing, is performed heavily to find out design defects by the programmer. The imperfection of human nature makes it almost impossible

to make a moderately complex program correct the first time. Finding the problems and get them fixed is the purpose of debugging in programming phase.

11.4.2 Verification and Validation (V&V)

Another important objective of software testing is verification and validation (V&V). Testing can serve as metrics. It is heavily used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results, which either the product works under certain situations, or it does not work.

Software quality cannot be tested directly but the related factors to make quality visible can be tested. Quality has three sets of factors:

- *Functionality,*
- *Engineering, and*
- *Adaptability.*

These three sets of factors can be thought of as dimensions in the software quality space. Each dimension may be broken down into its component factors and considerations at successively lower levels of detail. **Table 11.1** illustrates some of the most frequently cited quality considerations suggested by Hetzel in 1988.

TABLE 11.1: Typical Software Quality Factors

Functionality (exterior quality)	Engineering (interior quality)	Adaptability (future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

Good testing provides measures for all relevant factors. The importance of any particular actor varies from application to application. Any system where human lives are at stake nust place extreme emphasis on reliability and integrity. In the typical business system sability and maintainability are the key factors, while for a one-time scientific program either may be significant. Our testing, to be fully effective, must be geared to measuring ach relevant factor and thus forcing quality to become tangible and visible.

Tests with the purpose of validating the product works are named clean tests, or positive tests. The drawbacks are that it can only validate that the software works for the pecified test cases. A finite number of tests cannot validate that the software works for all ituations. On the contrary, only one failed test is sufficient enough to show that the oftware does not work.

Dirty tests, or negative tests, refer to the tests aiming at breaking the software, or showing that it does not work. A piece of software must have sufficient exception handling capabilities to survive a significant level of dirty tests.

A testable design is a design that can be easily validated, falsified and maintained. Because testing is a rigorous effort and requires significant time and cost, design for testability is also an important design rule for software development.

11.4.3 Software Reliability Estimation

Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to.

The objective of testing is to discover the residual design errors before delivery to the customer. The failure data during the testing process are taken down in order to estimate the software reliability. The testing process may function with regular feedback from the reliability analysis to the testers and designers that is shown in **Fig. 11.2.**

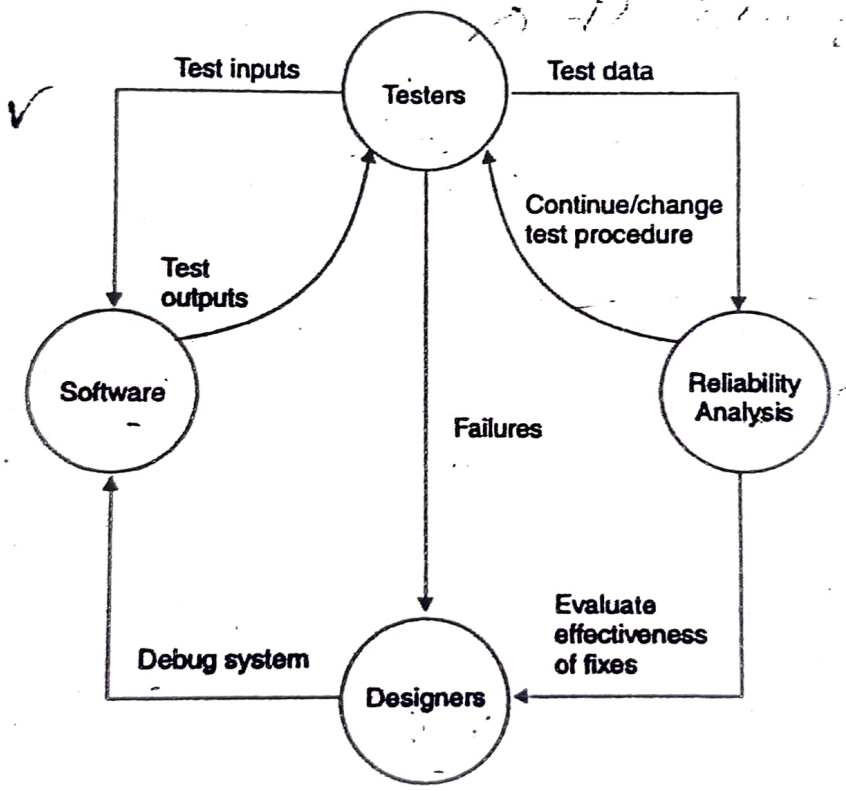

Fig. 11.2: *Role of Reliability in Software Testing*

Based on an operational profile, testing can serve as a statistical sampling method to gain failure data for software reliability estimation.

11.5 Software Bug Characteristics and Bug Types

The following are certain ***characteristics*** of software bugs (Cheung 1990; Pressman 1997):

- *The symptom and the cause of a bug may exist geographically remote from each other.*

- *The symptom may be caused by human error that is not easily traced.*
- *The symptom may be a result of timing problems, rather than processing problems.*
- *The symptom may disappear (temporarily) when another error is corrected.*
- *The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).*
- *It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).*
- *The symptom may be due to causes that are distributed across a number of tasks running on different processors.*
- *The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software.*

There are several types of errors (or bugs) in software (or program). Some important types of software bugs are listed below:

Syntax Errors

- These are generally caught by compiler.

Logic/Algorithm Errors

- These errors occur due to:
  - Branching too soon
  - Branching too late
  - These two branching errors are often associated with off-by-one errors.
  - Testing the wrong condition
  - Initialisation errors
  - Forgetting to test for a particular condition
  - Data type mismatch
  - Incorrect formula or computation

Documentation Errors

- These occur due to mismatch between documentation and code.
- These errors lead to difficulties especially during maintenance.

Capacity Errors

- These errors are due to system performance degradation at capacity.

Hardware Errors

Timing/coordination Errors

- These errors are mainly found in real time systems.
- These errors deal with process coordination and are very difficult to find and correct.

Computation and Precision Errors

- These errors are caused due to rounding/truncation issues, while dealing with real numbers and conversion.

Stress/overload Errors

- These errors are caused when users/device capacities exceed.
- Examples: buffer sizes, etc.

Throughput/performance Errors

- These errors come across due to throughput or performance degradation. For example, response time degradation, etc.

Recovery Errors

- These are error-handling faults.

Standards/Procedures

- These don't cause errors in and of themselves but rather create an environment where errors are created/introduced as the system is tested & modified.

11.6 Reasons For Software Bugs

The following are certain reasons why software bugs exist:

1. ***Miscommunication or no communication***
   - As to specifics of what an application should or shouldn't do (the application's requirements).
2. ***Software Complexity***
   - The complexity of current software applications can be difficult to comprehend for anyone without experience in modern-day software development.
   - Windows-type interfaces, client-server and distributed applications, data communications, enormous relational databases, and sheer size of applications have all contributed to the exponential growth in software/system complexity.
   - And the use of object-oriented techniques can complicate instead of simplify a project unless it is well-engineered.
3. ***Programming Errors***
   - Programmers, like anyone else, can make mistakes.
4. ***Changing Requirements***
   - The customer may not understand the effects of changes, or may understand and request them anyway - redesign, rescheduling of engineers, effects on other projects, work already completed that may have to be redone or thrown out, hardware requirements that may be affected, etc.

- If there are many minor changes or any major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems, and the complexity of keeping track of changes may result in errors.
- Enthusiasm of engineering staff may be affected. In some fast-changing business environments, continuously modified requirements may be a fact of life. In this case, management must understand the resulting risks, and QA and test engineers must adapt and plan for continuous extensive testing to keep the inevitable bugs from running out of control

5. *Time Pressures*
   - Scheduling of software projects is difficult at best, often requiring a lot of guesswork.
   - When deadlines loom and the crunch comes, mistakes are made.
6. *Egos*
   - People prefer to say things like: *'no problem' 'piece of cake' 'I can whip that out in a few hours' 'it should be easy to update that old code'* instead of: *'that adds a lot of complexity and we could end up making a lot of mistakes' 'we have no idea if we can do that; we'll wing it' 'I can't estimate how long it will take, until I take a close look at it' 'we can't figure out what that old spaghetti code did in the first place'*. If there are too many unrealistic 'no problems', the result is bugs.
7. *Poorly Documented Code*
   - It's tough to maintain and modify code that is badly written or poorly documented; the result is bugs.
   - In many organizations management provides no incentive for programmers to document their code or write clear, understandable code.
   - In fact, it's usually the opposite: they get points mostly for quickly turning out code, and there' job security if nobody else can understand it.
   - Software development tools - visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

11.7 Principles of Software Testing

Software testing is a critical component of the software engineering process. It is an element of software quality assurance and can be described as a process of running a program in such a manner as to uncover any errors. This process, while seen by some as tedious, tiresome and unnecessary, plays a vital role in software development.

The process of software testing involves creating test cases to "break the system" but before these can be designed, a few principles have to be observed:

Software testing is an extremely creative and intellectually challenging task. The following are some important principles that should be kept in mind while carrying softw testing (Shooman 1983; Sommerville 1998; Pressman 1997):

- ***Testing should be based on user requirements.*** This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.
- ***Testing time and resources are limited.*** Avoid redundant tests.
- ***It is impossible to test everything.*** Exhaustive tests of all possible scenarios are impossible, simple because of the many different variables affecting the system and the number of paths a program flow might take.
- ***Use effective resources to test.*** This represents use of the most suitable tools, procedures and individuals to conduct the tests. The test team should use tools that they are confident and familiar with. Testing procedures should be clearly defined. Testing personnel may be a technical group of people independent of the developers.
- ***Test planning should be done early.*** This is because test planning can begin independently of coding and as soon as the client requirements are set.
- ***Test for invalid and unexpected input conditions as well as valid conditions.*** The program should generate correct messages when an invalid test is encountered and should generate correct results when the test is valid.
- ***The probability of the existence of more errors in a module or group of modules is directly proportional to the number of errors already found.***
- ***Testing should begin at the module.*** The focus of testing should be concentrated on the smallest programming units first and then expand to other parts of the system.
- ***Testing must be done by an independent party.*** Testing should not be performed by the person or team that developed the software since they tend to defend the correctness of the program.
- ***Assign best personnel to the task.*** Because testing requires high creativity and responsibility only the best personnel must be assigned to design, implement, and analyze test cases, test data and test results.
- ***Testing should not be planned under the implicit assumption that no errors will be found.***
- ***Testing is the process of executing software with the intent of finding errors.***
- ***Keep software static during test.*** The program must not be modified during the implementation of the set of designed test cases.
- ***Document test cases and test results.***
- ***Provide expected test results if possible.*** A necessary part of test documentation is the specification of expected results, even if providing such results is impractical.

