Software Testing and Maintenance

Software Testing Fundamentals

- Software Testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding.
- It concerned with the actively identifying errors in software
- Testing of software is a means of measuring or assessing the software to determine its quality.
- Testing is dynamic assessment of the software
  - Sample input
  - Actual outcome compare with expected outcome

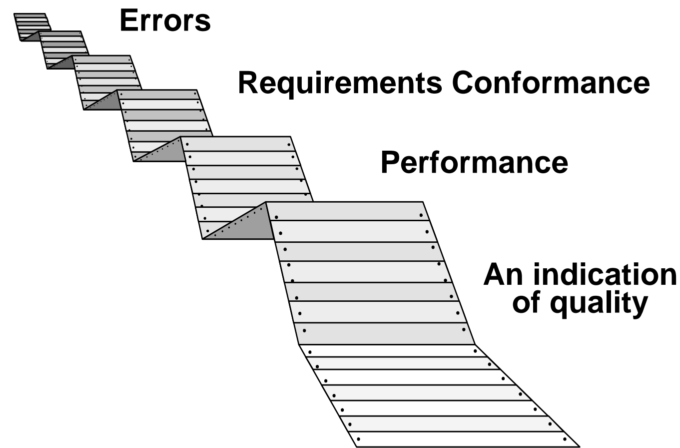
Testing Objectives

- Software Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
- It is the process of checking to see if software matches its specification for specific cases called **Test Case**.
- A **Good Test Case** is one that has a high probability of finding an as yet undiscovered error.
- A **Successful Test** is one that uncovers an as yet undiscovered error.

Testing vs. Debugging

- Testing is different from debugging
- Debugging is removal of defects in the software, a correction process
- Testing is an assessment process
- Testing consumes 40-50% of the development effort

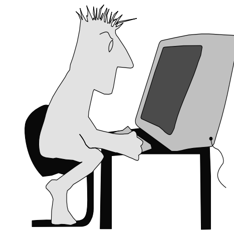
What can Testing Show?



Who Tests the Software?

Developer

Understands the system
but, will test "gently"
and, is driven by "delivery"



Independent Tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

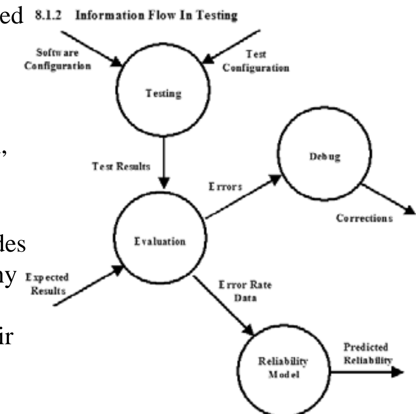
Testing Paradox

- To gain confidence, a successful test is one that the software does as in the functional spec.
- To reveal error, a successful test is one that finds an error.
- In practice, a mixture of Defect-revealing and Correct-operation tests are used.

Developer performs Constructive Actions
Tester performs Destructive Actions

Information Flow in Testing

- Two classes of input are provided to the test process:
  - **Software Configuration**: includes Software Requirements Specification, Design Specification, and source code.
  - **Test Configuration:** includes Test Plan and Procedure, any testing tools that are to be used, and test cases and their expected results.



Necessary Conditions for Testing

- A controlled/observed environment, because tests must be exactly reproducible
  - Sample Input – test uses only small sample input (limitation)
  - Predicted Result – the results of a test ideally should be predictable
- Actual output must be able to compare with the expected output

Attributes of a "Good Test"

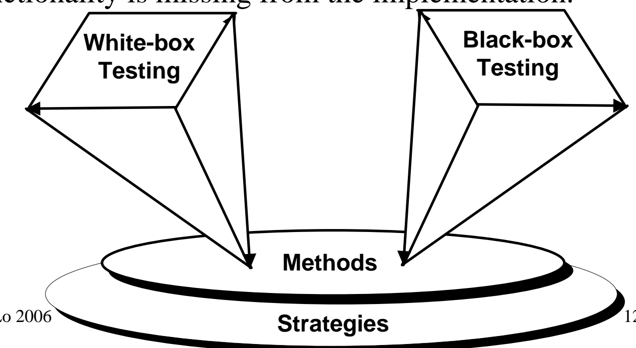
- A good test has a high probability of finding an error.
  - The tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant.
  - Testing time and resources are limited.
  - There is no point in conducting a test that has the same purpose as another test.
- A good test should be neither too simple nor too complex.
  - Side effect of attempting to combine a series of tests into one test case is that this approach may mask errors.

Attribute of Testability?

