UNIT -1

PURPOSE OF TESTING:

- Testing consumes atleast half of the time and work required to produce a functional program.
- MYTH: Good programmers write code without bugs. (Its wrong!!!)
- History says that even well written programs still have 1-3 bugs per hundred statements.

Productivity and Quality in software:

- In production of comsumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.
- If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.
- Productivity is measured by the sum of the costs of the material, the rework, and the discarded componenets, and the cost of quality assurance and testing.
- There is a trade off between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.
- Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Where as the manufacturing cost of a software is trivial.
- The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
- For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.

Goals for testing:

- Testing and Test Design are parts of quality assurance should also focus on bug prevention. *A prevented bug is better than a detected and corrected bug.*

- **Phases in a tester's mental life can be categorized into the following 5 phases:**

Phase 0: (Until 1956: Debugging Oriented)

There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.

Phase 1: (1957-1978: Demonstration Oriented)

The purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. *i.e.* The more you test, the more likely you' ill find a bug.

Phase 2: (1979-1982: Destruction Oriented)

The purpose of testing is to show that software doesnt work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.

Phase 3: (1983-1987: Evaluation Oriented)

The purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. The product is released when the confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years of use.)

Phase 4: (1988-2000: Prevention Oriented)

Testability is the factor considered here. One reason is to reduce the labour of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

Test Design:

We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the acutal code.

Testing is'nt everything: There are approaches other than testing to create better software. Methods other than testing include:

0. **Inspection Methods:** Methods like walkthroughs, deskchecking, formal inspections and code reading appear to be as effective as testing but the bugs caught donot completely overlap.
1. **Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.
2. **Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job.
3. **Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.
4. **Development Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

DICHOTOMIES:

- **Testing Versus Debugging:** Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.
- Debugging usually follows testing, but they differ as to goals, methods and most important psychology. The below tab le shows few important differences between testing and debugging.

Testing	**Debugging**
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown intial conditions and the end can not be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intutive leaps, experimentation and freedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

- **Function Versus Structure:** Tests can be designed from a functional or a structural point of view. In **functional testing**, the program or system is treated as a blackbox. It is subjected to inputs, and its outputs are verified for conformance to specified behaviour.

Functional testing takes the user point of view- bother about functionality and features and not the program's implementation. **Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.

- Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tets can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.
- **Designer Versus Tester:** Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person.
- Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.
- **Modularity Versus Efficiency:** A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organised into modular components. Small, independent test cases can be designed to test independent modules.
- **Small Versus Large:** Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.
- **Builder Versus Buyer:** Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.
- The different roles / users in a system include:
    1. **Builder:** Who designs the system and is accountable to the buyer.
    2. **Buyer:** Who pays for the system in the hope of profits from providing services.
    3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
    4. **Tester:** Who is dedicated to the builder's destruction.
    5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints.

MODEL FOR TESTING:

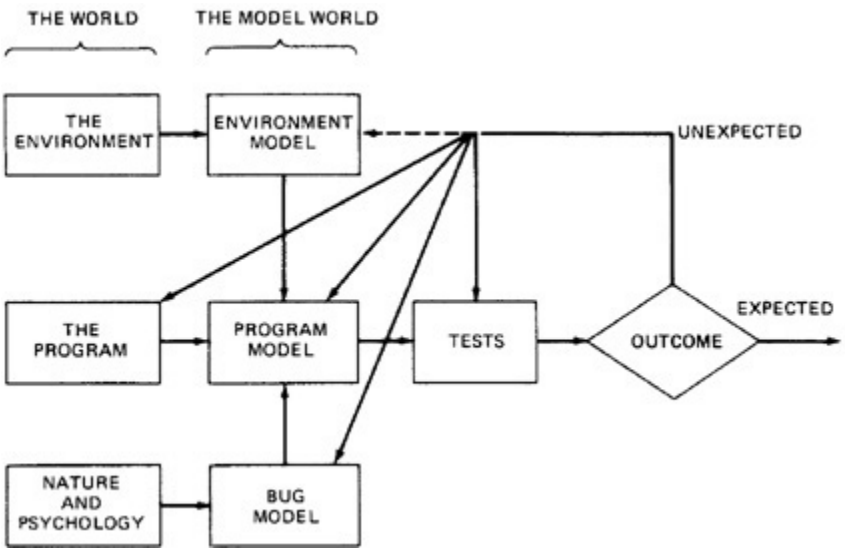


Figure 1.1: A Model for Testing

Above figure is a model of testing process. It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