11.8 Software Testability

Testability is the ability of software (or program) with which it can easily be tested (Pressman 1997; Conte 1996). The following are some key characteristics of testability:

- *The better it works, the more efficient is testing process.*
- *What you see is what you test (WYSIWYT).*
- *The better it is controlled, the more we can automate or optimise the testing process.*
- *By controlling the scope of testing we can isolate problems and perform smarter retesting.*
- *The less there is to test, the more quickly we can test it.*
- *The fewer the changes, the fewer the disruptions to testing.*
- *The more information we have, the smarter we will test.*

11.9 Software Testing Process

Except for small programs, systems should ***not*** be tested as a single unit.

Large systems are built out of *sub-systems*, which are built out of *modules* that are *composed of procedures and functions*. The testing process should therefore proceed in stages where testing is carried out *incrementally* in conjunction with system implementation.

The most widely used testing process consists of five stages that are illustrated in **Table 11.2.**

TABLE 11.2: Testing Process Stages

Component Testing	*Unit Testing*	Verification (Process-Oriented)	**White Box Testing Techniques** (Tests that are derived from knowledge of the program's structure and implementation)
	Module Testing		
Integrated Testing	*Sub-System Testing*		
	System Testing		
User Testing	*Acceptance Testing*	Validation (Product-Oriented)	**Black Box Testing Techniques** (Tests that are derived from the program specification)

However, as defects are discovered at any one stage, they require program modifications to correct them and this may require other stages in the testing process to be repeated.

Errors in program components, say may come to light at a later stage of the testing process. The process is therefore an ***iterative*** one with information being fed back from later stages to earlier parts of the process.

The iterative testing process is illustrated in **Fig.11.3** and described below:

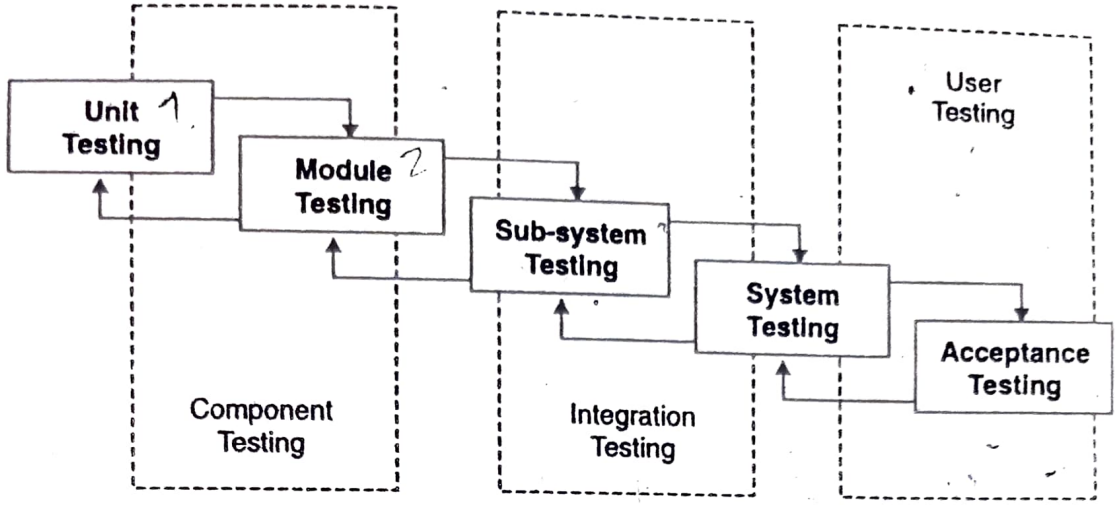

Fig. 11.3: *Iterative Software Testing Process*

Unit Testing

Unit testing is code-oriented testing. *Individual components* are tested to ensure that they operate correctly. Each component is tested *independently*, without other system components.

Module Testing

A module is a *collection of dependent components* such as an object class, an abstract data type or some looser collection of procedures and functions. A module *encapsulates related components* so it can be tested without other system modules.

Sub-system Testing

This phase involves testing *collections of modules*, which have been *integrated into sub-systems*. It is a design-oriented testing and is also known as *integration testing*.

Sub-systems may be independently designed and implemented. The most common problems, which arise in large software systems, are sub-systems *interface mismatches*. The sub-system test process should therefore concentrate on the *detection of interface errors* by rigorously exercising these interfaces.

System Testing

The sub-systems are integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between sub-systems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.

Acceptance Testing

This is the final stage in the testing process before the system is *accepted for operational use*. The system is tested with *data supplied* by the system client rather than simulated test

data. Acceptance testing may reveal errors and omissions in the *systems requirements definition (user-oriented)* because real data exercises the system in different ways from the test data.

Acceptance testing may also reveal requirement problems where the system facilities do not really meet the users needs (*functional*) or the *system performance (non-functional)* is unacceptable.

11.10 Static Analysis

Static analysis seeks to detect errors without direct execution of the test object. Static analysis techniques do not necessitate the execution of the software.

The activities involved in static testing concern *syntactic, structural and semantic analysis of the test object.* The goal is to localize, as early as possible, error-prone parts of the test object.

The most important activities of static program analysis are:

- *Code inspection*
- *Complexity analysis*
- *Structural analysis*
- *Data-flow analysis*

All these activities are discussed in the following sections.

11.10.1 Code Inspection

Code inspection is a useful technique for localizing design and implementation errors. Many errors prove easy to find if the author would only read the program carefully enough. Code inspections have already been discussed in detail in the earlier Chapter 10.

The idea behind code inspection is to have the author of a program discuss it step by step with other software engineers. The moderator notes every detected error and the inspection continues.

The *task of the inspection team* is to *detect, not to correct,* errors. Only on completion of the inspection do the designer and the implementer begin their correction work.

11.10.2 Complexity Analysis

The *goal* of *complexity analysis* is to *establish metrics* for the complexity of a program.

These metrics include complexity measures for modules, nesting depths for loops, lengths of procedures and modules, import and use frequencies for modules, and the complexity of interfaces for procedures and methods.

The results of complexity analysis permit statements about the quality of a software product (within certain limits) and localization of error-prone positions in the software system.

Complexity is difficult to evaluate objectively; a program that a programmer perceives as simple might be viewed as complex by someone else.

11.10.3 Structural Analysis

The *goal* of *structural analysis* is to uncover structural anomalies of a test object.

11.10.4 Data-flow Analysis

Data-flow analysis is intended to help discover data-flow anomalies. Data-flow analysis provides information about whether a data object has a value before its use and whether a data object is used after an assignment.

Data-flow analysis applies to both the body of a test object and the interfaces between test objects.

11.11 Dynamic Analysis

For *dynamic testing* the test objects are executed or simulated. Dynamic analysis is what is generally considered as *testing* i.e. it involves running the system.

Dynamic testing is an imperative process in the software life cycle. Every procedure, every module and class, every subsystem and the overall system must be tested dynamically, even if static tests and program verifications have been carried out.

The activities for dynamic testing include:

- *Preparation of the test object for error localization*
- *Availability of a test environment*
- *Selection of appropriate test cases and data*
- *Test execution and evaluation*

11.12 Software Test Design

The design of software tests is subject to the same basic engineering principles as the design of software. Good design consists of a number of stages that progressively elaborate the design of tests from an initial high-level strategy to detailed test procedures (Pressman 1997; Sommerville 1998). These stages are:

- *Test Strategy*
- *Test Planning*
- *Test Case design*
- *Test Procedures*
- *Test Results Documentation*

The design of tests has to be driven by the specification of the software. At the highest level this means that tests will be designed to verify that the software faithfully implements the requirements of the *Functional Specification*.

At lower levels tests will be designed to verify that items of software implement all design decisions made in the *Design Specification* and *Detailed Design Specifications*. As with any design process, each stage of the test design process should be subject to informal and formal review.

The ease with which tests can be designed is highly dependent on the design of the software. It is important to consider testability as a key requirement for any software development.

11.12.1 Test Strategy

The first stage is the formulation of a test strategy. A test strategy is a statement of the overall approach to testing, identifying what levels of testing are to be applied and the methods, techniques and tools to be used. A test strategy should ideally be organisation wide, being applicable to all of organisations' software development.

Developing a test strategy that efficiently meets the needs of an organisation is critical to the success of software development within the organisation. The application of a test strategy to a software development project should be detailed in the project's software quality plan.

11.12.2 Test Plans

The next stage of test design is the development of a test plan. A test plan states:

- *What the items to be tested are?*
- *At what level they will be tested?*
- *What sequence they are to be tested in?*
- *How the test strategy will be applied to the testing of each item and describes the test environment.*

A test plan may be project wide, or may in fact be a hierarchy of plans relating to the various levels of specification and testing:

Acceptance Test Plan

- Describing the plan for acceptance testing of the software.
- This would usually be published as a separate document, but might be published with the system test plan as a single document.

