7.5 RELIABILITY CONCEPTS

Define Operational Profile

The operational profile characterizes system (product) usage. Use of this profile is what distinguishes SRE from traditional software development. In order to make a good reliability prediction we must be able to test the product as if it were in the field. Consequently we must define a profile that mirrors field use and then use it to drive testing, in contrast to testing which is driven by the design architecture. The operational profile differs from a traditional functional description in that the elements in the profile are quantified by assignment of a probability of occurrence, and in some cases a criticality factor. Development and test resources are allocated to functions based on these probabilities. Use of the operational profile as a guide for system testing ensures that if testing is terminated, and the software is shipped because of imperative schedule constraints, the most-used (or most critical) operations will have received the most testing and the reliability will be maximized for the given conditions. It facilitates finding earliest the faults that have the biggest impact on reliability. The cost of developing an operational profile varies but is nonlinear with respect to product size. Even simple and approximate operational profiles have been shown to be beneficial. A single product may entail developing several operational profiles depending on the varying modes of operation it contains and the criticality of some operations. Critical operations are designated for increased or accelerated testing.

Plan and Execute Tests

Under SRE the operational profile drives test planning for reliability. Probability and critical factors are used to allocate test cases to ensure that the testing exercises the most important or most frequently used functions first and in proportion to their significance to the system. Testing associated with development and maintenance of a software product fits into two broad categories: structure testing and usage testing. Structure testing is based on the software design while usage testing reflects how the software is actually used. The development team to the extent necessary to verify that the software has been implemented in accordance with the design typically performs unit testing, regression testing, and system testing. Attention to quality in this task facilitates the usage testing that is done later for purposes of determining reliability. SRE focuses on usage testing and often refers to it as reliability growth testing because the purpose is to determine how reliable the software is becoming. Reliability growth testing is coupled with the removal of faults and is typically implemented when the software is fully developed and in the system test phase. Failures identified in testing are referred to the developers for repair. A new version of the software is built and test iteration occurs. The testing scenario includes feature, load, and regression tests. Failure intensity (failures per natural or time unit) is tracked in order to guide the test process, and to determine feasibility of release of the software. The testing procedure executes the test cases in random order but because of the allocation of test cases based on usage probability, test cases associated with events of greatest importance to the customer are likely to occur more often within the random selection. Reliability growth testing presumes that a test scenario will be performed more than once, and that deviations from the requirements are identified as failures. Each failure is recorded along with information that identifies where in the test cycle the failure occurred. Failures should be further classified by severity, and by functional area, and by life cycle phase in which they should have been discovered. In some cases it is more meaningful to track the time to failure, or the inter-fail times than to track total number of failures. The resources available, your philosophy on testing, and the criticality of meeting the reliability goals determine the amount of testing needed. Some experts feel that you can build reliability into your software with a strong testing program while others feel that the amount of testing needed can be reduced significantlyif attention is given up front to ensure complete requirements specification, and that requirements actually drive the design. (We have all experienced situations in which the software is built and then the requirements are defined). Careful review of requirements and subsequent review of design with respect to requirements is tedious but typically finds the most important defects early, allowing corrective action before coding. This process is much more cost effective than waiting for testing to uncover all such errors. Structural testing only tests the software built - it cannot tell you what was not built that should have been.

Use Test Results to Drive Decisions

A normalized reliability growth curve can be used to predict when the software will attain the desired reliability level. It can be used to determine when to stop testing. It can also be used to demonstrate the impact on reliability of a decision to deliver the software on an arbitrary (or early) date. Figure 7.3 illustrates a typical reliability curve. It plots failure intensity over the test interval.

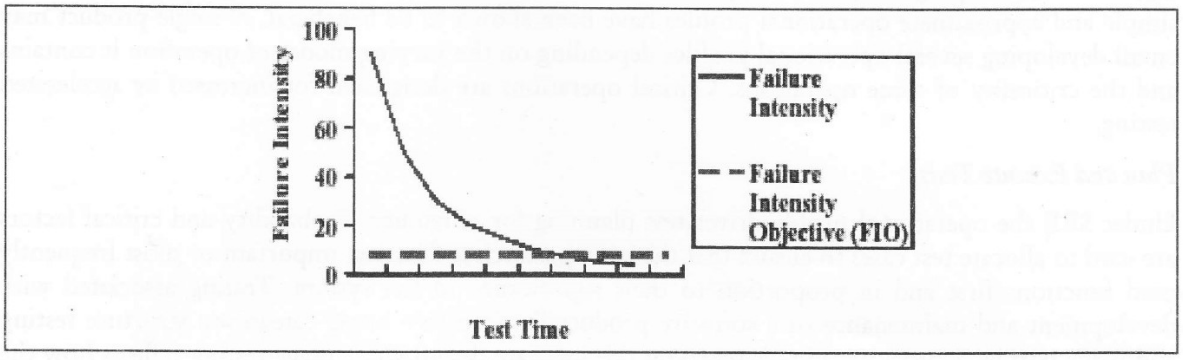

Figure 7.3: Typical Reliability Growth Curve for Software

The failure intensity figures are obtained from tracking failures during testing. Test time represents iterative tests (with test cases selected randomly based on the operational profile). It is assumed that following each test iteration, identified faults are fixed and a new version of the software is used for the next test iteration. Failure intensity drops and the curve approaches the pre-defined Failure Intensity Objective (reliability goal).