- **ENVIRONMENT:**
  - A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.
  - The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.
  - Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.
- **PROGRAM:**
  - Most programs are too complicated to understand in detail.
  - The concept of the program is to be simplified inorder to test it.
  - If simple model of the program doesnot explain the unexpected behaviour, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.
- **BUGS:**
  - Bugs are more insidious (deceiving but harmful) than ever we expect them to be.
  - An unexpected test result may lead us to change our notion of what a bug is and our model of bugs.
  - Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.

OPTIMISTIC NOTIONS ABOUT BUGS:

1. **Benign Bug Hypothesis:** The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)

2. **Bug Locality Hypothesis:** The belief that a bug discovered with in a component effects only that component's behaviour.

3. **Control Bug Dominance:** The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.

4. **Code / Data Separation:** The belief that bugs respect the separation of code and data.

5. **Lingua Salvator Est:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
6. **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
7. **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
8. **Sadism Suffices:** The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques.
9. **Angelic Testers:** The belief that testers are better at test design than programmers are at code design.

TESTS:

- Tests are formal procedures, Inputs must be prepared, Outcomes should predicted, tests should be documented, commands need to be executed, and results are to be observed. *All these errors are subjected to error*
- **We do three distinct kinds of testing on a typical software system. They are:**
    1. **Unit / Component Testing:** A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure does not match the intended design structure. A **Component** is an integrated aggregate of one or more units.**Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.
    2. **Integration Testing: Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the componenets were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.
    3. **System Testing:** A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

Role of Models: The art of testing consists of creating , selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behaviour.

PLAYING POOL AND CONSULTING ORACLES

- Testing is like playing a pool game. Either you hit the ball to any pocket (kiddie pool) or you specify the pocket in advance (real pool). So is the testing. There is kiddie testing and real testing. In kiddie testing, the observed outcome will be considered as the expected outcome. In Real testing, the outcome is predicted and documented before the test is run.
- The tester who cannot make that kind of predictions does not understand the program's functional objectives.
- **Oracles:** An oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. Example of oracle : Input/Outcome Oracle - an oracle that specifies the expected outcome for a specified input.
- **Sources of Oracles:** If every test designer had to analyze and predict the expected behaviour for every test case for every component, then test design would be very expensive. The hardest part of test design is predicting the expected outcome, but we often have oracles that reduce the work. They are:

1. **Kiddie Testing:** run the test and see what comes out. If you have the outcome in front of you, and especially if you have the values of the internal variables, then it is much easier to validate that outcome by analysis and show it to be correct than it is to predict what the outcome should be and validate your prediction.
2. **Regression Test Suites:** Today's software development and testing are dominated not by the design of new software but by rework and maintenance of existing software. In such instances, most of the tests you need will have been run on a previous versions. Most of those tests should have the same outcome for the new version. Outcome prediction is therefore needed only for changed parts of components.
3. **Purchased Suits and Oracles:** Highly standardized software that differ only as to implementation often has commercially available test suites and oracles. The most common examples are compilers for standard languages.
4. **Existing Program:** A working, trusted program is an excellent oracle. The typical use is when the program is being rehosted to a new language, operating system, environment, configuration with the intention that the behavior should not change as a result of the rehosting.

IS COMPLETE TESTING POSSIBLE?

- If the objective of the testing were to prove that a program is free of bugs, then testing not only would be practically impossible, but also would be theoretically impossible.
- **Three different approaches can be used to demonstrate that a program is correct.They are:**
    1. **Functional Testing:**
        - Every program operates on a finite number of inputs. A complete functional test would consists of subjecting the program to all possible input streams.
        - For each input the routine either accepts the stream and produces a correct outcome, accepts the stream and produces an incorrect outcome, or rejects the stream and tells us that it did so.
        - For example, a 10 character input string has 280 possible input streams and corresponding outcomes, so complete functional testing in this sense is IMPRACTICAL.
        - But even theoritically, we can't execute a purely functional test this way because we don't know the length of the string to which the system is responding.
    2. **Structural Testing:**
        - The design should have enough tests to ensure that every path through the routine is exercised at least once. Right off that's is impossible because some loops might never terminate.
        - The number of paths through a small routine can be awesome because each loop multiplies the path count by the number of times through the loop.