System Test Plan

- Describing the plan for system integration and testing.
- This would also usually be published as a separate document, but might be published with the acceptance test plan.

Software Integration Test Plan

- Describing the plan for integration of tested software components.
- This may form part of the Design Specification.

Unit Test Plan(s)

- Describing the plans for testing of individual units of software.
- These may form part of the Detailed Design Specifications.

The objective of each test plan is to provide a plan for verification, by testing the software, the software produced fulfils the functional or design statements of the appropriate software specification. In the case of acceptance testing and system testing, this means the Functional Specification.

11.12.3 Test Case Design

Test cases should be designed in such a way as to uncover quickly and easily as many errors as possible. They should "exercise" the program by using and producing inputs and outputs that are both correct and incorrect.

Once the test plan for a level of testing has been written, the next stage of test design is to specify a set of test cases or test paths for each item to be tested at that level. A number of test cases will be identified for each item to be tested at each level of testing. Each test case will specify how the implementation of a particular requirement or design decision is to be tested and the criteria for success of the test.

The objective of test case design is to test all modules and then the whole system as completely as possible using a reasonably wide range of conditions.

Variables should be tested using all possible values (for small ranges) or typical and out-of-bound values (for larger ranges). They should also be tested using valid and invalid types and conditions. Arithmetical and logical comparisons should be examined as well, again using both correct and incorrect parameters.

The test cases may be documented with the test plan, as a section of a software specification, or in a separate document called a test *specification* or *test description*.

Acceptance Test Specification

- Specifying the test cases for acceptance testing of the software.
- This would usually be published as a separate document, but might be published with the acceptance test plan.

System Test Specification

- Specifying the test cases for system integration and testing.
- This would also usually be published as a separate document, but might be published with the system test plan.

Software Integration Test Specifications

- Specifying the test cases for each stage of integration of tested software components.
- These may form sections of the Design Specification.

Unit Test Specifications

- Specifying the test cases for testing of individual units of software.
- These may form sections of the Detailed Design Specifications.

It is important to design test cases for both *positive testing* and *negative testing*. *Positive testing* checks that the software does what it should. *Negative testing* checks that the software doesn't do what it shouldn't.

The process of designing test cases, including executing them as thought experiments, will often identify bugs before the software has even been built. It is not uncommon to find more bugs when designing tests than when executing tests.

11.12.4 Test Procedures

The final stage of test design is to implement a set of test cases as a test procedure, specifying the exact process to be followed to conduct each of the test cases. This is a fairly straightforward process, which can be equated to designing units of code from higher-level functional descriptions.

For each item to be tested, at each level of testing, a test procedure will specify the process to be followed in conducting the appropriate test cases. A test procedure cannot leave out steps or make assumptions. The level of detail must be such that the test procedure is deterministic and repeatable.

Test procedures should always be separate items, because they contain a great deal of detail that is irrelevant to software specifications.

11.12.5 Test Results Documentation

When tests are executed, the outputs of each test execution should be recorded in a *Test Results File*. These results are then assessed against criteria in the test specification to determine the overall outcome of a test.

Each test execution should also be noted in a test log. The test log will contain records of when each test has been executed, the outcome of each test execution, and may also include key observations made during test execution. Often a test log is not maintained for lower levels of testing (unit test and software integration test).

Test reports may be produced at various points during the testing process. A test report will summarise the results of testing and document any analysis. An acceptance test report often forms a contractual document within which acceptance of software is agreed.

11.13 Types of Software Testing

There are many ways to conduct software testing, but the most common methods rely on the common steps.

During the implementation phase, modules are informally tested by the programmer while they are being coded; this is referred to as *desk checking*. After the programmer is satisfied that the module appears to function correctly, methodical testing of the module is undertaken by a separate testing team.

There are two basic types of methodical testing:

- **Non-execution-based testing:** the module is reviewed by a team.
- **Execution-based testing:** the module is run against test cases.

Both above testing types are discussed below.

11.13.1 Non-execution Based Testing

The non-execution based testing relies on fault-detection strategy. The fault-detecting power of these non-execution-based techniques leads to rapid, thorough, and early fault detection. The additional time required for code reviews is more than repaid by the increased productivity due to fewer faults at the integration phase.

In general, non-execution-based code testing is less expensive than execution-based testing because:

- *Execution-based testing (running test cases) can be extremely time-consuming, and*
- *Reviews lead to detection of faults earlier in the lifecycle.*

Non-execution-based testing is also known as static testing (or static program analysis), which is already discussed above in this chapter.

11.13.2 Execution Based Testing

In this type of testing, the modules are run against test cases. Following are the two ways of systematically constructing test data to test a module:

- *Black-Box Testing:* The code itself is ignored; the only information used in designing test cases is the specification document.
- *White-Box Testing:* The code itself is tested, without regard of the specifications.

All the above testing techniques are discussed below in more details (Bezier 1990).

11.14 Black-box Testing

Black-box test design treats the system as a *black-box*. So it is a software testing technique whereby the internal workings of the item being tested are not known by the tester. It is also known as *functional testing*.

Other names for black-box testing include: *specifications* testing, *behavioral* testing, *data-driven* testing, *functional* testing, and *input/output-driven* testing.

Generally, black-box testing attempts to uncover the following:

- *Incorrect functions*
- *Data structure errors*
- *Missing functions*
- *Performance errors*
- *Initialisation and termination errors*
- *External database access errors*

The functionality of each module is tested with regards to its specifications (requirements) and its context (events). Only the correct input/output relationship is scrutinized.

The black-box testing is illustrated in **Fig. 11.4.**

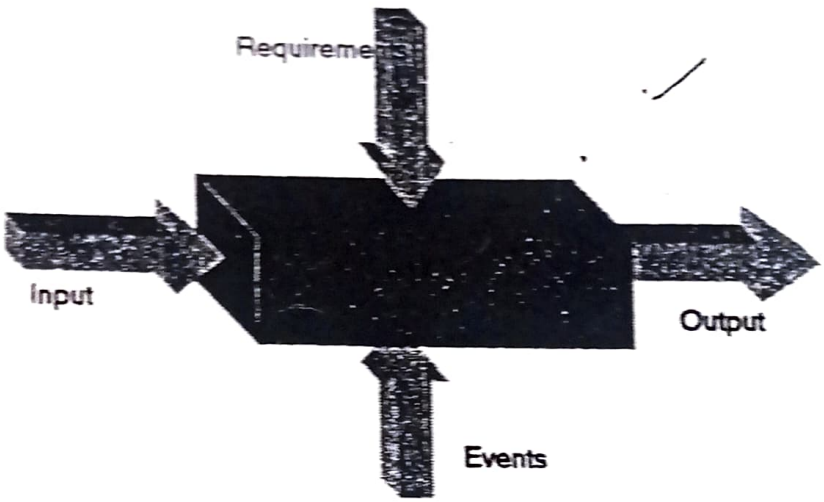

Fig. 11.4: *Black-box Testing.*

For example, in a black box test on software design the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. The tester does not ever examine the programming code and does not need any further knowledge of the program other than its specifications.

In general, every combination of input and output would require in inordinate number of test cases. Consequently, exhaustive black-box testing is usually either impossible or unreasonable. The art of testing is to design a small, manageable set of test cases so as to maximize the chances of detecting a fault while minimizing the redundancy amongst of the cases.

11.14.1 Advantages of Black-box Testing

The advantages of this type of testing include:

- *The test is unbiased because the designer and the tester are independent of each other.*

- *The tester does not need knowledge of any specific programming languages.*
- *The test is done from the point of view of the user, not the designer.*
- *Test cases can be designed as soon as the specifications are complete.*

11.14.2 Disadvantages of Black-box Testing

The disadvantages of this type of testing include:

- *The test can be redundant if the software designer has already run a test case.*
- *The test cases are difficult to design.*

Testing every possible input stream is unrealistic because it would take an inordinate amount of time; therefore, many program paths will go untested.

11.15 Black-box Testing Techniques

The following are main techniques to black-box testing:

- *Equivalence Class Partitioning*
- *Boundary Value Analysis (BVA)*
- *Cause-Effect Graphs*
- *Comparison Testing*

Equivalence testing, combined with boundary value analysis, is a black-box technique of selecting test cases in such a way that new cases are chosen to detect previously undetected faults.

Cause-effect graph overcomes the weakness of the above two methods of not considering potential combinations of input/output conditions.

All the above techniques are discussed below in more details.

11.15.1 Equivalence Class Partitioning

This method divides the input of a program into classes of data. Test case design is based on defining an equivalent class for a particular input. An equivalence class represents a set of valid and invalid input values.

An *equivalence* class is a set of test cases such that any one member of the class is representative of any other member of the class.

Suppose the specifications for a database product state that the product must be able to handle any number of records from 1 through 16,383. If the product can handle 34 records and 14,870 records, then the chances are good that it will work fine for, say, 8534 records. If the product works correctly for any one test case in the range 1 to 16,383, then it will probably work for any other test case in the range. The range from 1 to 16,383 constitutes an *equivalence class*.

For this product, there are three equivalence classes:

- *Equivalence class 1: Less then one record.*
- *Equivalence class 2: From 1 to 16,383 records.*
- *Equivalence class 3: More than 16,383 records.*

Testing the database product then requires that one test class from each equivalence class be selected.

Guidelines For Equivalence Partitioning

Equivalence classes may be defined according to the following guidelines (Pressman 1997):

- *Range:* If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
- *Specific Value:* If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
- *Member of a Set:* If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
- *Boolean:* If an input condition is Boolean, one valid and one invalid class are defined.

11.15.2 Boundary Value Analysis

Boundary value analysis (BVA) is complementary to equivalence partitioning. Rather than selecting arbitrary input values to partition the equivalence class, the test case designer chooses values at the extremes of the class.