Figure 7.4 illustrates what may happen when the process for fixing detected errors is not under control, or a major shift in design has occurred as a result of failures detected.

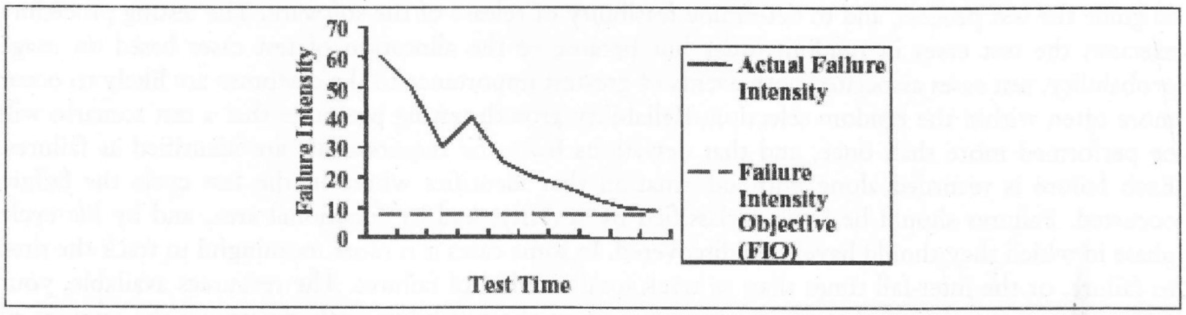

Figure 7.4: Reliability Growth Curve Reflecting Out-of-control Process

Failure intensity drops, spikes, and then makes a gradual decline. Any predictions made prior to the 3rd test iteration would be grossly inaccurate because the spike could not be foreseen. The changes actually introduced new errors while attempting to fix the known errors. This graph actually identifies several potential problems. (1) Significant resources are being used for testing. (2) Testing is spanning a significant amount of time and may delay delivery, or be excessive relative to overall project resources. (3) The process for fixing errors may be inadequate. (4) There may be weak areas in the development process (analysis and design) itself which are the root cause of this erratic reliability profile.

This type of graph is more likely to occur in efforts where the developer prefers to let the testing find the errors rather than design for defect prevention up front. Figure 7.5 overlays the desired outcome (goal) from applying software reliability engineering principles on a typical reliability curve. The goal is to converge quickly to the desired reliability goal thus reducing the rework and the time and resources required for testing and enabling a shorter time to delivery without sacrificing reliability.

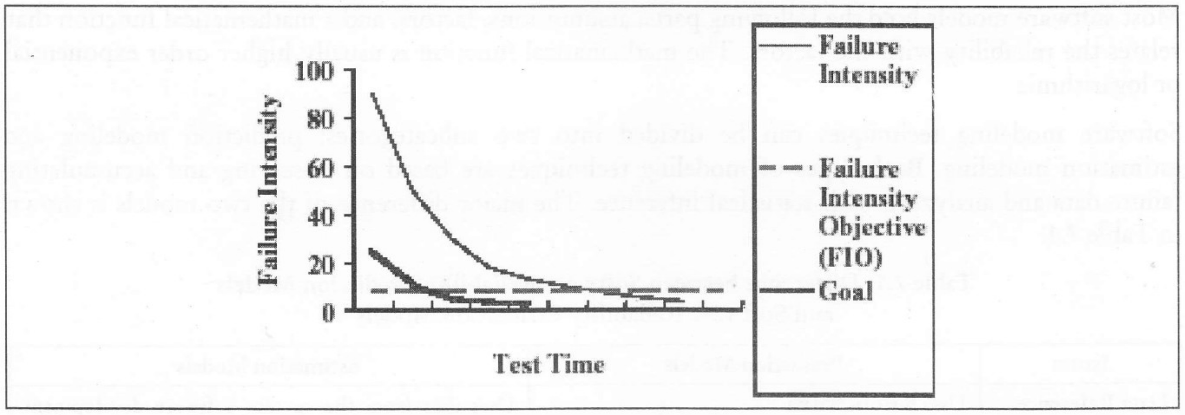

Figure 7.5: Reliability Growth Curve: Desired vs. Typical

This graph suggests that if the goal was achieved the software could be ready for release and testing could stop at least four versions (test iterations) earlier.

AT&T combined the operational profile with other quality improvement initiatives on a PBX switching system. This resulted in customer reported problems and maintenance costs being reduced by a factor of 10, system-test interval by a factor of 2, and product-introduction interval by 30 percent. Hewlett-Packard's application of SRE practices to their system test process for multiprocessor operating systems resulted in a 50% reduction in the system-test time and cost. SRE is a specialized subset of software quality assurance and as such is aligned with the high level programs for process improvement such as SEI's Capability Maturity Model (CMM), and requirements for ISO 9000 certification. An organization which is already traveling one of these paths has a head start on SRE.

In short, the key to success with SRE is to start simple. Stay focused on these essential elements and adds more sophistication only after you are comfortable implementing these basic principles. Remember the goal is to build and deliver software that your customer perceives as reliable. The reliability engineering practices described here are simply tools to facilitate your attainment of that goal. In order to obtain meaningful reliability estimates you must be able to quantify and measure the software behavior with respect to the desired reliability objectives. You must test and testing should be driven by a usage profile. If there are not enough resources to test until the reliability goals are met you will at least know where you are and hopefully will be able to apply lessons learned to your next effort. This article provides only a cursory view of SRE noting the key factors of each topic.

