SOFTWARE TESTING METHODLOGIES

BY
Ch.Mallikarjuna Rao
M.Tech,Ph.D
Professor of CSE
G O K A R A J U R A N G A R A J U
INSTITUTE OF ENGINEERING & TECHNOLOGY

Chapter - 1

INTRODUCTION

What is software testing

- "The process of executing computer software in order to determine whether the results it produces are correct"

OR

- "The process of executing a program with the intent of finding errors"

OR

- "Program testing can be used to show the presence of bugs, but never their absence"

Aim of Testing

- "The aim is not to discover errors but to provide convincing evidence that there are none, or to show that particular classes of faults are not present"

OR

- "Testing is the measure of software quality"

Testing is a state of mind

- "If our goal is to show the absence of errors, we will find very few of them"
- "If our goal is to show the presence of errors, we will discover a large number of them"

Software Testing Techniques

1 The purpose of testing
2 Some Dichotomies
3 A Model For Testing
4 Playing pool and consulting Oracles
5 Is Complete Testing Possible

1.The Purpose of Testing

What we do?

- Testing consumes at least half of the labor Expended to produce a working program
- The effort put in to testing seems wasted if the tests do not reveal bugs

Purpose of Testing

- The purpose of Testing is to show that a program has bugs.
- The purpose of testing is to show that the software works.
- The purpose of testing is to show that the software doesn't works.

Productivity and Quality in Software

- In Production of consumer 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 will either be discarded or cycled back for rework and correction.
- Productivity is measured by the sum of the costs of the materials, the rework, and the discarded components and the cost of quality-assurance and testing.

Productivity and Quality in Software-continued.,

- There is a trade off between quality assurance costs and manufacturing costs.
  - If insufficient effort is spent in quality assurance, the reject rate will be high and so will the net cost.
  - If inspection is so good that all faults are caught as they occur, inspection costs will dominate, and again net cost will suffer.

Productivity and Quality in Software-continued.,

- 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 almost indistinguishable because the cost of a software copy is trivial.

Goals for Testing

- Primary Goal: Bug Prevention
- Secondary Goal: Bug Discovery
- Unfortunately the primary goal we cant achieve because we are human, so it must reach its secondary goal.
- Bug: A Bug is manifested in deviations from Expected behavior.
- Test Design: a test design must document expectations, the test procedures in details and the results of the actual test.

Phases in a Tester's Mental Life

- There's an attitudinal progression characterized by the following five phases:
- PHASE 0: There is no difference between testing and debugging, other than in support of debugging, testing has no purpose.
- PHASE 1:The purpose of testing is to show that the software works.
- PHASE 2:The purpose of testing is to show that the software doesn't works.
- PHASE 3: The purpose of testing is not to prove anything but to reduce the risk of not working to an acceptable value.
- PHASE 4: Testing is not an act, it is a mental discipline that results in low-risk software without much testing effort.

Test Design

- Tests themselves are need to be designed as like how the code must be designed and tested.
- The test designing phase of programming should be explicitly identified.
- Programming Process should be described as:

design->test design->code->test code->

-> program debugging-> testing.

Testing isn't Every thing

- Inspection methods
- Design style
- Languages
- Design methodologies and development environment

Inspection Methods

- These include walkthroughs, desk checking, formal inspections and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overlap.

Design style

- Design style means the stylistic criteria used by programmers to define what they mean by a good program.
- Sticking to outmoded style like "tight code" or "optimizing" for performance destroys quality.
- Adopting stylistic objectives such as testability, openness, and clarity can do much to prevent bugs.

Languages

- The source language can help reduce certain kinds of bugs.
- Prevention of Bugs is a driving force in the evolution of new Languages.
- Curiously, though, programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the language used.

Design methodologies and Development Environment

- The design methodology (the development process used and the environment in which that methodology is embedded) can prevent many kinds of bugs.

The Pesticide Paradox and the Complexity Barrier

- First Law: the Pesticide Paradox – Every method you use to prevent or find to bugs leave a residue of subtler bugs against which those methods are ineffectual
- Second Law: the Complexity Barrier- Software Complexity grows to the limits of our ability to manage that complexity.

