# Software Testing

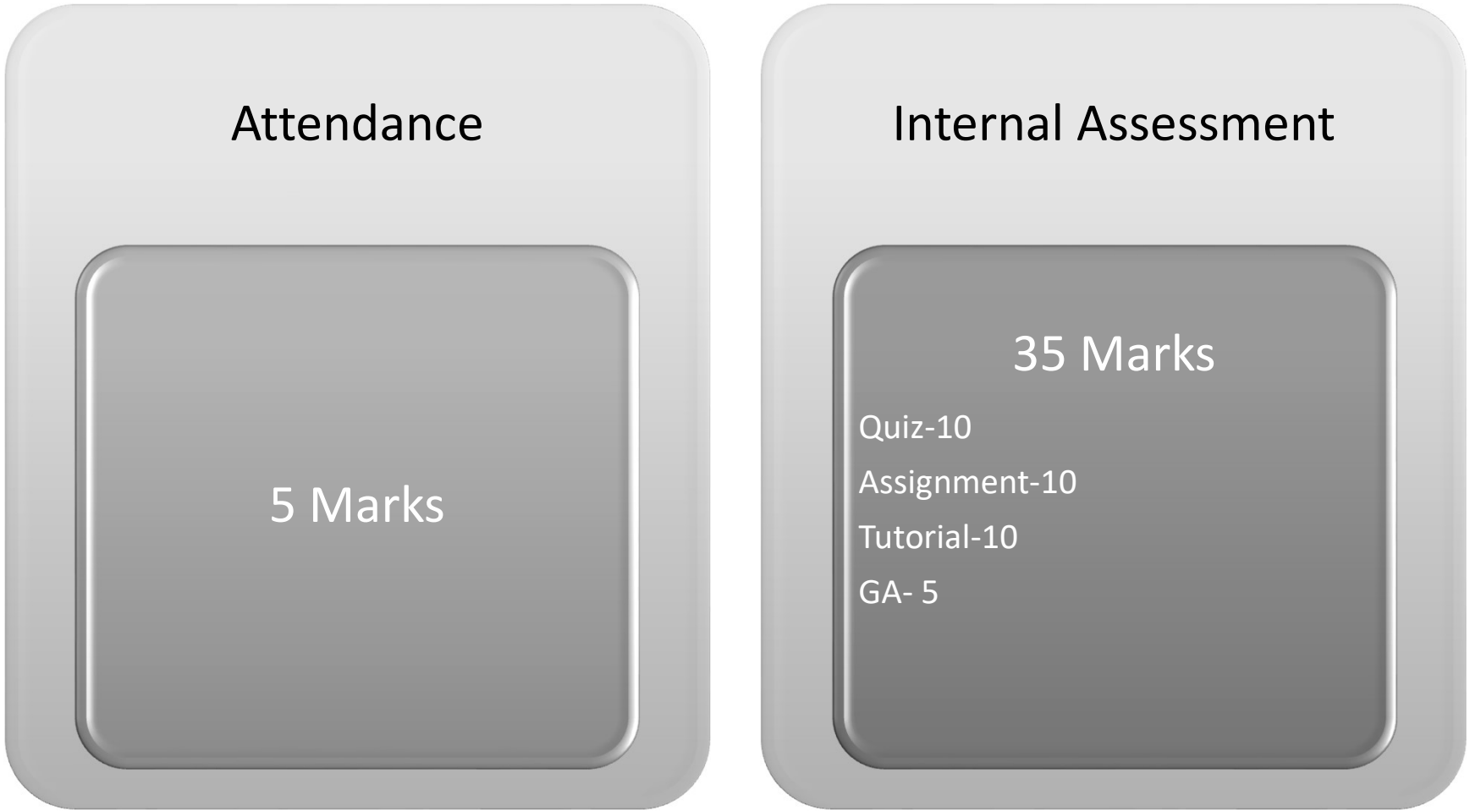
Dr. Sandip Mal

# Software Testing

CSE4017
LT Course
3 Credit

| CAT I ( Continuous Assessment Test I) | CAT II ( Continuous Assessment Test I) | FAT ( Final Assessment Test ) |
|---|---|---|
| • 50 Marks<br>• 90 Min. Exam<br>• 15 Marks Conversion Value | • 50 Marks<br>• 90 Min. Exam<br>• 15 Marks Conversion Value | • 100 Mark Exam<br>• 180 Min. Exam<br>• 30 Marks Conversion Value |

# Assessment Components



# Syllabus

## Module 1

Overview of Testing: Software Testing Definition, Debugging, Testing vs debugging. Purpose of testing, Dichotomies, model for testing, consequences of bugs, taxonomy of Bugs.