7.6 RELIABILITY MODELS

A propagation of software reliability models have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to measure software reliability still remains largely unsolved. As many models as there are and many more emerging, none of the models can detain a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models hold the following parts: assumptions, factors, and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

Software modeling techniques can be divided into two subcategories: prediction modeling and estimation modeling. Both types of modeling techniques are based on observing and accumulating failure data and analyzing with statistical inference. The major difference of the two models is shown in Table 7.1.

Table 7.1: Difference between Software Reliability Prediction Models and Software Reliability Estimation Models

Issues	Prediction Models	Estimation Models
Data Reference	Uses historical data	Uses data from the current software development effort
When Used in Development Cycle	Usually made prior to development or test phases; can be used as early as concept phase	Usually made later in life cycle(after some data have been collected); not typically used in concept or development phases
Time Frame	Predict reliability at some future time	Estimate reliability at either present or some future time

Representative prediction models comprises of Musa's Execution Time Model, Putnam's Model. and Rome Laboratory models TR-92-51 and TR-92-15, etc. Using prediction models, software reliability can be predicted early in the development phase and enhancements can be initiated to improve the reliability.

Representative estimation models contain exponential distribution models, Weibull distribution model, Thompson and Chelson's model, etc. Exponential models and Weibull distribution model are usually named as classical fault count/fault rate estimation models, while Thompson and Chelson's model belong to Bayesian fault rate estimation models.

The field has developed to the point that software models can be applied in practical situations and give meaningful results and, second, that there is no one model that is best in all situations. Because of the complexity of software, any model has to have extra assumptions. Only limited factors can be put into consideration. Most software reliability models ignore the software development process and focus on the results -- the observed faults and/or failures. By doing so, complexity is reduced and abstraction is achieved, however, the models tend to specialize to be applied to only a portion of the situations and a certain class of the problems. We have to carefully choose the right model that suits our specific case. Furthermore, the modeling results can not be blindly believed and applied.

7.6.1 Reliability Techniques

Reliability techniques can be divided into two categories: Trending and Predictive.

1. ***Trending reliability*** tracks the failure data produced by the software system to develop a reliability operational profile of the system over a specified time.
2. ***Predictive reliability*** assigns probabilities to the operational profile of a software system; for example, the system has a 5 percent chance of failure over the next 60 operational hours.

In practice, reliability trending is more appropriate for software, whereas predictive reliability is more suitable for hardware. Trending reliability can be further classified into four categories: Error Seeding, Failure Rate, Curve Fitting, and Reliability Growth.

1. ***Error Seeding:*** Estimates the number of errors in a program by using multistage sampling. Errors are divided into indigenous and induced (seeded) errors. The unknown number of indigenous errors is estimated from the number of induced errors and the ratio of errors obtained from debugging data.
2. ***Failure Rate:*** Is used to study the program failure rate per fault at the failure intervals. As the number of remaining faults change, the failure rate of the program changes accordingly.
3. ***Curve Fitting:*** Uses statistical regression analysis to study the relationship between software complexity and the number of faults in a program, as well as the number of changes, or failure rate.
4. ***Reliability Growth:*** Measures and predicts the improvement of reliability programs through the testing process. Reliability growth also represents the reliability or failure rate of a system as a function of time or the number of test cases.

7.7 RELIABILITY ALLOCATION

Reliability Allocation deals with the setting of reliability goals for individual subsystems such that a specified reliability goal is met and the hardware and software subsystem goals are well balanced among themselves. Well-balanced usually refers to approximate relative equality of development time, difficulty, risk, or to the minimization of overall development cost. In the process of developing a new product, the engineer is often faced with the task of designing a system that conforms to a set of reliability specifications. The engineer is given the goal for the system and must then develop a design that will achieve the desired reliability of the system, while performing all of the system's intended functions at a minimum cost. This involves a balancing act of determining how to allocate reliability to the components in the system so the system will meet its reliability goal while at the same time ensuring that the system meets all of the other associated performance specifications.

The simplest method for allocating reliability is to distribute the reliabilities uniformly among all components. For example, suppose a system with five components in series has a reliability objective of 90% for a given operating time. The uniform allocation of the objective to all components would require each component to have a reliability of 98% for the specified operating time. While this manner of allocation is easy to calculate, it is generally not the best way to allocate reliability for a system. The optimum method of allocating reliability would take into account the cost or relative difficulty of improving the reliability of different subsystems or components.

The reliability optimization process begins with the development of a model that represents the entire system. This is accomplished with the construction of a system reliability block diagram that represents the reliability relationships of the components in the system. From this model, the system reliability impact of different component modifications can be estimated and considered alongside the costs that would be incurred in the process of making those modifications. It is then possible to perform an optimization analysis for this problem, finding the best combination of component reliability improvements that meet or exceed the performance goals at the lowest cost.

7.7.1 System Reliability Allocation

Reliability allocations for hardware/software systems can be started as soon as the system reliability models have been created. The initial values allocated to the system itself should either be the specified values for the various reliability metrics for the system, or a set of reliability values which are marginally more difficult to achieve than the specified values. Reliability values that are slightly more aggressive than the required values are sometimes allocated to the system to allow for later system functionality growth and to allow those parts of the system which cannot achieve their allocated values to be given some additional reliability margin later in the design process. The apportionment of reliability values between the various subsystems and elements can be made on the basis of complexity, criticality, estimated achievable reliability, or any other factors considered appropriate by the analyst making the allocation. The procedures provided for allocation of software failure rates can be applied to both hardware and system elements provided the user recognizes that these elements typically operate concurrently. System-level allocations are successively decomposed using the reliability model(s) until an appropriate set of reliability measures has been apportioned to each hardware and hardware/software component of the system.