- Operability — It operates cleanly
- Observability — The results of each test case are readily observed
- Controllability — The degree to which testing can be automated and optimized
- Decomposability — Testing can be targeted
- Simplicity — Reduce complex architecture and logic to simplify tests
- Stability — Few changes are requested during testing
- Understandability — The purpose of the system is clear to the evaluator

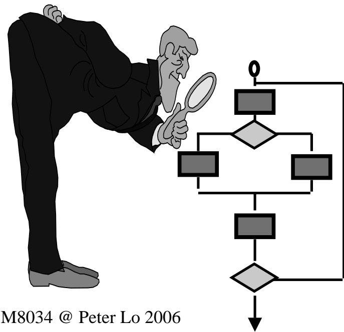
Software Testing Technique

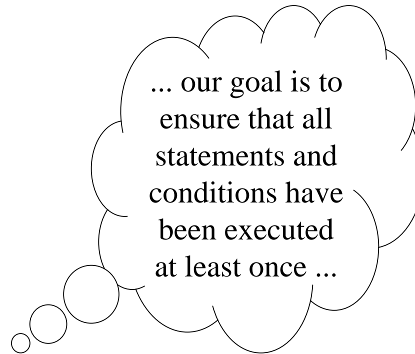
- **White Box Testing**, or **Structure Testing** is derived directly from the implementation of a module and able to test all the implemented code
- **Black Box Testing**, or **Functional Testing** is able to test any functionality is missing from the implementation.



White Box Testing Technique

- White box testing is a test case design method that uses the control structure of the procedural design to derive test cases.



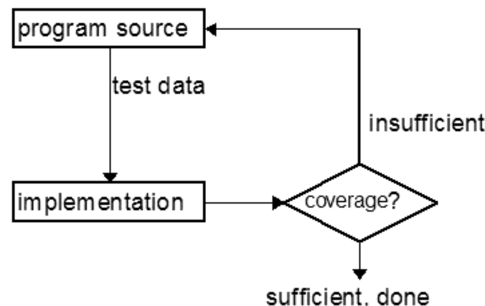


White Box Testing Technique

- White Box Testing of software is predicated on close examination of procedural detail.
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and / or loops.
- The status of the program may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

Process of White Box Testing

- Tests are derived from an examination of the source code for the modules of the program.
- These are fed as input to the implementation, and the execution traces are used to determine if there is sufficient coverage of the program source code



Benefit of White Box Testing

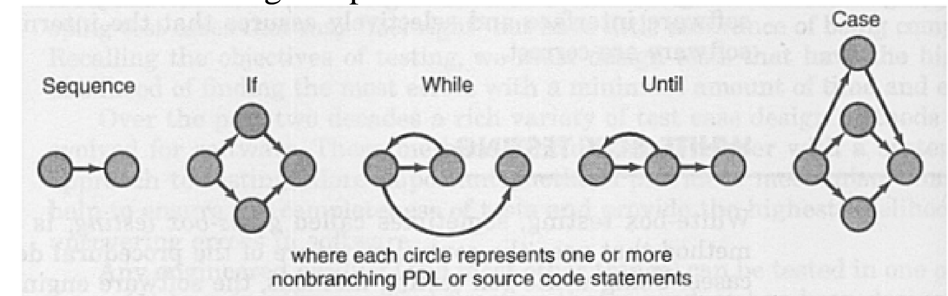
- Using white box testing methods, the software engineer can derive test cases that:
  - Guarantee that all independent paths within a module have been exercised at least once;
  - Exercise all logical decisions on their true or false sides;
  - Execute all loops at their boundaries and within their operational bounds ; and
  - Exercise internal data structures to ensure their validity.

Reasons to Test the Program Logic

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will evaluated
- Misconceptions that a logical path is unlikely to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random.

Control Flow Graph (CFG)

- It is a graphical representation of the way in which the sequence of control flows from one part of the program to the next.
  - Nodes represent Statements
  - Edges represent Flow of Control



Definition of Control Flow Graph

- Control Flow Graph is a directed graph, G, augmented with a unique entry node START and a unique exit node STOP such that each node in the graph has at most two successors.
- For any node N in G there exists a path from START to N and a path from N to STOP.

Basis Path Testing

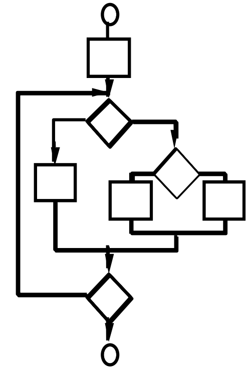
- Basis Path Testing is a white box testing techniques proposed by Tom McCabe.
- Enables that test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Basis Path Testing Procedure

- The basis path testing can be applied as a series of steps:
  - Using the design or code as foundation, draw a corresponding flow graph.
  - Determine the Cyclomatic Complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

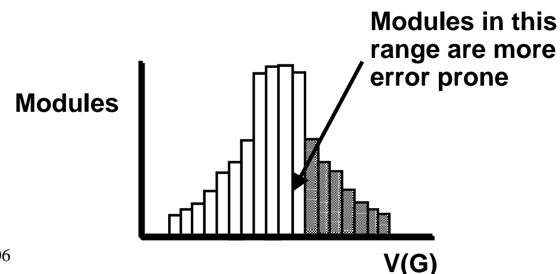
Basis Path Testing Notes

- You don't need a flow chart, but the picture will help when you trace program paths
- Count each simple logical test, compound tests count as 2 or more
- Basis path testing should be applied to critical modules