Some Dichotomies

- Testing Versus Debugging
- Function Versus Structure
- The Designer Versus the Tester
- Modularity Versus Efficiency
- Small Versus Large
- The Builder Versus the Buyer

Testing Versus Debugging

- Testing and Debugging are often lumped under the same heading, and it's no wonder that their roles are often confused;
- The purpose of testing is to show that a program has bugs.
- The purpose of debugging is find the error or misconceptions 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 psychology.

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 black box. It is subjected to inputs and its outputs are verified for conformance to specified behavior.
  - Structural testing does look at the implementation details. Things as programming style, control method, source of language, database design, and coding details dominate structural testing.

The Designer Vs The Tester

- If testing were wholly based on functional specifications and independent of implementation details, then the designer and the tester could be completely separated.
- Conversely, to design a test plan based only on a system's structural details would require the software designer's knowledge, and hence only he could design the tests.
- As one goes from unit testing to unit integration, to component testing and integration, to system testing, and finally to formal system feature testing, it is increasingly more effective if the tester and the programmer are different persons.

Modularity Versus Efficiency

- A module is a discrete, well defined, small component of a system.
- There is a trade off between modularity and efficiency.
- The overall complexity minimization can be achieved by the balance between internal complexity and interface complexity.
- As with system design, artistry comes into test design in setting the scope of each test and groups of tests so that test design, test debugging, and test execution labor are minimized without compromising effectiveness.