7.7.2 Hardware Reliability Allocation

The allocation of reliability values to lower-tiered hardware elements is a continuation of the allocation process begun at the system level. The system level hardware reliability models are used to successively apportion the required reliability measures among the various individual pieces of hardware and from the hardware equipment level to the various internal elements. For existing hardware items, the reliability allocations used should be based on the reliability performance of previously produced equipment. Reliability allocations to internal elements of existing hardware are not typically performed. Hardware equipment level allocations are further allocated to various internal elements within the equipment.

7.7.3 Software Reliability Allocation

The first step in the allocation process is to describe the system configuration (system reliability model). Next, trial component reliability allocations are selected, using best engineering judgment. Compute system reliability for this set of component reliability values. Compare the result against the goal. Adjust component reliability values to move system reliability toward the goal, and component reliability values toward better balance. Repeat the process until the desired goal and good balance are achieved. The allocation of a system requirement to software elements makes sense only at the software system or CSCI level. Once software CSCIs have been allocated reliability requirements, a different approach is needed to allocate the software CSCI requirements to lower levels.

Check Your Progress

1. Software reliability is defined as the probability of failure-free software operation. (True/False)
2. In Hardware reliability, material deterioration cannot cause failure even when the system is not in use. (True/False)
3. Two models of software reliability are prediction and .... models.

7.8 LET US SUM UP

Software reliability model extends measurements, enabling collected data to be extrapolated into projected failure rates and reliability predictions. Software Reliability Engineering (SRE) is defined as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. Software systems typically contain design and code defects that manifest themselves as software failures at various points during program execution.

Reliability concepts consist of Define Operational Profile, Plan and Execute, Tests Use Test Results to Drive Decisions and SRE cost effective. Reliability allocations for hardware/software systems can be started as soon as the system reliability models have been created.

7.9 KEYWORDS

SRE: Software Reliability Engineering

MTBF: Mean Time Between Failures

CMM: Capability Maturity Model

7.10 QUESTIONS FOR DISCUSSION

1. Explain software and hardware reliability with the example of bathtub curve.
2. What is the difference between operational profile and plan & execute concept of software reliability?
3. Discuss the prediction and estimation models.
4. Explain three categories of software allocation.

Check Your Progress: Model Answers

1. True
2. False
3. Estimation

7.11 SUGGESTED READINGS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Rajib Mall, *Fundamentals of Software Engineering*, PHI, 2nd Edition.

Sommerville, *Software Engineering*, Pearson Education, 6th Edition.

Boehm B, *Software Engineering Economics*, Prentice Hall, 1981.

LESSON

8

SOFTWARE TESTING

CONTENTS

8.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Explain software testing, how to improve software quality, software factors
- State testing process

- Define functional testing and structural testing
- Identity test activities and testing tools
- Describe debugging approaches and processes

8.1 INTRODUCTION

Dear students it should be clear in mind that the philosophy behind testing is to find errors. Test cases are devised with this purpose in mind. A test case is a set of data that the system will process as normal input. However, the data with the express intent of determining whether the system will process then correctly. For example, test cases for inventory handling should include situation in which the quantities to be withdrawn from inventory exceed, equal, and are less than the actual quantities on hand. Each test case is designed with the intent of finding errors in the way the system will process it. There are two general strategies for testing software: code testing and specifications testing. In code testing, the analyst develops that case to execute every instructions and path in a program. Under specification testing, the analyst examines the program specifications and then writes test data to determine how the program operates under specific conditions. Regardless of which strategy the analyst follows, there are preferred practices to ensure that the testing is useful. The levels of tests and types of test data, combined with testing libraries, are important aspects of the actual test process.

8.2 SOFTWARE TESTING

Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear -- generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects – or bugs – will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable – and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out.

Discovering the design defects in software is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^64 distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and unpredictable environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it didn't work for previously. But its behavior on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive.

Regardless of the limitations, testing is an integral part in software development. It is broadly deployed in every phase in the software development cycle. Typically, more than 50% percent of the development time is spent in testing. Testing is usually performed for the following purposes:

8.2.1 To Improve Quality

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. Bugs can cause disasters. The so-called year 2000 (Y2K) bug has given birth to a cottage industry of consultants and programming tools dedicated to making sure the modern world doesn't come to a screeching halt on the first day of the next century. In a computerized embedded world, the quality and reliability of software is a matter of life and death.

Quality means the conformance to the specified design requirement. 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.

8.2.2 For Verification and Validation (V&V)

Just as topic Verification and Validation indicated, another important purpose of 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. We can also compare the quality among different products under the same specification, based on results from the same test.

We can not test quality directly, but we can test related factors to make quality visible. 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. The table below illustrates some of the most frequently cited quality considerations.

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 factor varies from application to application. Any system where human lives are at stake must place extreme emphasis on reliability and integrity. In the typical business system usability and maintainability are the key factors, while for a one-time scientific program neither may be significant. Our testing, to be fully effective, must be geared to measuring each 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 specified test cases. A finite number of tests can not validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software 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.