Furthermore, boundary value analysis also encourages test case designers to look at output conditions and design test cases for the extreme conditions in output.

A successful test case is one that detects a previously undetected fault. In order to maximize the chances of finding a new fault, a high-payoff technique is *boundary-value analysis.*

Experience has shown that when a test case on or just to one side of a boundary of an equivalence class is selected, the probability of detecting a fault increases. Thus, when testing the database product, the following cases should be selected:

Test case 1	0 records	Member of equivalence class 1 and adjacent to boundary value
Test case 2	1 record	Boundary value
Test case 3	2 records	Adjacent to boundary value
Test case 4	723 records	Member of equivalence class 2
Test case 5	16,382 records	Adjacent to boundary value
Test case 6	16,383 records	Boundary value
Test case 7	16,384 records	Member of equivalence class 3 and adjacent to boundary value

This example applies to the input specifications; the same technique should be applied to the output specifications.

The use of equivalence classes, together with boundary value analysis, is a valuable technique for generating a relatively small set of test data with a high probability of uncovering most faults.

The following are some important guidelines for performing boundary value analysis:

Range

- For an input range bounded by values **x** and **y**, test cases should be designed with values **x** and **y**, and values just above and just below **x** and **y**.

Maximum and Minimum Numbers

- If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers.
- Values above and below the minimum and maximum are also tested.

Output Conditions

- Apply the above guidelines to output conditions.

Internal Data Structures

- For internal data structures, be certain to design test cases to exercise the data structure at its boundary.

11.15.3 Cause-Effect Graphs

A weakness of the above two methods is that they do not consider potential combinations of input/output conditions. Cause-effect graphs connect input classes (**causes**) to output classes (**effects**) yielding a directed graph.

Cause-effect graphing is a test case design approach that offers a concise depiction of logical conditions and associated actions.

A simplified version of cause-effect graph symbology is shown below. The left hand column of the figure gives the various logical associations among causes and effects. The dashed notation in the right-hand columns indicates potentials constraining associations that might apply to either causes or effects.

Sample Symbols

Sample symbols used for drawing cause-effect graphs are illustrated in **Fig. 11.5.**

Following are some important guidelines for cause-effect graphs:

- *Causes and effects are listed for modules and an identifier is assigned to each.*
- *A cause-effect graph is developed (special symbols are required).*
- *The graph is converted to a decision table.*
- *Decision table rules are converted to test cases.*

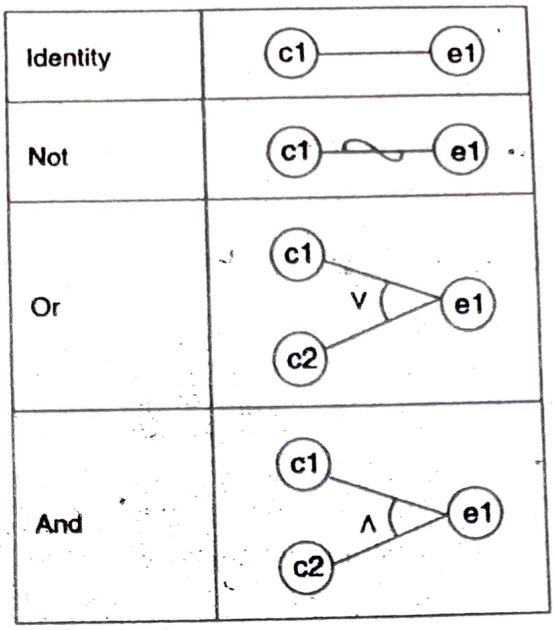

Fig. 11.5: *Cause-Effect Graph Symbols*

Cause-Effect Graphs translate equivalence partitions into decision table form via Boolean operator descriptions of the output conditions in terms of the input variables. Test data can be generated from the decision table form, reducing the number required. Some authors state that partition testing is more effective than testing with randomly generated data. However, random testing is more cost effective in terms of time and manpower.

11.15.4 Comparison Testing

For critical applications requiring fault-tolerance, a number of independent versions of software are developed for the same specifications. In that case, test cases using black-box method are applied to each version.

If output from each version is the same, then it is assumed that all implementations are correct. If the output is different, each version is examined to see if is responsible for the differing output.

Comparison testing is not foolproof. If the specification applied to all versions is incorrect, all versions will likely reflect the error, and there may be the same output (Sommerville 1998; Pressman 1997).

11.16 White-box Testing

White-box test design allows one to peek inside the *box*, and it focuses specifically on using internal knowledge of the software to guide the selection of test data.

White-box testing is also known by other names such as *Glass-Box testing, Structural testing, Clear Box testing, Open-box testing, Logic-driven testing,* and *Path-oriented testing*.

In white-box testing, test cases are selected on the basis of examination of the code, rather than the specifications. White-box testing is illustrated in **Fig. 11.6.**

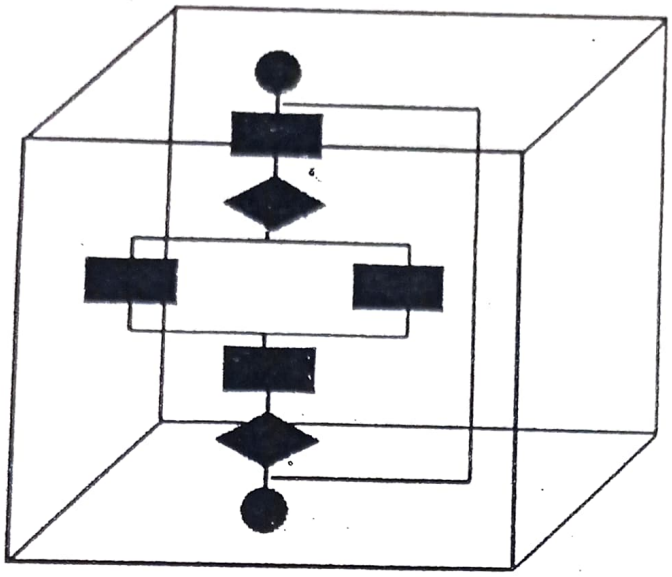

Fig. 11.6: *White-box Testing*

White-box testing is a software testing approach that examine the program structure and derive test data from the program logic. Structural testing is sometimes referred to as clear-box testing since white boxes are considered opaque and do not really permit visibility into the code.

White-box testing requires the intimate knowledge of program internals, while black box testing is based solely on the knowledge of the system requirements. Being primarily concerned with program internals, it is obvious in software engineering literature that the primary effort to develop a testing methodology has been devoted to glass box tests. However, since the importance of black box testing has gained general acknowledgement, also a certain number of useful black box testing techniques were developed.

It is important to understand that these methods are used during the test design phase, and their influence is hard to see in the tests once they're implemented.

11.16.1 Advantages of White-box Testing

The major advantages of white-box testing include the following:

- *Forces test developer to reason carefully about implementation.*
- *Approximates the partitioning done by execution equivalence.*
- *Reveals errors in hidden code.*
- *Beneficent side effects.*
- *Optimisations.*

11.16.2 Disadvantages of White-box Testing

The following are few disadvantages of white-box testing:

- *Expensive*
- *Miss cases omitted in the code*

11.17 White-box Testing Techniques

There are a number of different forms of white-box testing. The important of these include the following:

- *Basis Path Testing*
- *Structural Testing*
- *Logic-based Testing*
- *Fault-Based Testing*

All the above white-box testing techniques are discussed below.

11.17.1 Basis Path Testing

Basis path testing is a white-box technique. It allows the design and definition of a basis set of execution paths. The test cases created from the basis set allow the program to be executed in such a way as to examine each possible path through the program by executing each statement at least once (Pressman 1997).

To be able to determine the different program paths, the engineer needs a representation of the logical flow of control. The control structure can be illustrated by a flow graph. A *flow graph* can be used to represent any procedural design.

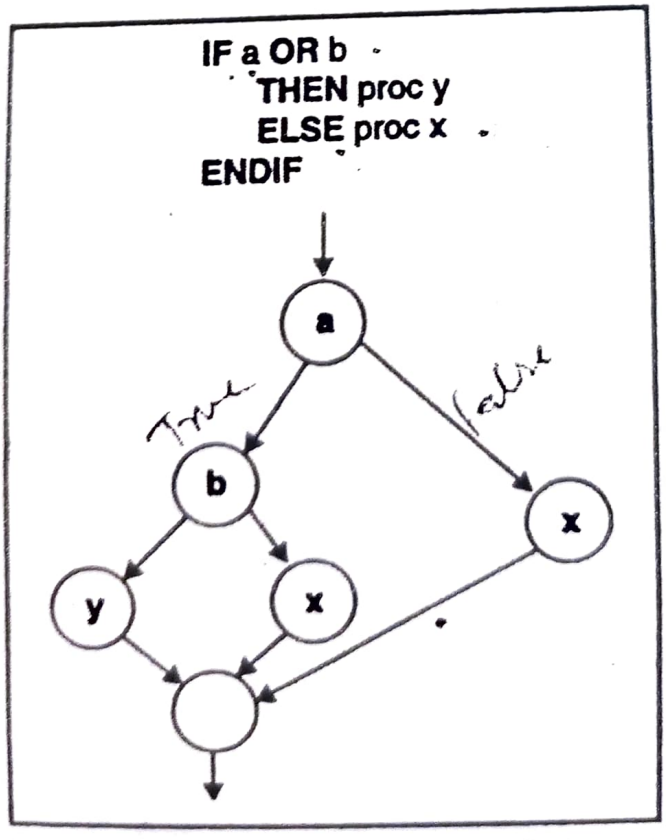

Fig. 11.7: *Flow Graph of an 'If-then-else' Statement*

The program condition IF-ELSE-ENDIF and its corresponding control flow graph (CFG) are illustrated in **Fig. 11.7.**

Next a metric, Cyclomatic Number, can be used to determine the number of independent paths (McCabe 1976). It is also called *Cyclomatic Complexity* and provides the number of test cases that have to be designed. Cyclomatic complexity is discussed in more details in Chapter 12. This insures coverage of all program statements (Pressman, 1997).