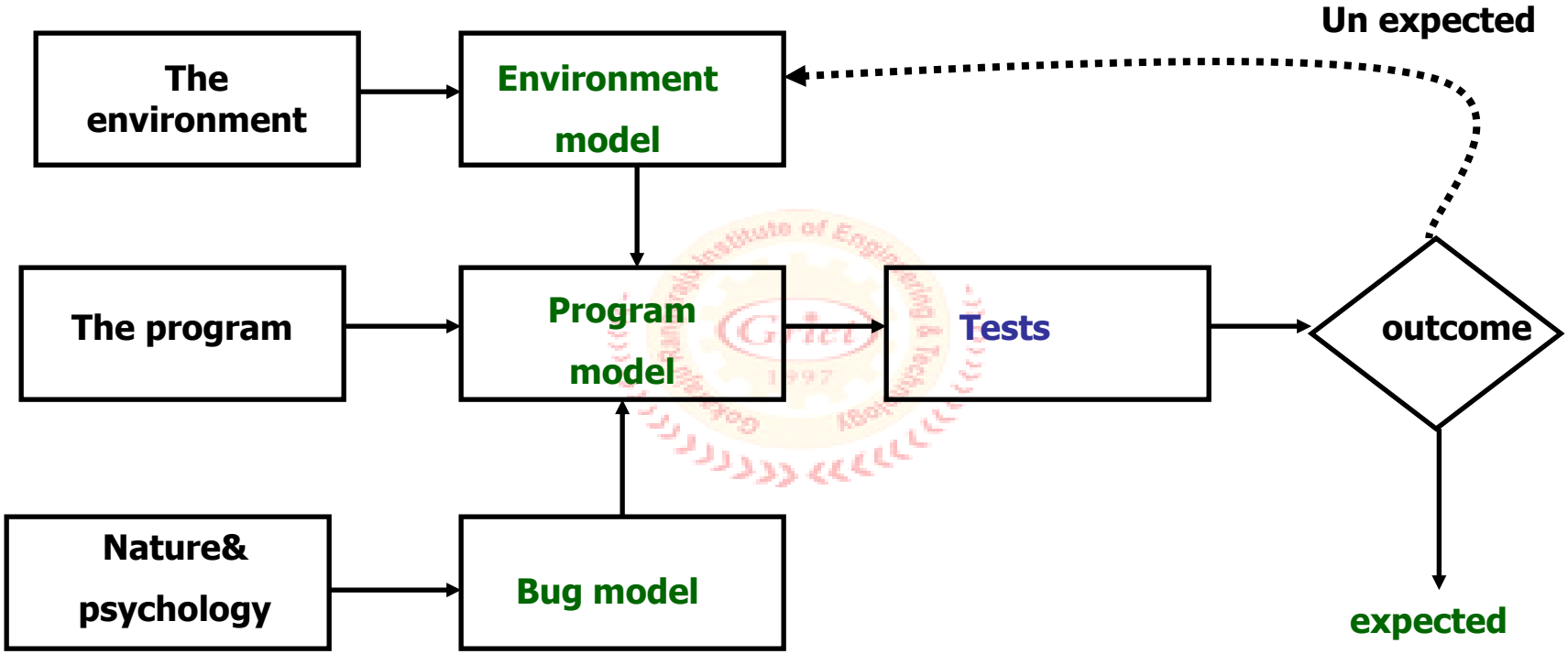
Small versus Large

- Programming in large means constructing programs that consists of many components written by many different persons,
- 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.

The Builder Versus the Buyer

- The builder, who designs for and is accountable to
- The buyer, who pays for the system in the hope of profits from providing services to
- The user, the ultimate beneficiary or victim of the system. The user's interests are guarded by
- The tester, who is dedicated to the builder's destruction and
- The operator, who has to live with the builder's mistakes, the buyer's murky specifications, the tester's oversights, and the user's complaints.

A Model for Testing



The Environment

- A program's environment includes the hardware and software required to make it run. For example., for online systems the environment may include communication lines, other systems, terminals, and operators.
- The environment also includes all programs that interact with and used to create the program under test such as operating system, loader, linkage editor, compiler and utility routines.
- Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

The 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 behavior, 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 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 unable to test effectively and unable to justify the dirty tests most programs need.

Optimistic notions about bugs.

- Benign Bug Hypothesis
- Bug Locality Hypothesis
- Control Bug Dominance
- Lingua Salator Est
- Corrections Abide
- Silver Bullets
- Angelic Testers

Optimistic notions about bugs- continued.,

- Benign Bug Hypothesis: The belief that bugs are nice, tame and logical.
- Bug Locality Hypothesis: the belief that a bug discovered with in a component affects only that component's behavior.
- Control Bug Dominance: the belief that errors in the control structures of programs dominate the bugs.
- Lingua Salator Est: the hopeful belief that language syntax and semantics eliminate most bugs.
- Corrections Abide: the mistaken belief that a corrected bug remains corrected.
- Silver Bullets: the mistaken belief that language, design method, representation, environment grants immunity from bugs.
- 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 predicted, tests documented, commands executed, and the results observed; all these steps are subject to error.

Testing and Levels

- We do many distinct kinds of testing on a typical software system.
  - Unit testing
  - Component testing
  - Integration testing
  - System testing

Unit, Unit Testing

- Unit : A Unit is the smallest testable piece of software, by which It mean that it can be compiled, assembled, linked, loaded and put under the control of a test harness or driver.
- A unit is usually the work of one programmer and consists of several hundred or fewer lines of code.
- Unit Testing: 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.

Component, Component Testing

- Component: a component is an integrated aggregate of one or more units.
- Component Testing: 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, Integration Testing

- Integration: It is a process by which components are aggregated to create larger components.
- Integration Testing: Integration Testing is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent.

System, System Testing

- System: a system is a big component.
- System Testing: system testing is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects.
- It includes testing for performance, security, accountability, configuration sensitivity, start up and recovery.

The 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 behavior.

Playing pool and Consulting oracles

- Playing pool: Testing is like a Playing Pool. There is real testing and kiddie testing.
- In kiddie testing the tester says, after the fact, that the observed outcome of the test was 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 is input/outcome oracle-an oracle that specifies the expected outcome for a specified input.

Sources of Oracle

- If every test designer had to analyze and predict the expected behavior 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 named:
  - Kiddie Testing
  - Regression Test Suites
  - Purchased suites and Oracles
  - Existing Program

Sources of Oracle-continued.,

- 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's 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.
- Regression Test Suites: As 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.

Sources of Oracle-continued.,

- Purchased Suites 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.
- 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.
  - Functional Testing
  - Structural Testing
  - Formal Proofs of correctness
- Each approach leads to the conclusion that complete testing, in the sense of a proof is neither theoretically nor practically possible.