8.2.3 History of Software Testing

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug, it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dr. Dave Gelperin and Dr. William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented
- 1957-1978 - Demonstration oriented
- 1979-1982 - Destruction oriented
- 1983-1987 - Evaluation oriented
- 1988-2000 - Prevention oriented

Definition

Testing is not a technique for building quality systems; rather, it is a technique that is used because we recognize that we have failed to build a fault free system. Testing assists in its repair by identifying where the program fails to meet its specification. I will broaden the usual definition of testing that typically refers only to the testing of code. The expanded definition includes the testing of the many types of models, requirements, analysis, architectural and detailed design, that are constructed in the early development phases. These models may be represented in an executable format in a CASE tool such as BetterStateÂ or they may require a form of symbolic execution or inspection. My justification for classifying these activities as testing is that the inputs to one of these executions will be much more specific than the inputs to a typical review process (more about this below).

Using this expanded definition, there are a number of products of the software development process that should be tested including:

- Requirements models
- Analysis and design models
- Architectural models

- Individual components
- Integrated system code

In fact, there should be a testing activity associated with each step in the development process. Adding the testing of models will find some faults earlier in the development process resulting in cheaper repair costs.

Although these products most often will be tested to determine their correctness, testing may be used to investigate a variety of attributes including:

- Performance
- Usability
- Robustness
- Reusability
- Extensibility
- Flexibility.

The testing perspective is an attitude that questions the validity of every product and utilizes a thorough search of the product to identify faults. Systematic algorithms and intuitive insights guide this search. It is this systematic search that makes testing a more powerful tool than reviews and inspections. A review will almost never find something that isn't there. That is, a review typically only seeks to validate what is in the model and does not systematically search to determine if all the entities and relationships that should be there are. The testing perspective requires that a piece of software demonstrate that it is performing as its complete specification indicates that it should (and that there is no extra behavior). A product is tested to determine that it will do what it is supposed to do (a positive test). It should also be tested to ensure that it does not do what it is not supposed to do (a negative test). This leads to the need to define the parameters of the search.

Test cases are created as part of the testing process to direct the search. A test case presents a context in which the software is to operate and a description of the behavior expected from the software within that context. In most traditional testing situations the context is an operational one in which the software is executed to determine its actual behavior. If the product under test is a model rather than actual code, the test case may contain a textual description of the context as well as some specific input values.

The testing perspective may be adopted by the same person who developed the product under test or by another person who brings an independent view of the specification and the product. Developer-led testing is efficient since there is no learning curve on the product under test, but it can be less effective because the developer may not search as exhaustively as the independent tester. The completeness and clarity of the specification becomes very important when a second person becomes involved in the testing. If a tester is not given a specification, then any test result is correct!

8.2.4 Problems in Testing

1. Software Testing is a critical element of Software Quality Assurance.
2. It represents the ultimate review of the requirements, specification, the design and the code.
3. It is the most widely used method to ensure Software Quality.

4. For most software, exhaustive testing is not possible.
5. Many organizations spend 40-50% of development time in testing.
6. Testing is the most expensive way to remove defects during software development.
7. Problems in testing are one of the major contributors to cost and scheduled overruns.

8.2.5 Testing Objectives

Which ones should we test?

One of the most intense arguments in testing object-oriented systems is whether detailed component testing is worth the effort. That leads me to state an obvious (at least in my mind) axiom: Select a component for testing when the penalty for the component not working is greater than the effort required to test it. Not every class will be sufficiently large, important or complex to meet this test so not every class will be tested independently.

There are several situations in which the individual classes should be tested regardless of their size or complexity:

- *Reusable components:* Components intended for reuse should be tested over a wider range of values than a component intended for a single focused use.
- *Domain components:* Components that represent significant domain concepts should be tested both for correctness and for the faithfulness of the representation.
- *Commercial components:* Components that will be sold, as individual products should be tested not only as reusable components but also as potential sources of liability.

8.2.6 Scope of Software Testing

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the quality aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

8.3 TESTING PROCESS

A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process).

Testing can be done on the following levels:

Unit testing tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

Integration testing exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System testing tests a completely integrated system to verify that it meets its requirements. System integration testing verifies that a system is integrated to any external or third party systems defined in the system requirements.

Before shipping the final version of software, alpha and beta testing are often done additionally:

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Finally, acceptance testing can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Regression Testing

After modifying software, either for a change in functionality or to fix defects, a regression test re-runs previously passing tests on the modified software to ensure that the modifications haven't unintentionally caused a regression of previous functionality. Regression testing can be performed at any or all of the above test levels. These regression tests are often automated.

More specific forms of regression testing are known as sanity testing, when quickly checking for bizarre behavior, and smoke testing when testing for basic functionality.

Benchmarks may be employed during regression testing to ensure that the performance of the newly modified software will be at least as acceptable as the earlier version or, in the case of code optimization, that some real improvement has been achieved.

Finding Faults

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in requirements is found only post-release, then it would cost 10-100 times more to fix it comparing to the cost if the same fault was already found by the requirements review.

Time Introduced	Time Detected				
	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	-	-	1	10	10-25

8.4 FUNCTIONAL TESTING

Software testing methods are traditionally divided into black box testing and white box testing. Functional testing is also known as black box testing. Black box testing treats the software as a black-box without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing, specification based testing, etc. Black-box test design treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements. Synonyms for black-box include: behavioral, functional, opaque-box, and closed-box.