11.17.2 Structural Testing

Structural testing examines source code and analyses what is present in the code. Structural testing techniques are often dynamic, meaning that code is executed during analysis. This implies a high-test cost due to compilation or interpretation, linkage, file management and execution time. Test cases are derived from analysis of the *program control flow.*

A *Control Flow Graph* is a representation of the flow of control between program regions such as a group of statements bounded by a single entry and exit point.

Structural testing cannot expose errors of code omission but can estimate the test suite adequacy in terms of code coverage, that is, execution of components by the test suite or its fault-finding ability.

The following are some important types of structural testing:

- *Statement Coverage Testing*
- *Branch Coverage Testing*
- *Condition Coverage Testing*
- *Loop Coverage Testing*
- *Path Coverage Testing*
- *Domain and Boundary Testing*
- *Data-flow Testing*

All the above structural testing techniques are discussed below.

11.17.2.1 *Statement Coverage Testing*

This is the simplest form of white-box testing, whereby a series of test cases are run such that each statement is executed at least once. Its achievement is insufficient to provide confidence in a software product's behavior.

Often, a CASE tool keeps track of how many times each statement has been executed. A weakness of this approach is that there is no guarantee that all outcomes of branches are properly tested.

Example:

```
if (s>1 && t==0)
    x = 9;
Test case: s = 2; t = 0;
```

Here, the programmer has made a mistake; the compound conditional (s>1 && t==0) should have been (s>1 || t==0). The chosen test data, however, allow the statement x = 9 to be executed without the fault being detected.

11.17.2.2 *Branch Coverage Testing*

Branch coverage is an improvement over statement coverage, in that a series of tests are run to ensure that all branches are tested at least once. It is also called decision coverage. Techniques such as statement or branch coverage are called *structural tests.*

Branch coverage requires sufficient test cases for each program decision or branch to be executed so that each possible outcome occurs at least once. It is considered to be a minimum level of coverage for most software products, but decision coverage alone is insufficient for high-integrity applications.

11.17.2.3 *Condition Coverage Testing*

This criteria requires sufficient test cases for each condition in a program decision to take on all possible outcomes at least once. It differs from branch coverage only when multiple conditions must be evaluated to reach a decision.

Multi-Condition Coverage requires sufficient test cases to exercise all possible combinations of conditions in a program decision.

11.17.2.4 *Loop Coverage Testing*

This criteria requires sufficient test cases for all program loops to be executed for zero, one, two, and many iterations covering initialisation, typical running and termination (boundary) conditions.

11.17.2.5 *Path Coverage Testing*

Path coverage is the most powerful form of white-box testing; all paths are tested. The number of paths in a product with loops can be very large, however, and methods have been devised to reduce the number of paths to be examined. This technique is capable to uncover more faults than by branch testing (Woodward 1980).

This criteria requires sufficient test cases for each feasible path, basis path, etc., from start to exit of a defined program segment, to be executed at least once. Because of the very large number of possible paths through a software program, path coverage is generally not achievable. The amount of path coverage is normally established based on the risk or criticality of the software under test.

One criterion for selecting paths is to restrict test cases to *linear code sequences*. To do this, one first identifies the set of points, ***L***, from which control flow may jump. The set ***L*** contains entry and exit points and branch statements. The linear code sequences are those paths that begin at an element of ***L*** and end at an element of ***L***.

Another method of reducing the number paths is all-definition-use-path coverage. In this technique, each occurrence of a variable is labelled either as a definition of the variable or a use of the variable.

11.17.2.6 *Domain and Boundary Testing*

Domain Testing is a form of path coverage. Path domains are a subset of the program input that causes execution of unique paths. The input data can be derived from the program control flow graph. Test inputs are chosen to exercise each path and also the boundaries of each domain.

For example, in a program analysing the height and weight of a population, the input domain is height and weight, where the inputs are real numbers greater than zero and bounded by some upper limit. The statement

'if (weight >= 50.0 and height >= 1.8) then S1 else S2'

would partition the path domain into two from the true and false evaluation of the predicate. A true evaluation would result in statement S1 being executed and S2 is executed when the predicate evaluates to false.

Test inputs for branch, statement and domain testing could be:

Test 1: weight = 48.0 height = 1.8

Test 2: weight = 50.0 height = 1.8

A boundary test would incorporate test inputs on and slightly off the boundaries of the paths. To determine data slightly off the boundary an amount, ϵ, must be added or subtracted to the value which lies on the boundary.

When the boundary is determined by an integer, ϵ is 1. That is, the value 1 must be added or subtracted to the value in a predicate to form an input value that will be close to the domain boundary. When working with real numbers the procedure is more complex. The value ϵ must be the smallest number distinguishable by the base system of the program under test. For example, if the reals are single precision Î could be of the order of 0.000001.

To test the boundary of 'weight = 50.0' the following three input cases would be valid:

Test 1: weight = 50.0, height = 1.8

Test 2: weight = 50.0, height = 1.6

Test 3: weight = 50.000001, height = 1.8

Great care must be taken when working with real numbers in predicates because of the precision problems of reals. Boundary testing aids the identification of these problems and errors of path selection. However, domain and boundary analysis is only suitable for programs with a small number of input variables and with simple linear predicates.

11.17.2.7 *Data Flow Testing*

Data flow testing focus on the points at which variables receive values and the points at which the values are used (Pressman 1997; Sommerville 1998). This kind of testing serves as a reality check on path testing and that's why many authors view it as a *path testing* technique.

This technique requires sufficient test cases for each feasible data flow to be executed at least once. *Data flow analysis* studies the sequences of actions and variables along program paths. It can be considered and applied as both static and as a dynamic technique.

Test data must traverse all the interactions between a variable definition and each of its uses. The program path between a variable *definition* and a *use* without an intervening definition is known as a DU path. Variables can be created, killed and/or used.

Variable uses may be in predicates, *p uses*, in which the variable is referenced in the conditional expression. A computational use, *c use*, refers to all other references of variables. Testing all-DU-paths subsumes all other data flow testing criteria. The all-DU-paths criteria require every definition clear sub-path to be loop-free or to include a simple, one iteration, loop.

Data-flow testing uses the flow graph to explore unreasonable things that can happen to data. Enough paths must be selected to ensure that each and every variable has been initialised prior to its use or that all defined variables have been used or referenced for something.

Data flow testing is considered viable for incorrect uses of variables and constants as well as misspelled identifiers. As with code coverage strategies, data flow testing cannot detect missing statements.

11.17.3 Logic Based Testing

Logic based testing is used when the input domain and resulting processing are amenable to a decision table representation. The following steps are applied to develop a decision table:

1. *List all actions that can be associated with a specific algorithm*
2. *List all conditions (or decisions made) during execution of the algorithm*
3. *Associate specific conditions with specific actions eliminating impossible combinations of conditions*
4. *Define rules by indicating what action(s) occurs for a set of conditions*

Alternatively, for step 3, develop every possible permutation of conditions. This will reveal any inconsistencies or gaps in the user specification and can thus be corrected.

11.17.4 Fault Based Testing

Fault based testing attempts to show the absence of certain classes of faults in code is analysed for uninitialised or unreferenced variables, parameter type checking, etc. The main technique, and its variants, which perform fault based testing is called *Mutation Analysis*, which is discussed below.

11.17.4.1 *Mutation Analysis*

Mutation Analysis (MA) is a fault-based technique for determining the adequacy of a test suite in terms of its test effectiveness. Mutation analysis is one of the most thorough of testing techniques. Empirical studies have shown it to be more stringent than other techniques.

A mutant is a copy of the original test program with one component, such as an operand or operator, altered to simulate a syntactically correct programming fault. The syntactic transformation is a mutation.

Example:

while (index > 10) do could be mutated to ***while (index >=10) do***

Thus, mutation analysis simulates simple programming errors. The test suite must be enhanced until all non-equivalent mutants are detected by generating incorrect output. It incorporates strategies from coverage, data flow anomaly and domain testing strategies.

For example, the above statement has to be traversed by the test input ***index = 10*** to differentiate between the correct and the incorrect version. All statements, branches and (some) paths must be executed to differentiate incorrect mutants from the original program.

By altering the constant 10, in the example, to the constants 11 and 9, boundary testing is performed. The test inputs must include cases of ***index = 9, 10 and 11*** to detect those mutants. By altering the definition of 'index' or replacing use of it by another integer variable in scope, data flow anomalies can be detected.

Mutation analysis provides the tester with guidelines for the development of the test suite. However, it is resource intensive requiring a large number of mutants to be created and executed on the test suite. Research indicates that the number of mutants varies with the number of code statements and variables squared.

A mutation test of a large program, such as would be found in an industrial or commercial environment, would require the generation of a substantial number of mutants. A test on this scale would require management of resources. A strategy must be found to make mutation testing applicable to unit testing in a reasonable time scale and without tying up valuable resources such as time and manpower.

11.18 Software Testing Strategies

Software testing strategy provides a road map for the software developer, quality assurance organization, and the customer.

A strategy must provide guidance for the practitioner and a set of milestones for the manager. Progress must be measurable and problems must surface as soon as possible.

Common characteristics of software testing strategies include the following:

- *Testing begins at the module level and works outward toward the integration of the entire system*
- *Different testing techniques are appropriate at different times*
- *Testing is conducted by the developer and, for large projects, by an independent test group*
- *Testing and debugging are different activities, but debugging must be accommodated in any testing strategy*

In order to conduct a proper and thorough set of tests, the types of testing mentioned below should be performed in the order in which they are described (Basili 1987).