Functional testing-impossible

- 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 2^{80} possible input streams and corresponding outcomes, so complete functional testing in this sense is IMPRACTICAL.
- But even THEORETICALLY, 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.

Structural Testing-impossible

- 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.

Formal proofs of correctness-impossible

- Formal proofs of correctness rely on a combination of functional and structural concepts.
- Requirements are stated in a formal language 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.

Theoretical 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."

Chapter - 2

THE TAXONOMY OF BUGS

The Consequences 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's been found? That cost is the sum of two factors.(1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size.

The Consequences of BUGS-continued.,

- 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? This can be measured by the size of the awards made by juries to the victims of the bug.
- A reasonable metric for bug importance is: *Importance($)=frequency*(correction_ cost+installation_cost+consequential_cost)*

How Bugs Affect Us- Consequences

- Bug consequences range from mild to catastrophic. These are measured in terms of human rather than machine. Some consequences on a scale of one to ten.
- Mild: The symptoms of the bug offend us aesthetically; a misspelled output or a misaligned printout.
- Moderate: outputs are misleading or redundant. The bug impacts the system's performance.

How Bugs Affect Us- Consequences-continued.,

- Annoying: The system's behavior, because of the bug is dehumanizing. Names are truncated or arbitrarily modified.
- Disturbing: it refuses to handle legitimate transactions. The ATM wont give you money. My credit card is declared invalid.
- Serious: it loses track of transactions. Not just the transaction it self but the fact that the transaction occurred. Accountability is lost.

How Bugs Affect Us- Consequences-continued.,

- 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 into withdrawals.
- Extreme: the problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
- Intolerable: Long term un recoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.

How Bugs Affect Us- Consequences-continued.,

- Catastrophic: the decision to shutdown is taken out of our hands because the system fails.
- Infectious: what can be worse than a failed system? One that corrupts other systems even though it does not 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 the number of bugs and their severity is only one component.
- Many organizations have designed and use satisfactory, quantitative, quality metrics. 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 are:
  - **Correction Cost**
  - **Context and Application Dependency**
  - **Creating Culture Dependency**
  - **User Culture Dependency**
  - **The Software Development Phase**

A Taxonomy for Bugs

- There is no universally correct way to 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:**
  - Requirements, features and functionality Bugs
  - Structural Bugs
  - Data Bugs
  - Coding Bugs
  - Interface, integration and system Bugs
  - Test and Test Design Bugs

Requirements, Features and Functionality Bugs

- Various categories in Requirements, Features and Functionality Bugs:
  - Requirements and Specifications
  - Feature Bugs
  - Feature Interaction Bugs

Requirements and Specifications Bugs

- **Requirement and Specification**: Requirements and the 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 in progress. The features are added, modified and deleted.
- Requirements, especially as expressed in a specification are a major source of expensive bugs.
- The range is from a few percentage to more than 50%, depending upon the application and the environment.
- What hurts most about these bugs is that they're the earliest to invade the system and the last to leave.

Feature Bugs

- **Feature Bugs:** Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. 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.

Feature Interactions Bugs

- **Feature interactions:** 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 within each 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 feature interaction bugs.

Specification & feature bug remedies

- Specification & feature bug Remedies: most feature bugs are rooted in human to human communication problems. One solution is the use of high level, formal specification languages or systems.
- Such languages and systems provide short term support but, in the long run, do 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 it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria.
- A specification written in that language could theoretically be compiled into object code into HOL programming.
- The specification problem has been shifted to a higher level but not eliminated.

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:
  - Control and Sequence Bugs
  - Logic Bugs
  - Processing Bugs
  - Initialization Bugs
  - Data flow Bugs and Anomalies

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, and ill-conceived switches.
- One reason for control flow bugs is that this area is amenable 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 are related to:

misunderstanding how case statements and logic operators behave singly and combinations

Include non existent cases

Improper layout of cases

"Impossible" cases that are not impossible

a "don't care" case that matters

Improper negation of boolean expression

Improper simplification and combination of cases

Overlap of exclusive cases

Confusing "exclusive OR" with "inclusive OR"

The best defense against this kind of bugs is a systematic analysis of cases.