Specification Based Testing

Specification Based Testing aims to test the functionality according to the requirements. Thus, the tester inputs data and only sees the output from the test object. This level of testing usually requires thorough test cases to be provided to the tester who then can simply verify that for a given input, the output value (or behavior), is the same as the expected value specified in the test case. Specification based testing is necessary but insufficient to guard against certain risks.

Non-functional Software Testing

Special methods exist to test non-functional aspects of software.

Performance testing checks to see if the software can handle large quantities of data or users. This is generally referred to as software scalability.

Usability testing is needed to check if the user interface is easy to use and understand.

Security testing is essential for software which processes confidential data and to prevent system intrusion by hackers.

Internationalization and localization is needed to test these aspects of software, for which a pseudo-localization method can be used.

8.5 STRUCTURAL TESTING

Structural Testing is also known as the white box testing. White box testing, by contrast to black box testing, is when the tester has access to the internal data structures and algorithms (and the code that implement these). While black-box and white-box are terms that are still in popular use, many people prefer the terms "behavioral" and "structural". Behavioral test design is slightly different from black-box test design because the use of internal knowledge isn't strictly forbidden, but it's still discouraged. In practice, it hasn't proven useful to use a single test design method. One has to use a mixture of different methods so that they aren't hindered by the limitations of a particular one. Some call this "gray-box" or "translucent-box" test design, but others wish we'd stop talking about boxes altogether. 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. Synonyms for white-box include: structural, glass-box and clear-box.

Types of White Box Testing

The following types of white box testing exist:

(a) Code coverage - creating tests to satisfy some criteria of code coverage. For example, the test designer can create tests to cause all statements in the program to be executed at least once.

(b) Mutation testing methods.

(c) Fault injection methods.

(d) Static testing - White box testing includes all static testing.

Code Completeness Evaluation

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are: function coverage, which reports on functions executed and statement coverage, which reports on the number of lines executed to complete the test.

They both return coverage metric, measured as a percentage.

Grey Box Testing

In recent years the term grey box testing has come into common usage. This involves having access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level.

Manipulating input data and formatting output do not qualify as grey-box because the input and output are clearly outside of the black-box we are calling the software under test. This is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

8.6 TEST ACTIVITIES

- *Compatibility Test:* Test to ensure compatibility of an application or Web site with different browsers, OS, and hardware platforms. Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

- *Conformance Test:* Verifying implementation conformance to industry standards. Producing tests for the behavior of an implementation to be sure it provides the portability, interoperability, and/or compatibility a standard defines.
- *Functional Test:* Validating an application or Web site conforms to its specifications and correctly performs all its required functions. This entails a series of tests, which perform a feature-by-feature validation of behavior, using a wide range of normal and erroneous input data. This can involve testing of the product's user interface, APIs, database management, security, installation, networking; etc testing can be performed on an automated or manual basis using black box or white box methodologies.
- *Load Test:* Load testing is a generic term covering Performance Testing and Stress Testing.
- *Performance Test:* Performance testing can be applied to understand your application or WWW site's scalability, or to benchmark the performance in an environment of third party products such as servers and middleware for potential purchase. This sort of testing is particularly useful to identify performance bottlenecks in high use applications. Performance testing generally involves an automated test suite as this allows easy simulation of a variety of normal, peak, and exceptional load conditions.
- *Regression Test:* Similar in scope to a functional test, a regression test allows a consistent, repeatable validation of each new release of a product or Web site. Such testing ensures reported product defects have been corrected for each new release and that no new quality problems were introduced in the maintenance process. Though regression testing can be performed manually an automated test suite is often used to reduce the time and resources needed to perform the required testing.
- *Smoke Test:* A quick-and-dirty test that the major functions of a piece of software work without bothering with finer details. Originated in the hardware testing practice of turning on a new piece of hardware for the first time and considering it a success if it does not catch on fire.
- *Stress Test:* Test conducted to evaluate a system or component at or beyond the limits of its specified requirements to determine the load under which it fails and how. A graceful degradation under load leading to non-catastrophic failure is the desired result. Often Stress Testing is performed using the same process as Performance Testing but employing a very high level of simulated load.

8.6.1 Levels of Testing

Unit Testing

In unit testing, the analyst tests the program making up a system. For this reason, unit testing is sometimes called program testing. Unit testing gives stress on the modules independently of one another, to find errors. This helps the tester in detecting errors in coding and logic that are contained within that module alone. The errors resulting from the interaction between modules are initially avoided. For example, a hotel information system consists of modules to handle reservations; guest check in and checkout; restaurant, room service and miscellaneous charges; convention activities; and accounts receivable billing. For each, it provides the ability to enter, modify or retrieve data and respond to different types of enquiries or print reports. The test cases needed for unit testing should exercise each condition and option.

Unit testing can be performed from the bottom up, starting with smallest and lowest level modules and proceedings one at a time. For each module in bottom-up testing a short program is used to execute the modules and provided the needed data, so that the modules is asked to perform the way it will when embedded within the larger system.

System Testing

The important and essential part of the system development phase, after designing and developing the software is system testing. We cannot say that every program or system design is perfect and because of lack of communication between the user and designer, some error is there in the software development. The number and nature of errors in a newly designed system depend on some usual factors like communication between the user and the designer; the programmer's ability to generate a code that reflects exactly the systems specifications and the time frame for the design.