- *Unit Testing*
- *Integration Testing*
- *Functional Testing*
- *Systems and Acceptance Testing*

However, some system or hardware can happen concurrently with software testing.

11.18.1 Unit Testing

Unit testing procedure utilizes the white-box method and concentrates on testing individual programming units. These units are sometimes referred to as *modules* or *atomic modules* and they represent the smallest programming entity.

Unit testing is essentially a set of path test performed to examine the many different paths through the modules. These types of tests are conducted to prove that all paths in the program are solid and without errors and will not cause abnormal termination of the program or other undesirable results.

11.18.2 Integration Testing

Integration testing focuses on testing multiple modules working together. Two basic types of integration are usually used:

- *Top-down Integration or*
- *Bottom up Integration*

Above types of integration are discussed below.

Top-down Integration

As the term suggests, starts at the top of the program hierarchy and travels down its branches. This can be done in either depth-first (shortest path down to the deepest level) or breadth-first (across the hierarchy, before proceeding to the next level).

The main advantage of this type of integration is that the basic skeleton of the program/ system can be seen and tested early.

The main disadvantage is the use of program stubs until the actual modules are written. This basically limits the up-flow of information and therefore does not provide for a good test of the top-level modules.

Bottom-up Integration

This type of integration has the lowest level modules built and tested first on individual bases and in clusters using test drivers. This insures each module is fully tested before its utilized by its calling module.

This method has a great advantage in uncovering errors in critical modules early.

Main disadvantage is the fact that most or many modules must be build before a working program can be presented.

Integration testing procedure can be performed in three ways (Humphrey 1989; Humphrey 1995):

- *Top-down Strategy*
- *Bottom-up Strategy*
- *Big-Bang Strategy*
- *Sandwiched Strategy*

All the above integration testing strategies are discussed below.

11.18.2.1 *Top-Down Strategy*

Top down integration is basically an approach where modules are developed and tested starting at the top level of the programming hierarchy and continuing with the lower levels.

It is an incremental approach because we proceed one level at a time. It can be done in either *"depth"* or *"breadth"* manner.

- *Depth* means we proceed from the top level all the way down to the lowest level.
- *Breadth*, on the other hand, means that we start at the top of the hierarchy and then go to the next level. We develop and test all modules at this level before continuing with another level.

Either way, this testing procedure allows us to establish a complete **skeleton** of the system or product.

Benefits

The following are the major benefits of benefits of top-down integration testing:

- *Having the skeleton, we can test major functions early in the development process.*
- *At the same time, we can also test any interfaces that we have and thus discover any errors in that area very early on.*

- *The major benefit of this procedure is that we have a partially working model to demonstrate to the clients and the top management. This of course builds everybody's confidence not only in the development team but also in the model itself.*

Drawbacks

There are some drawbacks to this procedure as well:

- *Using stubs does not permit all the necessary upward data flow.*
- *There is simply not enough data in the stubs to feed back to the calling module. As a result, the top-level modules can not be really tested properly and every time the stubs are replaced with the actual modules, the calling modules should be re-tested for integrity again.*

11.18.2.2 *Bottom-Up Strategy*

Bottom-up approach, as the name suggests, is the opposite of the Top-down method.

This process starts with building and testing the low level modules first, working its way up the hierarchy.

Because the modules at the low levels are very specific, we may need to combine several of them into what is sometimes called a cluster or build in order to test them properly.

Then to test these builds, a test driver has to be written and put in place.

Benefits

The advantages of bottom-up integration are:

- *There is no need for program stubs as we start developing and testing with the actual modules.*
- *Starting at the bottom of the hierarchy also means that the critical modules are usually built first and therefore any errors in these modules are discovered early in the process.*

Drawbacks

As with Top-down integration, there are some drawbacks to this procedure:

- *In order to test the modules we have to build the test drivers that are more complex than stubs. And in addition to that they themselves have to be tested. So more effort is required.*
- *No working model can be presented or tested until many modules have been built. This also means that any errors in any of the interfaces are discovered very late in the process.*

11.18.2.3 *Big-Bang Strategy*

Big-Bang approach is very simple in its philosophy where basically all the modules or builds are constructed and tested independently of each other and when they are finished, they are all put together at the same time.

The main advantage of this approach is that it is very quick as no drivers or stubs are needed, thus cutting down on the development time.

However, as with anything that is quickly slapped together, this process usually yields more errors than the other two. Since these errors have to be fixed and take more time to fix than errors at the module level, this method is usually considered the least effective.

Because of the amount of coordination that is required it is also very demanding on the resources.

Another drawback is that there is really nothing to demonstrate until all the modules have been built and integrated.

11.18.2.4 *Sandwiched* Strategy

Sandwiched strategy is the most widely used integration strategy as this strategy aims at overcoming the limitations of both top-down and bottom-up strategies. This strategy is a mixture of both top-down and bottom-up approaches.

This is also known as ***mixed integration strategy.***

11.18.3 Functional Testing

An alternative form of black-box testing is to base the test data on the functionality of the module, which is called functional testing. In functional testing, each function implemented in the module is identified. From this, test data are devised to test each function separately.

Functional testing verifies that an application does what it is supposed to do and doesn't do what it shouldn't do.

For example, if you are functionally testing a word processing application, a partial list of checks you would perform minimally includes creating, saving, editing, spell checking and printing documents.

Function testing usually includes testing of all the interfaces and should therefore involve the clients in the process. Because every aspect of the software system is being tested, the specifications for this test should be very detailed describing who, where, when and how will conduct the tests and what exactly will be tested.

The portion of the testing that will involve the clients is usually conducted as an alpha test where the developers closely monitor how the clients use the system. They take notes on what needs to be improved.

Functional testing can be difficult, however, for the following reasons:

- *The functions within a module may consist of lower-level functions, each of which must be tested first.*
- *Lower-level functions may not be independent.*
- *Functionality may not coincide with module boundaries; this tends to blur the distinction between module testing and integration testing. This problem arises in*

the object-oriented paradigm when an object sends a message to (invokes) a method of a different object.

Functional testing falls in two categories:

- *Positive Functional Testing*
- *Negative Functional Testing*

11.18.3.1 *Positive Functional Testing*

This testing entails exercising the application's functions with valid input and verifying that the outputs are correct.

Example

- Continuing with the word processing example, a positive test for the printing function might be to print a document containing both text and graphics to a printer that is online, filled with paper and for which the correct drivers are installed.

11.18.3.2 *Negative Functional Testing*

This testing involves exercising application functionality using a combination of invalid inputs, unexpected operating conditions and other "out-of-bounds" scenarios.

Example

- Continuing the word processing example, a negative test for the printing function might be to disconnect the printer from the computer while a document is printing.
- What probably should happen in this scenario is a plain-English error message appears, informing the user what happened and instructing him/her on how to remedy the problem.
- What might happen, instead, is the word processing software simply hangs up or crashes because the "abnormal" loss of communications with the printer isn't handled properly.

11.18.4 Regression Testing

Regression testing is the process of running a subset of previously executed integration and function tests to ensure that program changes have not degraded the system. (Pressman 1997; Sommerville 1998) The regressive phase concerns the effect of newly introduced changes on all the previously integrated code. Problems arise when errors made in incorporating new functions affect the previously tested functions, which are common in large systems.

Regression testing may be conducted manually or using automated tools. The basic regression testing approach is to incorporate selected test cases into a regression bucket that is run periodically to find regression problems. In many organizations regression testing consists of running all the functional tests every few months. This generally delays the regression problem detection and results in significant rework after every regression run.

It is wise to accumulate a comprehensive regression bucket and also define a subset of test cases. The full bucket is run occasionally, but the subset is run against every spin. The spin subset should consist of all the test cases for recently integrated functions and a selected sample from the full regression bucket. Depending on the success history of test subsets and the complexity of the program the test cases are selected from the subset.

11.18.5 *Systems and Acceptance Testing*

Final stage of the testing process should be System Testing and Acceptance Testing. It is concerned with the execution of test cases to evaluate the whole system with respect to the user's requirements. A *system test* checks for unexpected interactions between the units and modules, and also evaluates the system for compliance with functional requirements.

An *acceptance test* is the process of executing the test cases agreed with the customer as being an adequate representation of user requirements. These are often called *Black Box* or *Functional tests*.

These terms make reference to the tests being unconcerned with the internal structure of the code. They are concentrated on analysing the performance of the code with respect to the test suite.

At any stage in the software life-cycle errors may be discovered. This may lead to changes in design and/or code update resulting in a re-application of any of unit, integration, system or acceptance tests.

The process of re-testing a unit during its development is called a *Revision test*. This is similar to a *Regression test* that occurs during maintenance when a system is being modified. Regression testing is the selective re-testing of a system or unit to verify that modifications have not caused unintended side-effects and that the system or unit still complies with the current specification.

This type of test involves examination of the whole computer system: *all the software components, all the hardware components and any interfaces.*

The whole computer based system is checked not only for validity but also for meeting objectives. It should include the following types testing:

- *Recovery testing*
- *Security testing*
- *Performance testing*
- *Reliability testing*
  - *Robustness Testing*
  - *Stress Testing*
  - *Load Testing*
- *Thread Testing*
- *Back-to-back Testing*
- *Seeding Technique*

All the above types of testing are described below.

11.18.5.1 *Recovery Testing*

Recovery testing uses test cases designed to examine how easily and completely the system can recover from a disaster (power shut down, blown circuit, disk crash, interface failure, insufficient memory, etc.). It is desirable to have a system capable of recovering quickly and with minimal human intervention. It should also have a log of activities happening before the crash (these should be part of daily operations) and a log of messages during the failure (if possible) and upon re-start.