Cyclomatic Complexity

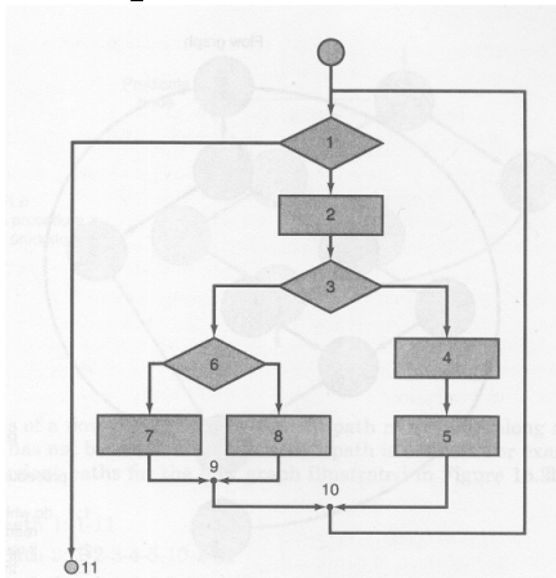
- Cyclomatic Complexity V(G) is a software metric that provides a quantitative measure of the logical complexity of a program.
- The higher V(G), the higher the probability or errors.



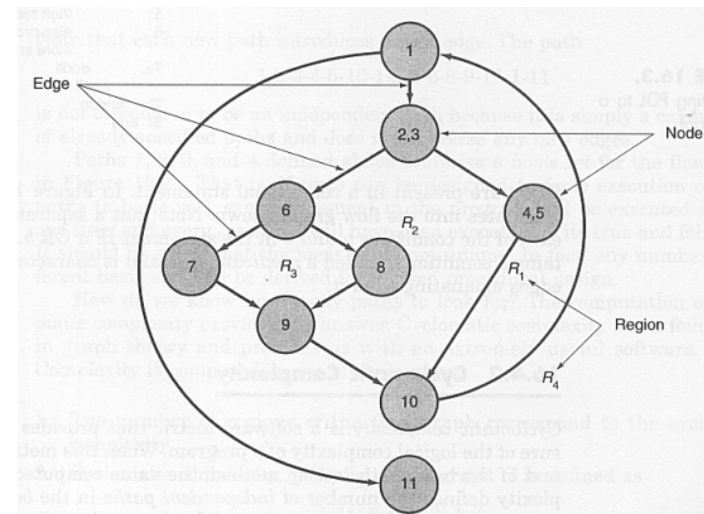
Calculation of Cyclomatic Complexity

- Complexity can be computed in one of several ways:
  - V(G) = Number of regions of the flow graph
  - V(G) = E – N + 2; where E is the number of flow graph edges and N is the number of flow graph nodes.
  - V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G (Predicate notes are characterized by two or more edges emanating from it).

Example



Answer: Graph



Answer: V(G) Calculation

- The flow has 4 regions (marked as R1, R2, R3 and R4); hence V(G) = 4
- V(G) = E - N + 2 = 11 edges - 9 nodes + 2 = 4
- V(G) = P + 1 = 3 Predicate Notes + 1 = 4
  - Predicate nodes are: node (2,3), node (1), and node (6).

Answer: Independent Path

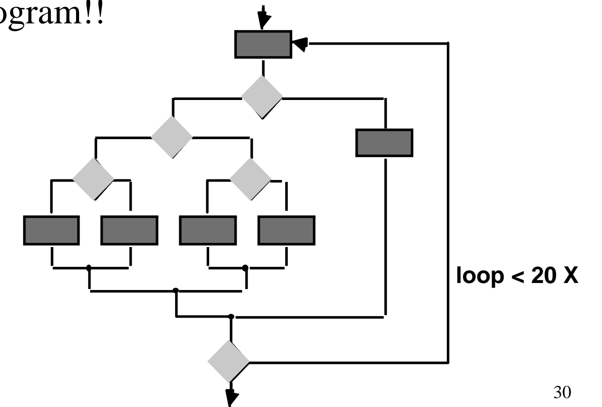
- Thus, the value of V(G) provides us with an upper bound for the number of independent paths ....(etc...)
- The set of independent paths are:
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11

Statement Coverage

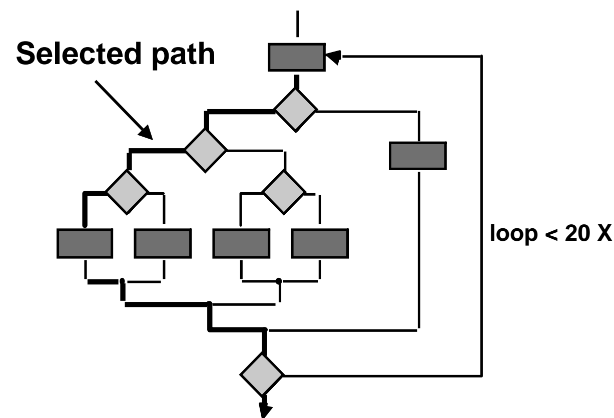
- Aim at design test which cause statements to be executed.
- The level of coverage a set of test cases achieves is equal to the fraction of statements which get executed by one or more test cases.
- 100% coverage means each statement of the program must be executed at least once.

Exhaustive Testing

- There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!