- A small routine can have millions or billions of paths, so total **Path Testing** is usually IMPRACTICAL.

3. **Formal Proofs of Correctness:**
   - Formal proofs of correctness rely on a combination of functional and structural concepts.
   - Requirements are stated in a formal language (e.g. Mathematics) and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input sequences.
   - The IMPRACTICAL thing here is that such proofs are very expensive and have been applied only to numerical routines or to formal proofs for crucial software such as system's security kernel or portions of compilers.

- Each approach leads to the conclusion that complete testing, in the sense of a proof is neither theoretically nor practically possible.

THEORITICAL BARRIERS OF COMPLETE TESTING:

- "We can never be sure that the specifications are correct"
- "No verification system can verify every correct program"
- "We can never be certain that a verification system is correct"

- Not only all known approaches to absoulte demonstrations of correctness impractical, but they are impossible. Therefore, our objective must shift from a absolute proof to a 'suitably convincing' demonstration.

THE TAXONOMY OF BUGS :

CONSEQUENCES OF BUGS:

- **IMPORTANCE OF BUGS:** The importance of bugs depends on frequency, correction cost, installation cost, and consequences.
    1. **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
    2. **Correction Cost:** What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.
    3. **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.
    4. **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

Importance= ($) = Frequence * (Correction cost + Installation cost + Consequential cost)

CONSEQUENCES OF BUGS: The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:

1. **Mild:** The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.
2. **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
3. **Annoying:** The system's behaviour because of the bug is dehumanizing. *E.g.* Names are truncated orarbitarily modified.
4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM wont give you money. My credit card is declared invalid.
5. **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.
6. **Very Serious:** The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits to withdrawals.
7. **Extreme:** The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.
8. **Intolerable:** Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.

9. **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.
10. **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it doesnot fall in itself ; that erodes the social physical environment; that melts nuclear reactors and starts war.

FLEXIBLE SEVERITY RATHER THAN ABSOLUTES:

- Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component.
- Many organizations have designed and used satisfactory, quantitative, quality metrics.
- Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.
- The factors involved in bug severity are:
    1. **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.
    2. **Context and Application Dependency:** Severity depends on the context and the application in which it is used.
    3. **Creating Culture Dependency:** Whats important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software then games software vendors.
    4. **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.
    5. **The software development phase:** Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has been around.

TAXONOMY OF BUGS:

- There is no universally correct way categorize bugs. The taxonomy is not rigid.
- A given bug can be put into one or another category depending on its history and the programmer's state of mind.
- The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs
  (6) Test and Test Design Bugs.

REQUIREMENTS, FEATURES AND FUNCTIONALITY BUGS:
Various categories in Requirements, Features and Functionlity bugs include:

1. **Requirements and Specifications Bugs:**
   - Requirements and specifications developed from them can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.
   - The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted.
   - Requirements, especially, as expressed in specifications are a major source of expensive bugs.
   - The range is from a few percentage to more than 50%, depending on the application and environment.
   - What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.
   -
2. **Feature Bugs:**

   - Specification problems usually create corresponding feature problems.
   - A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect and correct. A wrong feature could have deep design implications.
   - Removing the features might complicate the software, consume more resources, and foster more bugs.

3. **Feature Interaction Bugs:**
   - Providing correct, clear, implementable and testable feature specifications is not enough.
   - Features usually come in groups or related features. The features of each group and the interaction of features with in the group are usually well tested.
   - The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.
   - Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore result in feature interaction bugs.

4. **Specification and Feature Bug Remedies:**

- Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.
- Such languages and systems provide short term support but in the long run, does not solve the problem.
- ***Short term Support:*** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.
- ***Long term Support:*** Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete testsand consistent test criteria.
- The specification problem has been shifted to a higher level but not eliminated.

5. **Testing Techniques for functional bugs:**

Most functional test techniques- that is those techniques which are based on a behavioral description of software, such as transaction flow testing, syntax testing, domain testing, logic testing and state testing are useful in testing functional bugs.