# Module 2

- **Flow graphs and Path testing:** Basics concepts of path testing, predicates, path predicates and achievable paths, path sensitizing, path instrumentation, application of path testing.
- **Transaction Flow Testing:** Transaction flows, transaction flow testing techniques.

**Software Testing:**

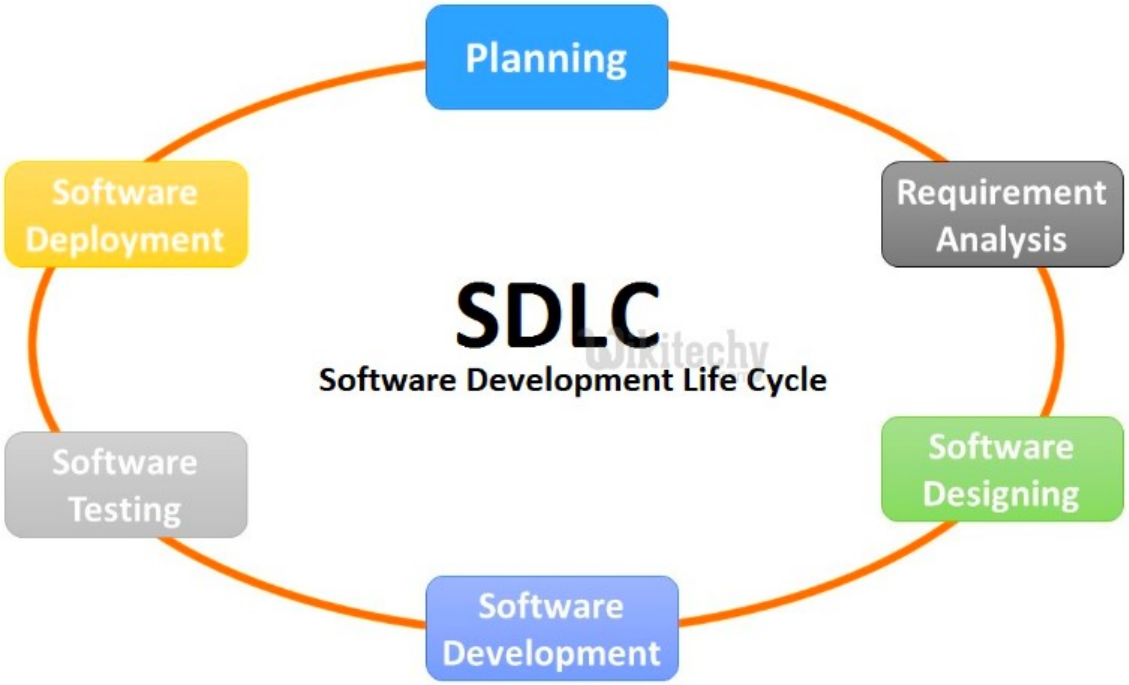
***Definition:*** *Software testing is the process of finding bugs or errors in the software.*

**Software Testing includes the following:**

- Find out the bugs / errors in the software
- Ensures the productivity and quality of the product
- Validate and verify the functionality and usability of the software
- Examination of code as well as execution of that code in various environments and conditions
- Examining the aspects of code: does it do what it is supposed to do and do what it needs to do.

# Overview of Testing

**The Purpose of Testing** Test design and testing takes longer than program design and coding. In testing each and every module is tested. Testing is a systematic approach that the given one is correct or not. Testing find out the bugs in a given software.



- Every software product has a target audience.
- For example, the audience for video game software is completely different from banking software.
- Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders, **Software testing** is the process of attempting to make this assessment.

**Goals for Testing:** Testing and test design as a parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs. Bug prevention is testing first goal. Bug prevention is better than the detection and correction. There is no retesting and no time wastage.

Bug prevention – Primary goal

Bug discovery – Secondary goal

# Phases in a Tester's Mental Life

**Phase 0:** There's no difference between testing and debugging. Here there is no effective testing, no quality assurance and no quality.

**Phase 1:** The purpose of testing is to show that the software works. Testing increases, software works decreases.

**Phase 2:** The purpose of testing is to show that the software doesn't works. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. It is never ending sequence.