Theoretically, a newly designed system should have all the parts or sub-systems are in working order, but in reality, each sub-system works independently. This is the time to gather all the sub-system into one pool and test the whole system to determine whether it meets the user requirements. This is the last change to detect and correct errors before the system is installed for user acceptance testing. The purpose of system testing is to consider all likely variations to which it will be subjected and then push the system to its limits.

Testing is an important function of the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully activated. Another reason for system testing is its utility as a user-oriented vehicle before implementation.

System Testing consists of the following five steps:

- *Program Testing:* A program represents the logical elements of a system. For a program to run satisfactorily, it must compile and test data correctly and tie in properly with other programs. It is the responsibility of a programmer to have an error free program. At the time of testing the system, there exist two types of errors that should be checked. These errors are syntax and logical. A syntax error is a program statement that violates one or more rules of the language in which it is written. An improperly defined field dimension or omitted key words are common syntactical errors. A logical error, on the other hand, deals with incorrect data fields out of range items, and invalid combinations. Compiler like syntax errors does not detect them; the programmer must examine the output carefully to detect them.
- *String Testing:* Programs are invariably related to one another and interact in a total system. Each program is tested to see whether it conforms to related programs in the system. Each part of the system is tested against the entire module with both test and live data before the whole system is ready to be tested.
- *System Testing:* It is deigned to uncover weaknesses that were not found in earlier tests. This includes forced system failure and validation of total system, as its user in the operational environment will implement it. Under this testing, we take low volumes of transactions based on live data. This volume is increased until the maximum level for each transaction type is reached.
- *System Documentation:* All design and test documentation should be well prepared and kept in the library for future reference. The library is the central location for maintenance of the new system.

- *User Acceptance Testing:* Acceptance test has the objective of selling the user on the validity and reliability of the system. It verifies that the system's procedures operate to system specifications and that the integrity of important data is maintained. Performance of an acceptance test is actually the user's show. User motivation is very important for the successful performance of the system.

8.7 DEBUGGING

The goal of testing is to identify errors in the program. The process of testing gives symptoms of the presence of an error. After getting the symptom, we begin to investigate to localize the error i.e. to find out the module causing the error. This section of code is then studied to find the cause of the problem. This process is called debugging. Hence, debugging is the activity of locating and correcting errors. It starts once a failure is detected.

8.7.1 Debugging Techniques

Although developers learn a lot about debugging from their experiences, but these are applied in a trial and error manner. Debugging is not an easy process as per human psychology as error removal requires the acceptance of error and an open mind willing to admit the error.

However, Pressman explains certain characteristics of bugs that can be useful for developing a systematic approach towards debugging. These are:

(a) The symptoms and cause may be geographically remote. That is, the symptom may appear in one part of the program, while the cause may actually be located in other part. Highly coupled programs can worsen this situation.

(b) The symptom may disappear (temporarily) when another error is corrected.

(c) The symptom may actually be caused by non errors (e.g. round off inaccuracies).

(d) The symptom may be caused by a human error that is not easily traced.

(e) The symptom may be result of timing problems rather than processing problems.

(f) It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).

(g) The symptom may be intermittent. This is particularly common in embedded systems that couple hardware with software inextricably.

(h) The symptom may be due to causes that are distributed across a number of tasks running on different processors.

8.7.2 Debugging Approaches

The debugging approach can be categorized into various categories. The first one is trial and error. The debugger looks at the symptoms of the errors and tries to figure out that from exactly which part of the code the error originated. Once found the cause, the developer fixes it. However, this approach is very slow and a lot of time and effort goes waste.

The other approach is called backtracking. Backtracking means to examine the error symptoms to see where they were observed first. One then backtracks in the program flow of control to a point where these symptoms have disappeared. This process identifies the range of the program code which might contain the cause of errors. Another variant of backtracking is forward tracking, where print statements or other means are used to examine a sequence of intermediate results to determine the point at which the result first becomes wrong.

The third approach is to insert watch points (output statements) at the appropriate places in the program. This can be done using software without manually modifying the code.

The fourth approach is more general and called induction and deduction. The induction approach comes from the formulation of a single working hypothesis based on the data, analysis of the existing data and on especially collected data to prove or disprove the working hypothesis. The inductive approach is explained in Figure 8.1.