Selective Testing



Branch Coverage

- A branching node (or predicate node) is one which has more than one outgoing edge.
- 100% branch coverage means each predicate is executed at least twice.
- One true and one false.
- Every edge in the CFG must be executed to achieve 100%.

Path Coverage

- To achieve 100% path coverage every path from the entry node to the exit node of the CFG of the program must be traversed during at least one execution of the program.
- Presence of loops means infinitely paths.
- To overcome this a limit in number of executions of loop will be placed.

Condition Testing

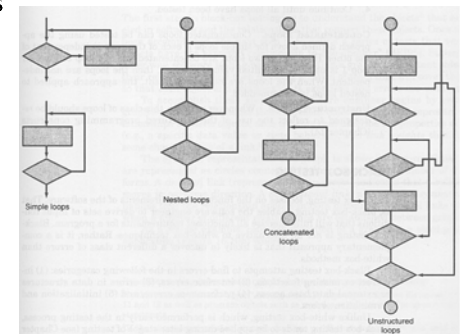
- Simple condition is a Boolean variable or relational expression
- Condition testing is a test case design method that exercises the logical conditions contained in a program module, and therefore focuses on testing each condition in the program.

Data Flow Testing

- The Data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program

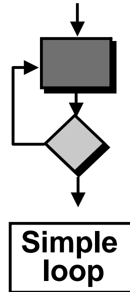
Loop Testing

- Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs
- Four classes of loops:
  1. Simple loops
  2. Concatenated loops
  3. Nested loops
  4. Unstructured loops



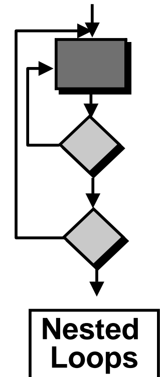
Test Cases for Simple Loops

- Where n is the maximum number of allowable passes through the loop:
  - Skip the loop entirely
  - Only one pass through the loop
  - Two passes through the loop
  - m passes through the loop where m<n
  - n-1, n, n+1 passes through the loop



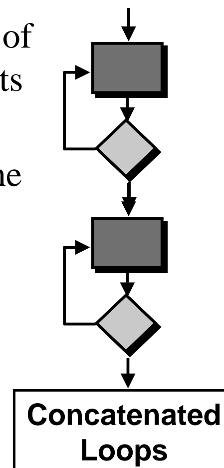
Test Cases for Nested Loops

- Start at the innermost loops. Set all other loops to minimum values
- Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter values. Add other tests for out-of -range or excluded values
- Work outward, conducting tests for the next loop but keeping all other outer loops at minimum values and other nested loops to "typical" values
- Continue until all loops have been tested



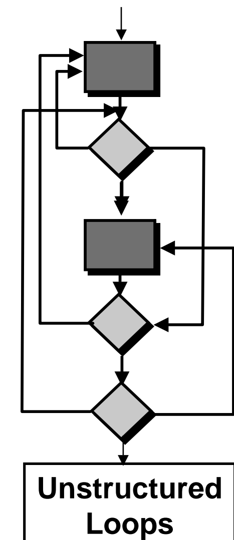
Test Cases for Concatenated Loops

- If each of the loops is independent of the others, perform simple loop tests for each loop
- If the loops are dependent, apply the nested loop tests



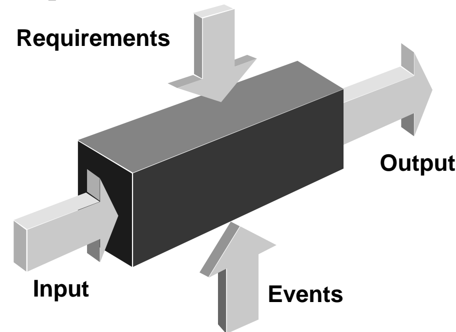
Test Cases for Unstructured Loops

- Whenever possible, redesign this class of loops to reflect the structured programming constructs



Black Box Testing

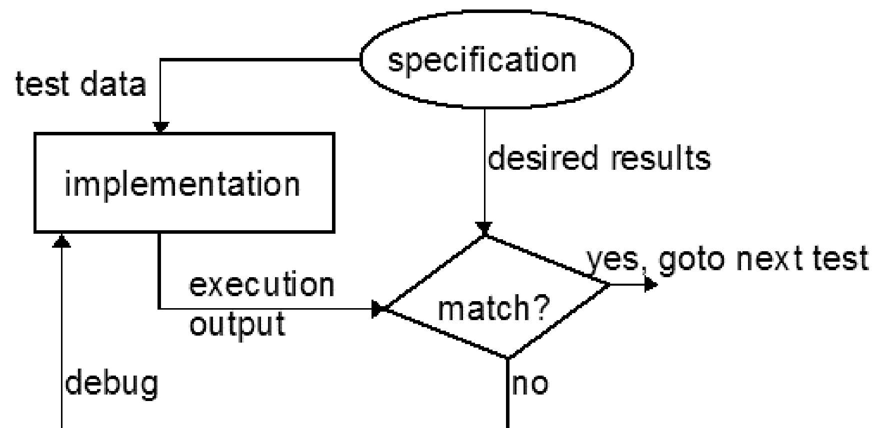
- **Black Box Testing** focus on the functional requirements of the software, i.e. derives sets of input conditions that will fully exercise all functional requirements for a program.
- Black box is based upon the specification of a module rather than the implementation of the module.