STRUCTURAL BUGS:

Various categories in Structural bugs include:

1.Control and Sequence Bugs:

- Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, sphagetti code, and worst of all, pachinko code.

- One reason for control flow bugs is that this area is amenable (supportive) to theoritical treatment.
- Most of the control flow bugs are easily tested and caught in unit testing.
- Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.
- Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

2. **Logic Bugs:**
    a. Bugs in logic, especially those related to misundertanding how case statements and logic operators behave singly and combinations
    b. Also includes evaluation of boolean expressions in deeply nested IF-THEN-ELSE constructs.
    c. If the bugs are parts of logical (i.e. boolean) processing not related to control flow, they are characterized as processing bugs.
    d. If the bugs are parts of a logical expression (i.e control- flow statement) which is used to direct the control flow, then they are categorized as control-flow bugs.
3. **Processing Bugs:**
    a. Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
    b. Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of grater-than-or-eual etc
    c. Although these bugs are frequent (12%), they tend to be caught in good unit testing.
4. **Initialization Bugs:**
    a. Initialization bugs are common. Initialization bugs can be improper and superfluous.
    b. Superfluous bugs are generally less harmful but can affect performance.
    c. Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc
    d. Explicit declaration of all variables, as in Pascal, can reduce some initialization problems.
5. **Data-Flow Bugs and Anomalies:**
    a. Most initialization bugs are special case of data flow anamolies.
    b. A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

DATA BUGS:

- Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values.

- Data Bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all.
- *Code migrates data:* Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.
- Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equal attention.

- **Dynamic Data Vs Static data:**
  - Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes and residues.
  - Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Clean up
  - Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bit pattern.
  - Compile time processing will solve the bugs caused by static data.

Information, parameter, and control:

Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

Content, Structure and Attributes:

- **Content** can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content.

- **Structure**relates to the size, shape and numbers that describe the data object, that is memory location used to store the content. (e.g A two dimensional array).

- **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (e.g. an integer, an alphanumeric string, a subroutine). *The severity and subtlelty of bugs increases as we go from content to attributes because the things get less formal in that direction.*

CODING BUGS:

- Coding errors of all kinds can create any of the other kind of bugs.
- Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
- If a program has many syntax errors, then we should expect many logic and coding bugs.
- The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.

INTERFACE, INTEGRATION, AND SYSTEM BUGS:

Various categories of bugs in Interface, Integration, and System Bugs are:

1.External Interfaces:

- The external interfaces are the means used to communicate with the world.
- These include devices, actuators, sensors, input terminals, printers, and communication lines.
- The primary design criterion for an interface with outside world should be robustness.
- All external interfaces, human or machine should employ a protocol. The protocol may be wrong or incorrectly implemented.
- Other external interface bugs are: invalid timing or sequence assumptions related to external signals

- Misunderstanding external input or output formats.

- Insufficient tolerance to bad input data.

2. **Internal Interfaces:**
    - Internal interfaces are in principle not different from external interfaces but they are more controlled.
    - A best example for internal interfaces are communicating routines.
    - The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.
    - Internal interfaces have the same problem as external interfaces.
3. **Hardware Architecture:**
    - Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
    - Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
    - The remedy for hardware architecture and interface problems is two fold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.
4. **Operating System Bugs:**
    - Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does.
    - Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls.
    - This approach may not eliminate the bugs but at least will localize them and make testing easier.
5. **Software Architecture:**
    - Software architecture bugs are the kind that called - interactive.
    - Routines can pass unit and integration testing without revealing such bugs.

- Many of them depend on load, and their symptoms emerge only when the system is stressed.
- Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
- Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

6. **Control and Sequence Bugs (Systems Level):**
   - These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.
   - The remedy for these bugs is highly structured sequence control.
   - Specialize, internal, sequence control mechanisms are helpful.
7. **Resource Management Problems:**
   - Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.
   - External mass storage units such as discs, are subdivided into memory resource pools.
   - Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc
   - **Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discovers them.
   - The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.
8. **Integration Bugs:**
   - Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
   - These bugs results from inconsistencies or incompatibilities between components.
   - The communication methods include data structures, call sequences, registers,

semaphores, communication links and protocols results in integration bugs.

- The integration bugs do not constitute a big bug category(9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.

9. **System Bugs:**

- System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.
- There can be no meaningful system testing until there has been thorough component and integration testing.
- System bugs are infrequent(1.7%) but very important because they are often found only after the system has been fielded.