**Phase 3 :** The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value. Here testing implements the quality control. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does.

**Phase 4 :** Here what testing can do and not to do. The testability is that goal for two reasons :

1.Reduce the labor of testing.

2.Testable code has fewer bugs that code that's hard to test.

## Some Differences

**Testing versus Debugging :** The phrase " Test and Debug " is treated as a single word.

- The purpose of testing is to show that a program has bugs.
- The purpose of debugging is find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

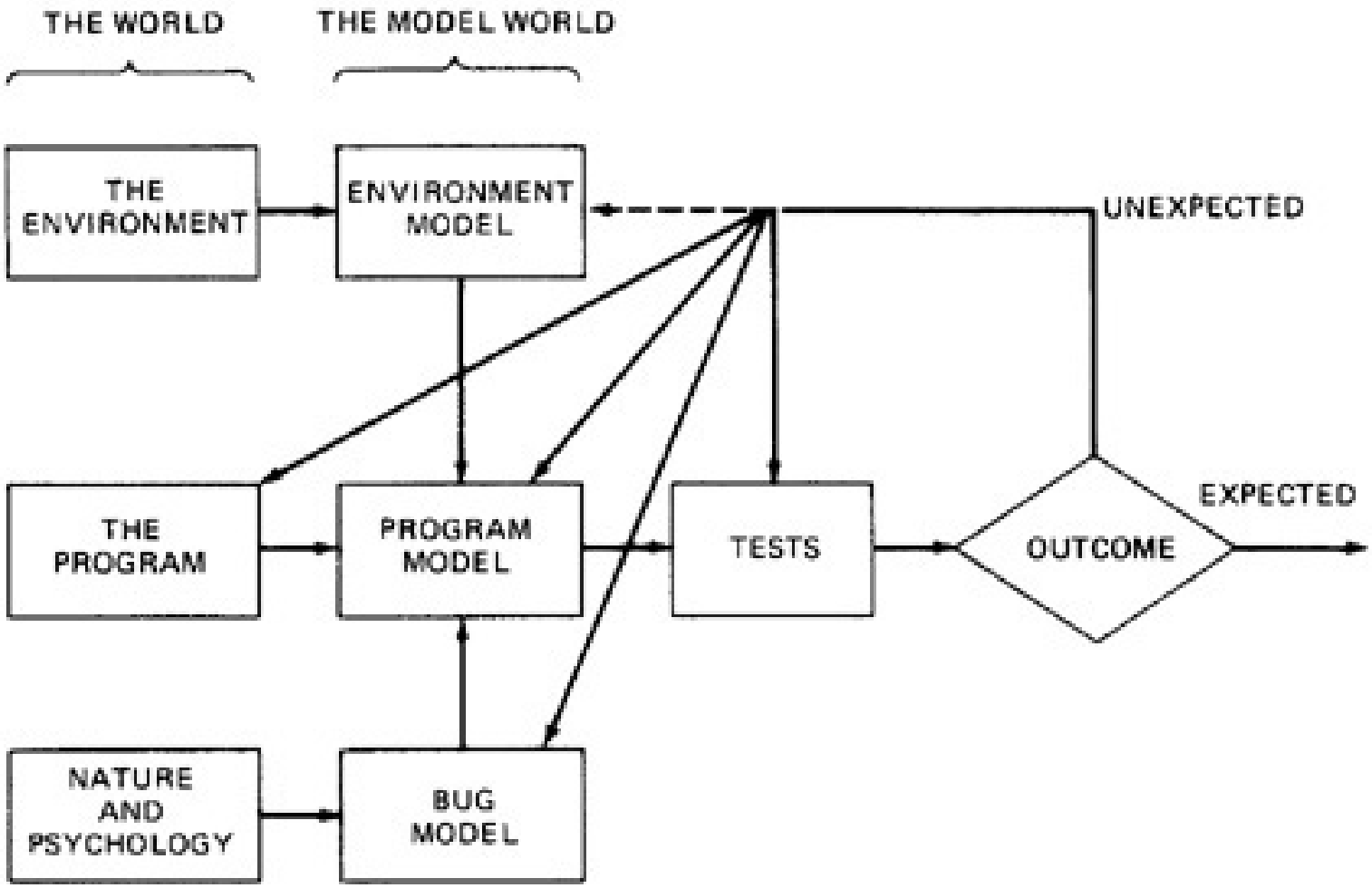
| Testing | Debugging |
| --- | --- |
| Testing starts with known conditions, uses predefined procedures and has predictable outcomes. | Debugging starts from possibly unknown initial conditions and the end can not be predicted except statistically. |
| Testing can and should be planned, designed and 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 intuitive 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. |

- Testing can be **Static** (examining the code and structure) and **Dynamic** (actually executing the program with test cases). Static testing, sometimes, is omitted.