11.18.5.2 *Security Testing*

Security testing involves testing the system in order to make sure that unauthorized personnel or other systems cannot gain access to the system and information or resources within it. Programs that check for access to the system via passwords are tested along with any organizational security procedures established.

Software quality, reliability and security are tightly coupled. Flaws in software can be exploited by intruders to open security holes. With the development of the Internet, software security problems are becoming even more severe.

Many critical software applications and services have integrated security measures against malicious attacks. The purpose of security testing of these systems include:

- *Identifying and removing software flaws that may potentially lead to security violations, and*
- *Validating the effectiveness of security measures.*

Simulated security attacks can be performed to find vulnerabilities.

11.18.5.3 *Performance Testing*

Every software system has its implicit performance requirements. The software should not take infinite time or infinite resource to execute. The software system should be free from *Performance Bugs.*

"Performance bugs" sometimes are used to refer to those design problems in software that cause the system performance to degrade.

The goal of performance testing can be performance bottleneck identification, performance comparison and evaluation, etc.

Performance has always been a great concern and a driving force of computer evolution. Performance evaluation of a software system usually includes:

- *Resource usage,*
- *Throughput,*

- *Stimulus-response time and*
- *Queue lengths detailing the average or maximum number of tasks waiting to be serviced by selected resources.*

Typical resources that need to be considered include network bandwidth requirements, CPU cycles, disk space, disk access operations, and memory usage

The typical method of doing performance testing is using a *benchmark*, a program, workload or trace designed to be representative of the typical system usage.

Performance testing involves monitoring and recording the performance levels during regular and low and high stress loads. It tests the amount of resource usage under the just described conditions and serves as basis for making a forecast of additional resources needed (if any) in the future. It is important to note that performance objectives should have been developed during the planning stage and performance testing is to assure that these objectives are being met. However, these tests may be run in initial stages of production to compare the actual usage to the forecasted figures.

11.18.5.4 *Reliability Testing*

Software reliability refers to the probability of failure-free operation of a system. It is related to many aspects of software, including the testing process. Directly estimating software reliability by quantifying its related factors can be difficult.

Testing is an effective sampling method to measure software reliability. Guided by the operational profile, software testing (usually black-box testing) can be used to obtain failure data, and an estimation model can be further used to analyze the data to estimate the present reliability and predict future reliability.

Therefore, based on the estimation, the developers can decide whether to release the software, and the users can decide whether to adopt and use the software.

Risk of using software can also be assessed based on reliability information. Hamlet (Hamlet 1992) advocates that the primary goal of testing should be to measure the dependability of tested software.

There is agreement on the intuitive meaning of dependable software: it does not fail in unexpected or catastrophic ways. The following two are the variances of reliability testing based on this simple criterion:

- *Robustness Testing*
- *Stress Testing*
- *Load Testing*

11.18.5.4.1 *Robustness Testing*

The robustness of a software component is the degree to which it can function correctly in the presence of exceptional inputs or stressful environmental conditions.

Robustness testing differs with correctness testing in the sense that the functional correctness of the software is not of concern. It only watches for robustness problems such as machine crashes, process hangs or abnormal termination. Therefore robustness testing can be made more portable and scalable than correctness testing.

11.18.5.4.2 *Stress Testing*

Stress testing encompasses creating unusual loads on the system in attempts to break it. System is monitored for performance loss and susceptibility to crashing during the load times. If it does crash as a result of high load, it provides for just one more recovery test. *Stress testing* is also known as ***endurance testing***.

Stress testing is often used to test the whole system rather than the software alone. In such tests the software or system are exercised with or beyond the specified limits.

Typical stress includes resource exhaustion, bursts of activities, and sustained high loads.

Stress testing is subjecting a system to an unreasonable load while denying it the resources (e.g., RAM, Disk, MIPS, Interrupts, etc.) needed to process that load. The idea is to stress a system to the breaking point in order to find bugs that will make that break potentially harmful. The system is not expected to process the overload without adequate resources, but to behave (e.g., fail) in a decent manner (e.g., not corrupting or losing data). Bugs and failure modes discovered under stress testing may or may not be repaired depending on the application, the failure mode, consequences, etc. The load (incoming transaction stream) in stress testing is often deliberately distorted so as to force the system into resource depletion.

11.18.5.4.3 *Load Testing*

One of the most common, but unfortunate misuse of terminology is treating "load testing" and "stress testing" as synonymous.

Load testing is subjecting a system to a statistically representative (usually) load. In load testing, load is varied from a minimum (zero) to the maximum level the system can sustain without running out of resources or having, transactions suffer (application-specific) excessive delay.

11.18.5.5 *Thread Testing*

Thread testing is a popular testing technique suitable for testing real-time systems. In this testing, the processing of each external 'threads' its way through the system processes or objects with some processing carried out at each stage (Sommerville 1998).

This type of testing involves identifying and executing each possible processing 'thread'.

11.18.5.6 *Back-to-back testing*

The back-to-back testing is used when several versions of a system exist for testing (Sommerville 1998). All versions are tested with same set of tests and then the results are compared for some system problems, if exists.

Following are some steps that may help carrying out back-to-back testing:

- *A general-purpose set of test cases is prepared.*
- *Using these test cases, run the different system versions and store the results in different files.*
- *Perform the automatic comparison of the results stored in different files and generate the **Difference Report**.*

The Difference Report indicates the system problem, if exists among the different system versions.

11.18.5.7 *Seeding Technique*

Seeding technique is based on a seeding model that attempts to provide an estimate of the number of defects in software (or a program). A program is randomly seeded with a number of known errors. Then the program is tested using the standard testing strategies to detect the overall errors.

This technique is based in the following assumptions:

- *Real and seeded faults have same distribution.*
- *Seeded errors are realistic.*

Seeding technique works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist in the software.

The result are presented in this form:

Found *r* real errors	Found *s* seeded errors
Total *R* real errors	Total *S* seeded errors

Based on these proportions, we can estimate the number of errors in the software.

$$r/R = s/S$$

Thus, we can find out the ***total real defects*** in this way, $R = (S/s) * r$

Thus the ***Remaining defects*** $= R - r = r * (S/s - 1)$

11.19 Defect Testing

Defect testing is aimed at discovering the latent defects before the system is delivered. Defect testing demonstrates *the presence* of program faults *not the absence*. A successful defect test is that causes the system to perform incorrectly and thus exposes a defect.

The testing of a system has two objectives:

- *Firstly, it is intended to show that the system meets its specification;*
- *Secondly, it is intended to exercise the system in such a way that latent defects are exposed.*

These different types of testing are carried out at different phases of the testing process. Final system testing and acceptance testing should be concerned with validation.

Earlier phases in the testing process, namely component and module testing and sub-system testing, should be oriented towards the discovery of defects in the program.

A successful test is therefore a test that discovers a problem in the system. In principle, testing of a program for defects should be exhaustive. In practice, this is impossible in a program that contains loops, as the number of possible path combinations is astronomical.

Defect testing approaches include mainly black-box and white-box testing, which have already been discussed in detail earlier in this chapter.

11.20 Interface Testing

Interface testing is intended to discover defects in the interfaces of objects or modules. Interface defects may arise because of errors made in reading the specification, specification misunderstandings or errors or invalid timing assumptions. Interface testing is particularly useful for object-oriented software development.

This type of testing is sought when modules or sub-systems are integrated to build software systems. Each and every module (or sub-system) has a defined interface that is invoked by other modules.

Unit testing can't detect most interface errors as the errors are a result of the interaction between modules (or components) rather than the modules alone themselves.

Several types of interface errors may occur such as parameter interface errors, shared memory interface errors, message passing interface errors, procedural interface errors, etc.

11.21 Alpha and Beta Testing

Acceptance testing is also sometimes known as ***alpha testing***. Custom-made systems are developed for a single client. The alpha testing process continues until the ***system developer and the client*** agrees that the delivered system is an acceptable implementation of the system requirements (Pressman 1997).

When a system is to be marketed as a software product, a testing process called beta testing is often used.

Beta testing involves delivering a system to a ***number of potential customers*** who agree to use that system. They report ***problems to the system developers***. This exposes the product to real use and detects errors that may not have been anticipated by the system builders.

After this ***feedback, the system is modified*** and either released for further beta testing or for general sale.

11.22 Object Oriented Testing Methods

While the jury is still out on whether "traditional" testing methods and techniques are applicable to OO models, there seems to be a consensus that because the OO paradigm is different from the traditional one, some alteration or expansion of the traditional testing methods is needed. The OO methods may utilize many or just some aspects of the traditional ones but they need to be broadened to sufficiently test the OO products (Pressman 1997).

Because of inheritance and inter-object communications in OO environment, much more emphasis is placed on the analysis and design and their "correctness" and consistency. This is imperative to prevent analysis errors to trickle down to design and development, which would increase the effort to correct the problem.

11.22.1 OO Test Case Design

Conventional test case designs are based on the process they are to test and its inputs and outputs. OO test cases need to concentrate on the states of a class. To examine the different states, the cases have to follow the appropriate sequence of operations in the class. Class, as an encapsulation of attributes and procedures that can be inherited, thus becomes the main target of OO testing (Pressman 1997).

Operations of a class can be tested using the conventional white-box methods and techniques (basis path, loop, data flow) but there is some notion to apply these at the class level instead.

11.22.2 Fault Based Testing

This type of testing allows for designing test cases based on the client specification or the code or both. It tries to identify plausible faults (areas of design or code that may lead to errors). For each of these faults a test case is developed to "flush" the errors out. These tests also force each line of code to be executed (Marick 1994; Pressman 1997).