TEST AND TEST DESIGN BUGS:

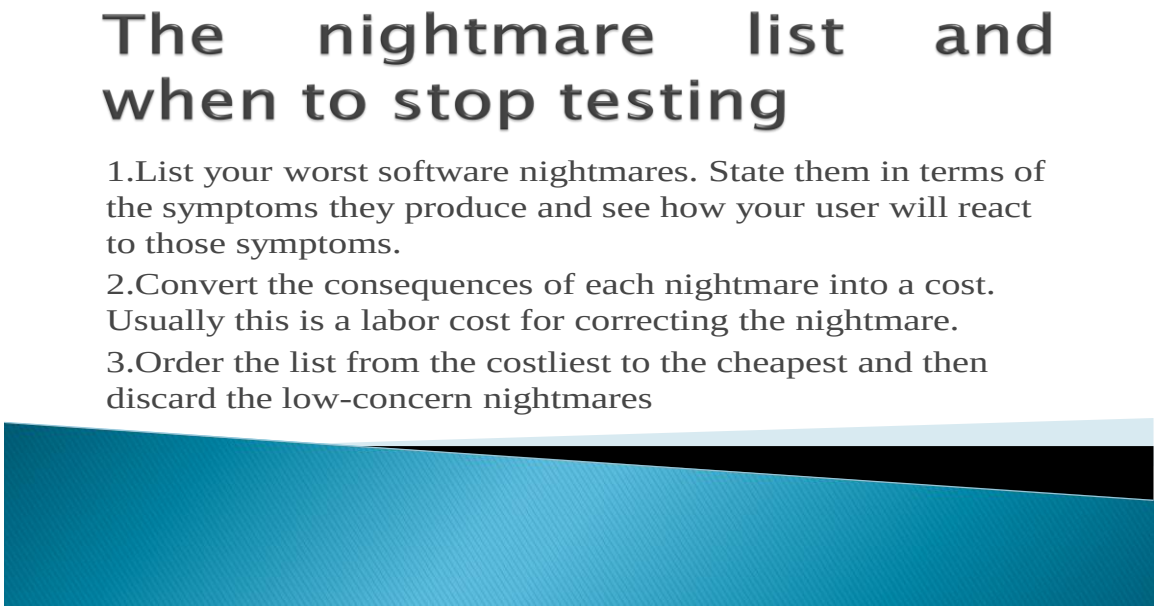
- Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.
- They require code or the equivalent to execute and consequently they can have bugs.
- Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavior is judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.

Remedies: The remedies of test bugs are:

1. **Test Debugging:** The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs and donot have to make concessions to efficiency.

2. **Test Quality Assurance:** Programmers have the right to ask how quality in independent testing is monitored.
3. **Test Execution Automation:** The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation errors. Test execution bugs are virtually eliminated by various test execution automation tools.
4. **Test Design Automation:** Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reduces the bug count - be it for software or be it for tests.

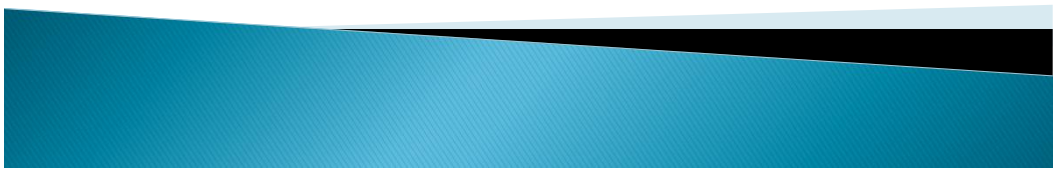
The Nightmare List and When to stop Testing :



4.Based on your experience, measured data and statistics postulate the bugs that are likely to create the symptoms expressed by each nightmare.

5.For each nightmare you've developed a list of possible causative bugs, order that list by decreasing probability.

6.Rank the bug types in order of decreasing importance to you.



7.Design tests(based on your knowledge of test techniques) and design your quality assurance inspection process using the methods that are most effective against the most important bugs

8.If a test is passed, then some nightmares or parts of them go away. If test is failed, then nightmare is possible, but upon correcting bug, it too goes away

9.Stop testing when probability of all nightmares has been shown to be inconsequential