- Software Testing is associated with **Verification** and **Validation**. Verification is process based and checks whether the software is correctly designed / functioning well or not?. Validation is product based and checks whether it is useful for customer or not?.

- Software Testing methods are traditionally divided into **Black Box Testing** and **White Box Testing**. Black box testing treats the software as a 'black box' , without any knowledge of internal implementation. Specification based testing, part of black box testing, tests the functionality of the software without considering the code. White box testing, in contrast to Black box testing, has access to the code, internal data structures and algorithms.

# MODEL FOR TESTING



## 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 in order to test it.
- If simple model of the program does not 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:**
  - **Benign Bug Hypothesis:** The belief that bugs are nice and logical. (Benign: Not Dangerous).
  - **Bug Locality Hypothesis:** The belief that a bug discovered with in a component effects only that component's behaviour.
  - **Control Bug Dominance:** The belief that errors in the control structures (if, switch etc) of programs dominate the bugs.
  - **Code / Data Separation:** The belief that bugs respect the separation of code and data.
  - **Lingua Salvator Est:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
  - **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.

**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:**
  - **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.
  - **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 components were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.
  - **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, start up and recovery.

- Testing can be done in different levels:
  - **Alpha Testing (Before Product Release):** Tests the software by potential end users or customers or an independent test team at developers site.
  - **Beta Testing (Before Product Release):** Simulated after Alpha testing. The software is released to groups of people outside the programming team. End products are called Beta-versions.

- Other types of tests include: *Grey Box testing*, *Regression testing*, *Acceptance testing*, *Performance testing*, *Stability testing*, *System integration testing*.

- **Testing Tools:** Software that tests Software. Many testing tools are available in market. But, there is no universal tool that can test any software.

# CONSEQUENCES OF BUGS

**IMPORTANCE OF BUGS:** The importance of bugs depends on frequency, correction cost, installation cost, and consequences.

- **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
- **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.
- **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.
- **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

***Importance = Frequency * (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:

- **Mild:** The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.
- **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
- **Annoying:** The system's behaviour because of the bug is dehumanizing. *E.g.* Names are truncated or arbitrarily modified.
- **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM wont give you money. My credit card is declared invalid.
- **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.

- **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.
- **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.
- **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.
- **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.

# TAXONOMY OF BUGS

**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.

# TAXONOMY OF BUGS

**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 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.

## 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.

## Functional Bugs:

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

# TAXONOMY OF BUGS

**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.

- **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.
  - One reason for control flow bugs is that this area is amenable (supportive) to theoretical 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.

- **Logic Bugs:**
  - Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and combinations
  - Also includes evaluation of boolean expressions in deeply nested IF-THEN-ELSE constructs.
  - If the bugs are parts of logical (i.e. boolean) processing not related to control flow, they are characterized as processing bugs.
  - 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.

- **Processing Bugs:**
  - Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
  - Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of grater-than-or-equal etc.
  - Although these bugs are frequent (12%), they tend to be caught in good unit testing.
- **Initialization Bugs:**
  - Initialization bugs are common. Initialization bugs can be improper and superfluous.
  - Superfluous bugs are generally less harmful but can affect performance.
  - 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
  - Explicit declaration of all variables, as in Pascal, can reduce some initialization problems.
- **Data-Flow Bugs and Anomalies:**
  - Most initialization bugs are special case of data flow anomalies.
  - 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.

# TAXONOMY OF BUGS

**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.

**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.*

# TAXONOMY OF BUGS

**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.

- **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.

# TAXONOMY OF BUGS

**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.

- **INTERFACE, INTEGRATION, AND SYSTEM BUGS:**
- Various categories of bugs in Interface, Integration, and System Bugs are:
  - **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.
  - **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.
  - **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.

- **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.
- **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.
- **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.

- **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.
- **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.
- **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.

# TAXONOMY OF BUGS

**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.

# 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 behaviour 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.