Black Box Testing

- Black box testing attempts to find errors in the following categories:
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external databases access
  - Performance errors
  - Initialization and termination errors.

Process of Black Box Testing

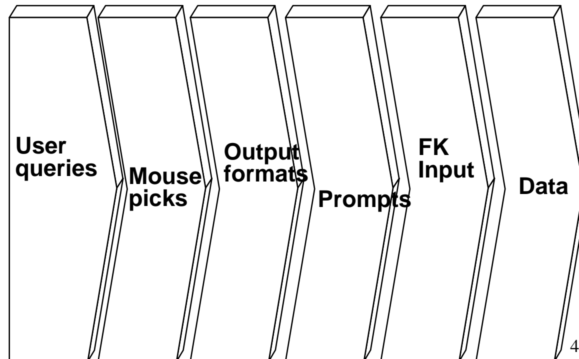


Random Testing

- Input is generated at random and submitted to the program and corresponding output is then compared.

Equivalence Partitioning

- Divides the input domain of a program into classes of data.
- Strives to define a test case that uncovers classes of errors



Equivalence Class Definition

- Equivalence classes may be defined according to the following guidelines:
  - If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - If an input condition specifies a member of a set, one valid and one invalid class are defined
  - If an input condition is Boolean, one valid and one invalid class are defined

Sample Equivalence Classes

Valid data

user supplied commands
responses to system prompts
file names
computational data
physical parameters
bounding values
initiation values
output data formatting
responses to error messages
graphical data (e.g., mouse picks)

Invalid data

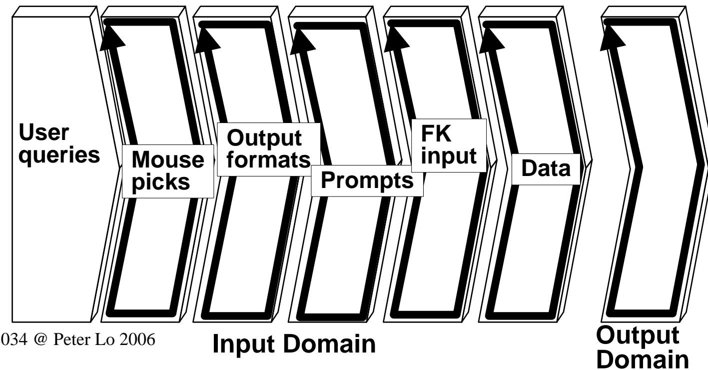
data outside bounds of the program
physically impossible data
proper value supplied in wrong place

Example

- Input Data variable is Employment Age, as used in a company database system. Employment age is defined as an integer variable, and has an acceptable range of 16 to 65 (years old).
- Since this is a range, there will be one valid, and two invalid equivalence classes.
- **The Valid Class:** Employment age = from 16 to 65, inclusive.
  - An example of a test case falling into this class would be Employment age = 30.
- **The First Invalid Class:** Employment age < 16.
  - An example of a test case falling into this class would be Employment age = 10.
- **The Second Invalid Class:** Employment age > 65.
  - An example of a test case falling into this class would be Employment age = 80.

Boundary Value Analysis

- Boundary value analysis (BVA) select test cases at the "edges", and thus exercises bounding values.
- BVA leads to the selection of test cases at the "edges" of the class.



Boundary Value Analysis

- Guidelines of designing test cases
  - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b, just above and below a and b, respectively
  - If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

Example

- Input Data variable is Employment Age, as used in a company database system. Employment age is defined as an integer variable, and has an acceptable range of 16 to 65 (years old).
- The two "edges" are 16 and 65.
  - Employment Age = 15, 16, 17, and 64, 65, and 66.

Comparison Testing

- Used when the reliability of software is absolutely critical.
- Multiple and independent versions of software is developed for critical applications, even when only a single version will be used in the delivered computer-based system
- Each version is tested with the same test data to ensure that all provide identical output.

Comparison Testing

- All the versions are executed in parallel with a real-time comparison of results to ensure consistency.
  - If the output from each version is the same, then it is assumed that all implementations are correct.
  - If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference.

Automated Testing Tools

- Code Auditors
- Assertion Processors
- Test File Generators
- Test Data Generators
- Test Verifiers
- Output Comparators

Code Auditors

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

Assertion Processors

- These pre-processors / postprocessors systems are employed to tell whether programmer-supplied claims, called assertions, about a program's behavior are actually met during real program executions.

Test File Generators

- These processors generate, and fill with predetermined values, typical input files for programs that are undergoing testing.

Test Data Generators

- These automated analysis systems assist a user in selecting test data that make a program behave in a particular fashion.