This testing method does not find all types of errors, however. Incorrect specifications and interface errors can be missed. You may remember that these types of errors can be uncovered by function testing in the traditional model. In OO model, interaction errors can be uncovered by *scenario-based testing*. This form of OO testing can only test against the client's specifications, so interface errors are still missed.

11.22.3 Class Level Methods

As mentioned above, a class (and its operations) is the module most concentrated on in OO environments. From here it should expand to other classes and sets of classes. Just like traditional models are tested by starting at the module first and continuing to module clusters or builds and then the whole program.

11.22.3.1 *Random Testing*

This is one of methods used to exercise a class. It is based on developing a random test sequence that tries the minimum number of operations typical to the behavior of the class.

11.22.3.2 *Partition Testing*

This method categorizes the inputs and outputs of a class in order to test them separately. This minimizes the number of test cases that have to be designed (Pressman 1997).

To determine the different categories to test, partitioning can be broken down as follows:

- *State-based partitioning* – categorizes class operations based on how they change the state of a class
- *Attribute-based partitioning* – categorizes class operations based on attributes they use
- *Category-based partitioning* – categorizes class operations based on the generic function the operations perform

11.22.4 Scenario-based Testing

This form of testing concentrates on what the user does. It basically involves capturing the user actions and then simulating them and similar actions during the test. These tests tend to find interaction type of errors (Marick 1994; Pressman 1997).

11.23 Cleanroom Software Development

Cleanroom software development is an approach to software development that relies on static techniques for program verification and statistical testing for system reliability certification. The 'cleanroom' process is named by analogy with semiconductor fabrication units, where defects are avoided by manufacturing in an ultra-clean atmosphere.

Cleanroom software development has been successful in producing systems that have a high level of reliability. The cleanroom approach to software development is based on the notion that defects in software should be avoided rather than detected and repaired. Instead of unit and module testing, software components are formally specified and mathematically verified as they are developed (Sommerville 1998; Mills 1987).

The cleanroom approach is reportedly no more expensive than conventional development and testing but it results in software with very few errors.

11.23.1 Key Characteristics

Following are the key characteristics of cleanroom development (Sommerville 1998):

- *Incremental Development:* The software is partitioned into increments that are developed separately using the cleanroom process.
- *Formal Specification:* The software to be developed is formally specified.
- *Static Verification:* The developed software is statically verified using mathematically-based correctness arguments.

- *Statistical Testing:* The integrated software increment is tested statistically to determine its reliability.

11.23.2 Cleanroom Development Teams

Cleanroom process is difficult for large software systems. There are several teams involved for this purpose such as (Sommerville 1998):

- *Specification Team*: This team is responsible for developing and maintaining the system specification. Both customer-oriented specifications and internal, mathematical specifications are produced by this team.
- *Development Team*: This team has the responsibility of developing and verifying the software.
- *Certification Team*: This team develops a set of statistical tests to exercise the software after it has been developed. These tests are based on the formal specification so this process can be carried out in parallel with software development.

11.24 Real-Time Systems Testing

The specific characteristic of real-time systems makes them a major challenge when testing. The time-dependent nature of real-time applications adds a new difficult element to testing. Not only does the developer have to look at black and white box testing, but also the timing of the data and the parallelism of the tasks.

In many situation test data for real-time system may produce errors when the system is in one state but to in others. Comprehensive test cases design methods for real-time systems have not evolved yet.

11.25 Automated Software Testing Tools

As testing can be 40% of the all effort expanded on the software development process tools that can assist by reducing the time involved is useful. As a response to this various researchers have produced sets of testing tools.

Miller described various categories for test tools:

Static Analyzers

These program-analysis support "proving" of static allegations-weak statements about program architecture and format.

Code Auditors

These special-purpose filters are used to examine the quality of software to ensure that it meets the minimum coding standards.

Assertion Processors

These systems tell whether the programmer-supplied assertions about the program are actually met.

Test Data Generators

These processors assist the user with selecting the appropriate test data.

Output Comparators

This tool allows us to contrast one set of outputs from a program with another set to determine the difference among them.

Dunn also identified additional categories of automated tools including:

Symbolic Execution Systems

This tool performs program testing using algebraic input, instead of numeric data values.

Environmental Simulators

This tool is a specialized computer-based system that allows the tester to model the external environment of real-time software and simulate operating conditions.

Data Flow Analyzers

This tool tracks the flow of data through the system and tries to identify data related errors.

Rational, Mercury Interactive, Seague, Compuware, Empirix, etc. are some of the major companies that provide automated software testing tools.

11.26 Debugging

Software testing is a process that can be systematically planed and specified. Test design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. When a test case uncovers an error, debugging is the process that results in the removal of the error.

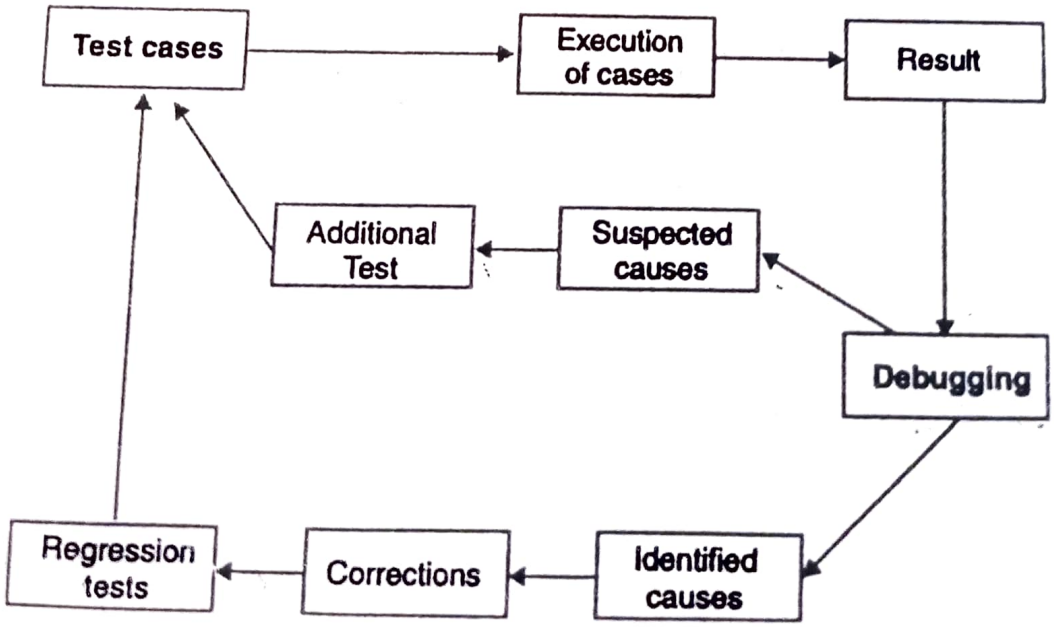

Fig. 11.8: *Debugging Process*

Debugging is not testing but it always occurs as a consequence of testing (Pressman 1997; Mall 1999; Sommerville 1998; Shooman 1986). The debugging process begins with the execution of a test case. Debugging is also treated an art (Bradley 1985). The debugging process is illustrated in **Fig. 11.8.**

The debugging process attempts to match symptom with cause, thereby leading to error correction. There are two outcomes of the debugging:

- *The cause will be found, corrected, and removed,*
- *The cause will not be found.*

11.27 Debugging Techniques

The following are some important debugging techniques:

- *Brute Force Debugging*
- *Backtracking Debugging*
- *Debugging by Induction*
- *Debugging by Deduction*
- *Cause Elimination Debugging*
- *Debugging by Testing*
- *Debugging by Program Slicing*

These approaches are discussed below:

11.27.1 Brute Force Debugging

Brute force debugging methods are applied when all else fails. Brute force debugging is debugging with a storage dump.

Brute force debugging is the most common debugging technique but also the least efficient one. In this technique, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement with error.

People who use thinking rather than a set of helps, exhibit superior performance. This approach becomes more systematic with the use of a symbolic debugger because values of different variables can be easily examined.

11.27.2 Backtracking Debugging

Backtracking is a fairly common debugging technique. This involves backtracking the incorrect results through the logic of the program. In this technique, the source code is traced backwards until the error is discovered.

This technique becomes unfit when the number of source lines to be traced back increases and the number of potential backward paths becomes unmanageable.

11.27.3 Debugging by Induction

This is a process that locates the data, organizes it and devises a hypothesis. This hypothesis must then be proved. A clue structure may be used for what, where, when and to what extent for categories of Is and Is not.

11.27.4 Debugging by Deduction

Deduction may also be used as a debugging technique. Determine the possible causes and use data to eliminate causes. Refine the remaining cause into a hypothesis and prove it.

11.27.5 Cause Elimination Debugging

In this technique, a list of causes that could have contributed to the error symptom is prepared and tests are carried out to eliminate each cause. Software fault tree analysis technique may be used to identify the errors from the error symptom. Software fault tree analysis technique is discussed in detail in Chapter 16.

11.27.6 Debugging by Testing

This thinking type debugging involves using test cases. There are 2 types of test cases.

- Cases that expose a previously undetected error.
- And cases that provide useful information in debugging to locate an error.

For an undetected error cases tend to cover many conditions per test case. But test cases for locating a specific error cover a single condition for each test case.

11.27.7 Debugging by Program Slicing

This debugging technique is similar to backtracking. In this technique, the overall search space is first divided in the program slices so that the search is confined to the program slice only.

A program slice for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

EXERCISES

1. What is the purpose of software testing?
2. When is the role of software testing start in software life cycle? When can planning for software testing start?
3. What are the stages in software testing? Explain each stage in detail.
4. Differentiate between Verification and Validation.
5. Where does software testing fit in Software Quality Assurance (SQA)?
6. Why can't we avoid errors in software development? Explain.