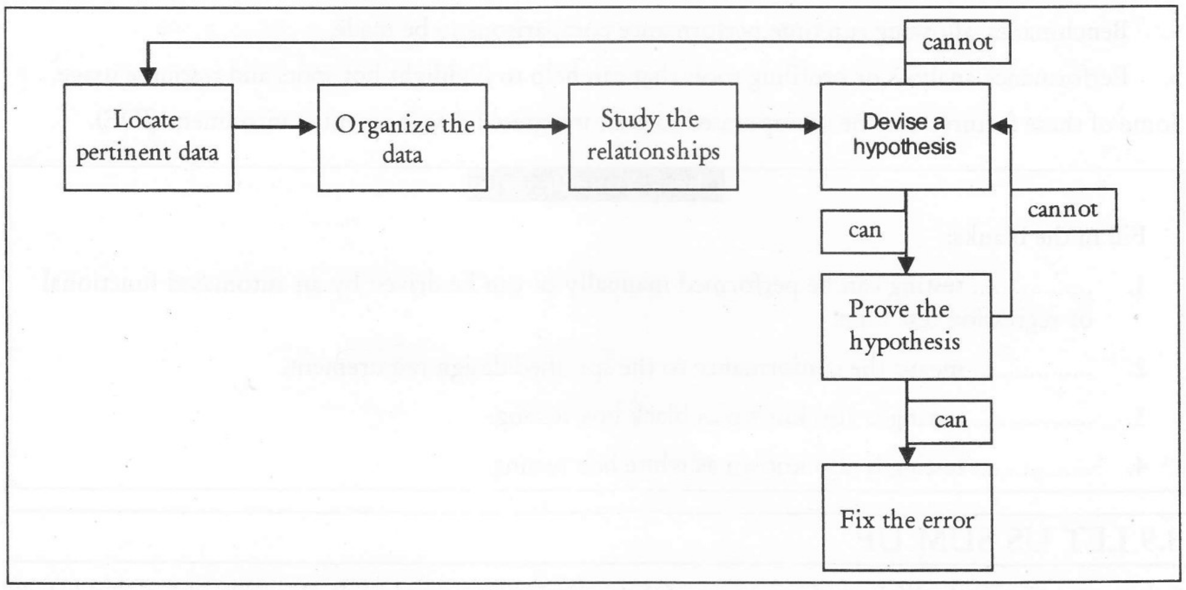

Figure 8.1: The Inductive Debugging Approach

The deduction approach begins by enumerating all causes or hypothesis, which seem possible. Then, one by one, particular causes are ruled out until a single one remains for validation. This is represented in Figure 8.2.

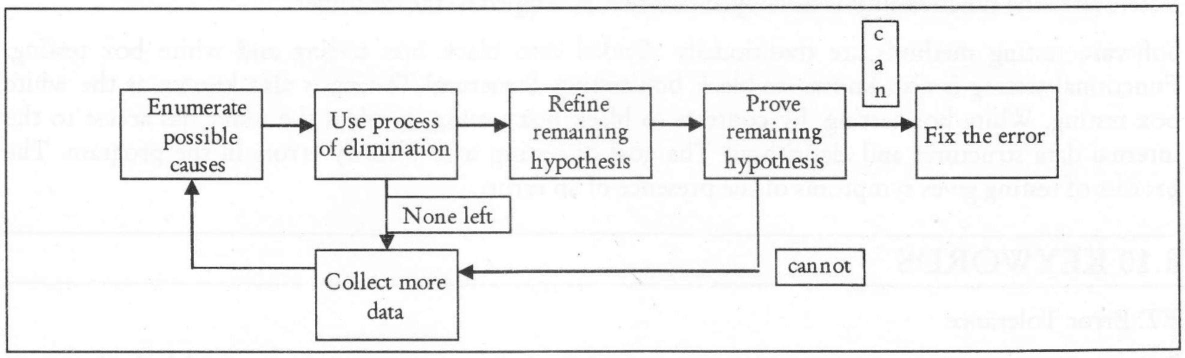
c a n
Enumerate possible causes
Use process of elimination
Refine remaining hypothesis
Prove remaining hypothesis
Fix the error
None left
Collect more data
cannot

Figure 8.2: The Deductive Debugging Approach

8.8 TESTING TOOLS

Program testing and fault detection can be aided significantly by Testing tools and debuggers. Types of testing/debug tools include features such as:-

1. Program monitors, permitting full or partial monitoring of program code including: Instruction Set Simulator, permitting complete instruction level monitoring and trace facilities
2. Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code or code coverage reports.
3. Formatted dump or Symbolic debugging, tools allowing inspection of program variables on error or at chosen points.
4. Benchmarks, allowing run-time performance comparisons to be made.
5. Performance analysis or profiling tools that can help to highlight hot spots and resource usage.

Some of these features may be incorporated into an integrated development environment (IDE).

Check Your Progress

Fill in the blanks:

1. .... testing can be performed manually or can be driven by an automated functional or regression test suite.
2. .... means the conformance to the specified design requirement.
3. .... testing is also known as black box testing.
4. .... testing is also known as white box testing.

8.9 LET US SUM UP

Software testing is the lifeline of any successful Project Implementation. Right tools and methods help in correctly identifying problems and correcting them before live runs. Software Testing is the process of executing a program or system with the intent of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer.

Software testing methods are traditionally divided into black box testing and white box testing. Functional testing is also known as black box testing. Structural Testing is also known as the white box testing. White box testing, by contrast to black box testing, is when the tester has access to the internal data structures and algorithms. The goal of testing is to identify errors in the program. The process of testing gives symptoms of the presence of an error.

8.10 KEYWORDS

ET: Error Tolerance

Software Testing: Software Testing is the process of executing a program or system with the intent of finding errors.

Debugging: Debugging is the activity of locating and correcting errors. It starts once a failure is detected.

Structural Testing: Structural Testing is also known as the white box testing.

Functional Testing: Functional testing is also known as black box testing.

IDE: Integrated Development Environment

8.11 QUESTIONS FOR DISCUSSION

1. What are the types of Software Engineering?
2. What do you know about various special system tests? Explain briefly.
3. Describe briefly about "Unit Testing".
4. Discuss the various debugging approaches.

Check Your Progress: Model Answers

1. Compatibility
2. Quality
3. Functional
4. Structural

8.12 SUGGESTED READINGS

R.S. Pressman, "Software Engineering a Practioner's Approach", (5th edition) Tata McGraw Hill Higher education.

Rajib Mall "Fundamentals of Software Engineering", PHI, Second Edition.

Sommerville, "Software Engineering", Pearson Education, Sixth edition.

Richard Fairpy. "Software Engineering Concepts", Tata McGraw Hill, 1997.

UNIT V