Test Verifiers

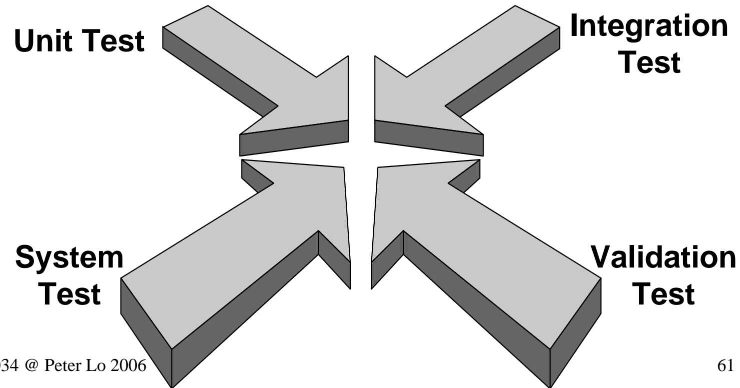
- These tools measure internal test coverage, often expressed in terms that are related to the control structure of the test object, and report the coverage value to the quality assurance expert

Output Comparators

- This tool makes it possible to compare one set of outputs from a program with another (previously archived) set to determine the difference between them.

Testing Strategy

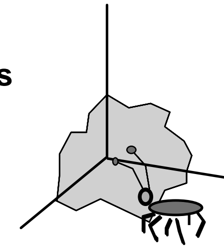
- A testing strategy must always incorporate test planning, test case design, text execution, and the resultant data collection and evaluation



Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer



OBJECTIVE	**to uncover errors**
CRITERIA	**in a complete manner**
CONSTRAINT	**with a minimum of effort and time**

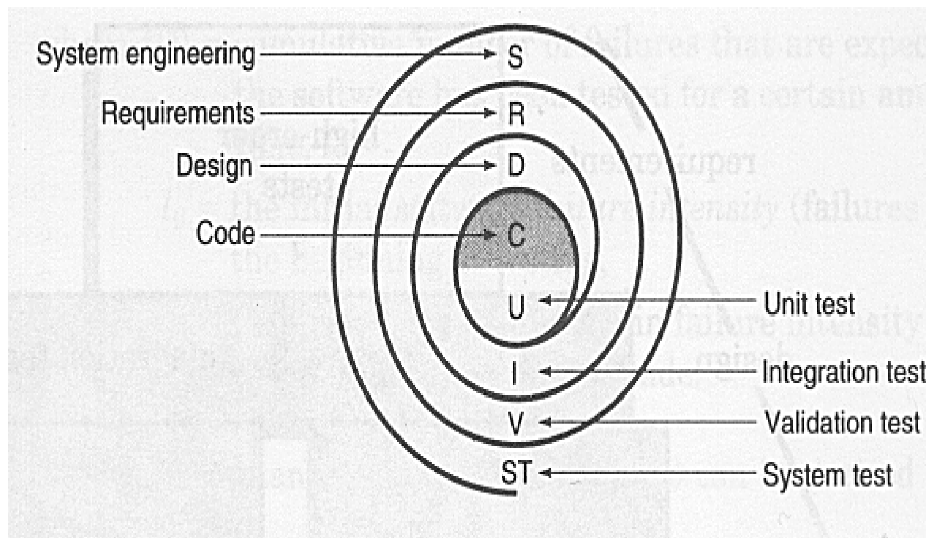
Verification and Validation

- **Verification** – Set of activities that ensure that software correctly implements a specific function
  - Are we building the project right?
- **Validation** – Set of activities that ensure that the software that has been built is traceable to customer requirements
  - Are we building the right product?

Generic Characteristics of Software Testing Strategies

- Testing beings at the module level and works toward the integration of the entire system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the software developer and an Independent Test Group (ITG)
- Debugging must be accommodated in any testing strategy

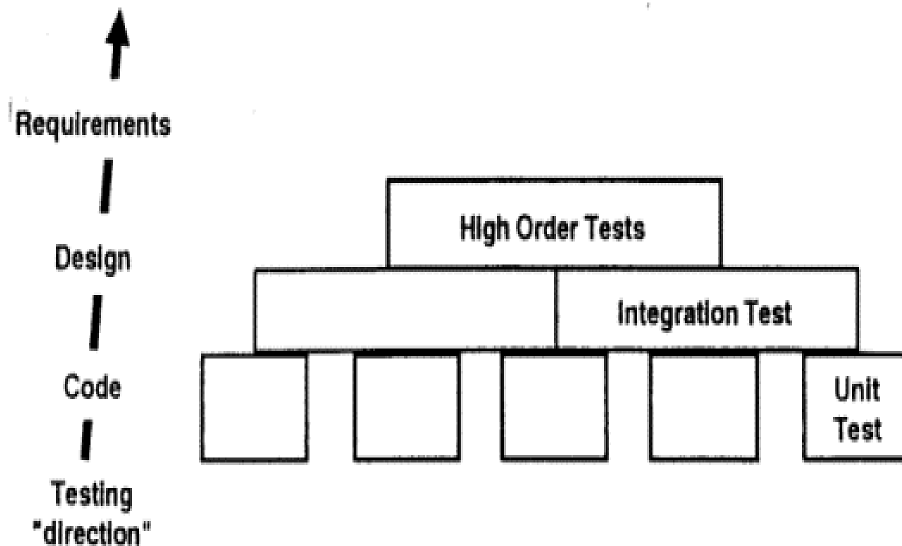
Software Testing Strategy



Software Testing Strategy

- A strategy for software testing moves outward along the spiral.
  - **Unit Testing**: Concentrates on each unit of the software as implemented in the source code.
  - **Integration Testing**: Focus on the design and the construction of the software architecture.
  - **Validation Testing**: Requirements established as part of software requirement analysis are validated against the software that has been constructed.
  - **System Testing**: The software and other system elements are tested as a whole.