Processing Bugs

- Processing Bugs include arithmetic bugs, algebraic and mathematical function evaluation, algorithm selection and general processing.
- The reasons for Processing Bugs:
  - Incorrect conversion from one data representation to another.
  - Improper use of <,>,<=,>=
  - Improper comparison between various formats
  - Although these bugs are frequent 12%, they tend to be caught in good unit testing and also tend to have localized effects.

Initialization Bugs

- Initialization bugs are common, both improper and superfluous initialization occur.
- Typical initialization bugs are:
  - Forgetting initialize work space, registers
  - Forgetting initialize data areas before first use or assuming that they are initialized else where
  - A bug in the first value of a loop control parameter
  - Accepting the initial value without a validation check
  - Initializing to the wrong format, data representation or type

Data flow Bugs and Anomalies

- Most initialization bugs are special case of data flow anomalies.
- A data flow anomaly occurs when 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.
- The increase in the proportion of the source statements devoted to data definition is a direct consequence of two factors:
  - The dramatic reduction in the cost of main memory and disc storage.
  - The high cost of creating and testing software.

Dynamic Versus Static

- Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, typically the processing time of the transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes, and residues.
- The dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of three ways:
  - Cleanup after use by the user
  - Common cleanup by the resource manager
  - 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.

Coding Bugs

- Coding errors of all kinds can create any of the other kinds 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 called 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:
  - External Interfaces
  - Internal interfaces
  - Hardware Architecture
  - Operating system
  - Software Architecture
  - Control and Sequence Bugs
  - Integration bugs
  - System bugs

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.
- There is a trade-off between the internal interfaces and the complexity of the interfaces.

Hardware Architecture

- Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
- Examples of hardware architecture bugs:
  - Paging mechanism ignored
  - Address generation error
  - i/o device operation or instruction error
  - i/o device address error
  - Misunderstood device status code
  - Device protocol error
  - Expecting the device to respond too quickly
  - Waiting too long for a response
  - Ignoring channel throughput limits
  - Incorrect interrupt handling

Remedy for hardware architecture and interface problems

- The remedy for hardware architecture and interface problems is two fold
  - Good programming and testing
  - 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 Code is reentrant or not reentrant
  - Assumption that memory and registers were initialized or not initialized
- Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

System level - Control and sequence Bugs

- These bugs include:
  - Ignored timing
  - Assuming that events occur in a specified sequence
  - Starting a process before its prerequisites are met
  - Waiting for an impossible combination of prerequisites
  - Not recognizing when prerequisites have been met
  - Specifying wrong priority, program state and level
  - 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.
- Here are some resource management and usage bugs:
  - Required resource not obtained
  - Wrong resource used
  - Resource is already in use
  - Resource not returned to the right pool
  - Fractionated resources not properly recombined
  - Resource dead lock

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

Resource management Remedies-continued.,

- Complicated resource structures are often designed in a misguided attempt to save memory by three sized transactions.
- The reasoning is faulty because:
- Memory is cheap and getting cheaper
- Complicated resource structures and multiple pools need management software
- The complicated scheme takes additional processing time, and all resources are held in use a little longer.
- The basis for sizing the resource is often wrong. A typical choice is to make the buffer block's length equal to the length required by an average transaction-usually a poor choice.
- A correct analysis shows that the optimum resource size is usually proportional to the square root of the transaction's length.
- The second design remedy is that is to centralize the management of all pools, either through centralized resource managers, common resource management sub routines, resource management macros, or a combination of these.

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.

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 is incorrect or impossible.

Test Design Bugs-Remedies

- The remedies for test bugs are:
  - Test Debugging
  - Test Quality Assurance
  - Test Execution Automation
  - Test Design Automation

Test Debugging

- The first remedy for test bugs is testing and debugging the tests.
- The differences between the test debugging and program debugging are not fundamental.
- Test debugging is easier because tests, when properly designed are simpler than programs.
- Test debugging tend to have a localized effects.

Test Quality Assurance

- Programmers have the right to ask how quality in independent testing and test design is monitored.
- The sequence like test testers, test test testers? Should not converge.

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 programmer and operator errors.
- Test execution bugs are virtually eliminated by various test execution automation tools

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.