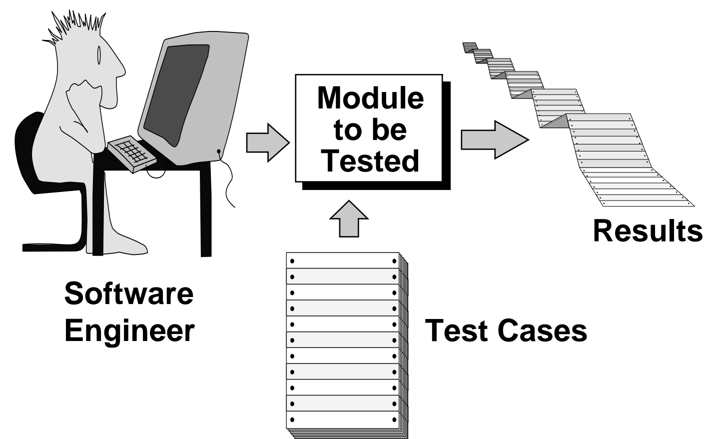
Testing Direction



Software Testing Direction

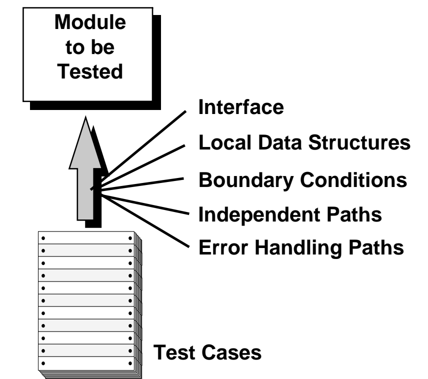
- Unit Tests
  - Focuses on each module and makes heavy use of white box testing
- Integration Tests
  - Focuses on the design and construction of software architecture; black box testing is most prevalent with limited white box testing.
- High-order Tests
  - Conduct validation and system tests. Makes use of black box testing exclusively (such as Validation Test, System Test, Alpha and Beta Test, and other Specialized Testing).

Unit Testing

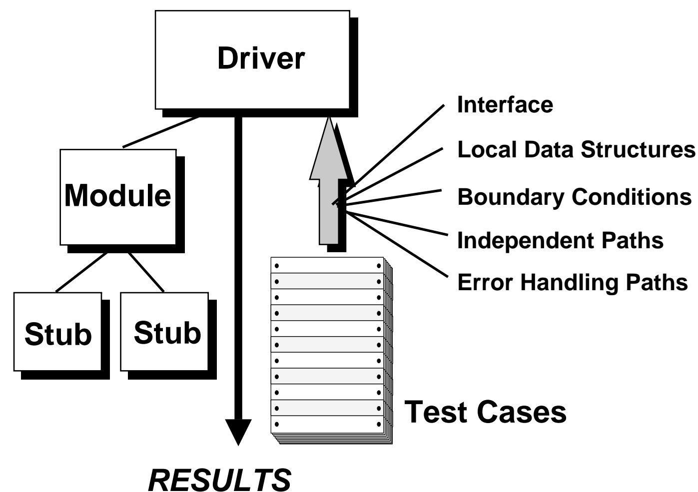


Unit Testing

- Unit testing focuses on the results from coding.
- Each module is tested in turn and in isolation from the others.
- Using the detail design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- Uses white-box techniques.



Unit Test Environment



Unit Testing Procedures

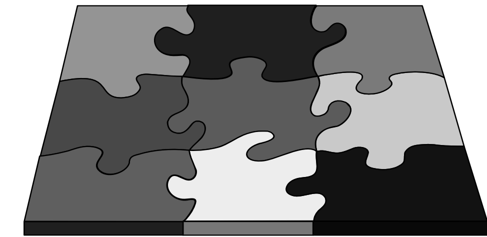
- Module is not a stand-alone program, driver & stub software must be developed for each unit test.
- A driver is a program that accepts test case data, passes such data to the module, and prints the relevant results.
- Stubs serve to replace modules that are subordinate the module to be tested.
- A stub or "dummy subprogram" uses the subordinate module's interface, may do nominal data manipulation, prints verification of entry, and returns.
- Drivers and stubs also represent overhead

Unit Test Considerations

- The module interface is tested to ensure that information properly flows into and out of the program unit under test.
- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- And finally, all error-handling paths are tested.

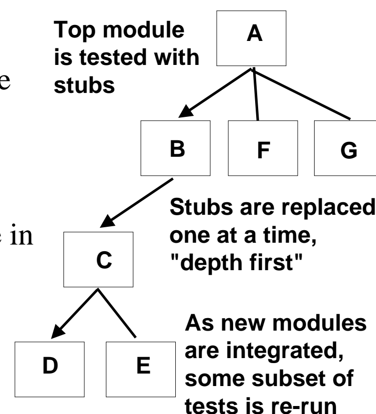
Integration Testing

- A technique for constructing the program structure while at the same time conducting tests to uncover tests to uncover errors associated with interfacing
- Objective is combining unit-tested modules and build a program structure that has been dictated by design.
- Integration testing should be done incrementally.
- It can be done top-down, bottom-up or in bi-directional.



Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with main control module.
- Subordinate modules are incorporated into the structure in either a depth-first or breadth-first manner.

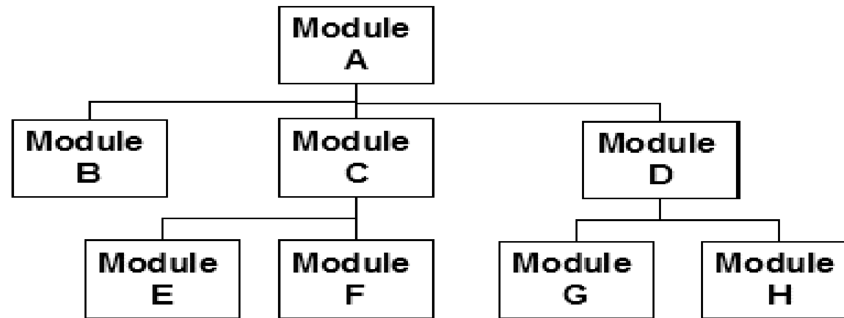


Procedure

- The main control module is used as a test driver and stubs are substituted for all modules directly subordinate to the main control module
- Subordinate stubs are replaced one at a time with actual modules
- Tests are conducted as each module is integrated
- On the completion of each set of tests, another stub is replaced with the real module
- Regression testing may be conducted to ensure that new errors have not been introduced

Example

- For the program structure, the following test cases may be derived if top-down integration is conducted:
  - Test case 1: Modules A and B are integrated
  - Test case 2: Modules A, B and C are integrated
  - Test case 3: Modules A., B, C and D are integrated (etc.)

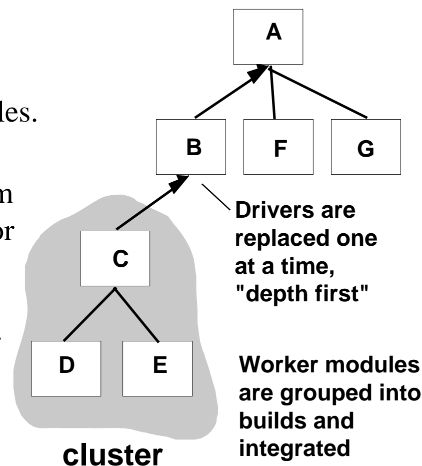


Problem of Top-Down Testing

- Inadequate testing at upper levels when data flows at low levels in the hierarchy are required
- Delay many test until stubs are replaced with actual modules; but this can lead to difficulties in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach
- Develop stubs that perform limited functions that simulate the actual module; but this can lead to significant overhead

Bottom-Up Integration Testing

- This integration process begins construction and testing with atomic modules.
- Because modules are integrated from the bottom up, processing required for modules subordinate to a given level is always available and the need for stubs is eliminated.

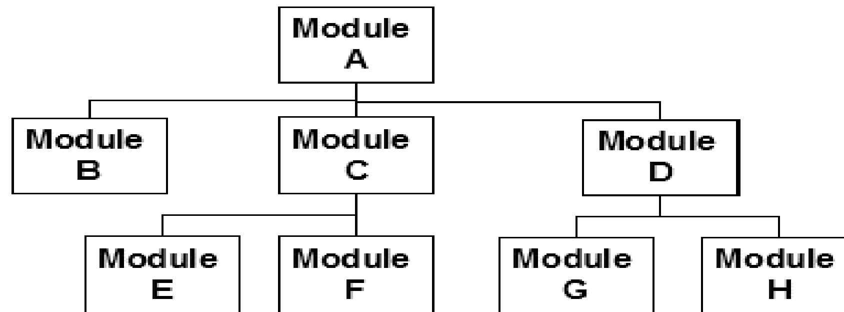


Procedure

- Low-level modules are combined into clusters that perform a specific software sub-function
- A driver is written to coordinate test case input and output
- The cluster is tested
- Drivers are removed and clusters are combined moving upward in the program structure

Example

- Test case 1: Modules E and F are integrated
- Test case 2: Modules E, F and G are integrated
- Test case 3: Modules E., F, G and H are integrated
- Test case 4: Modules E., F, G, H and C are integrated (etc.)
- Drivers are used all round.



Validation Testing

- Ensuring software functions can be reasonably expected by the customer.
- Achieve through a series of black tests that demonstrate conformity with requirements.
- A test plan outlines the classes of tests to be conducted, and a test procedure defines specific test cases that will be used in an attempt to uncover errors in conformity with requirements.
- Validation testing begins, driven by the validation criteria that were elicited during requirement capture.
- A series of acceptance tests are conducted with the end users

Validation Testing

- After the developers and the independent testers have satisfied, the end users carry out acceptance tests, which are part of the validation testing.
- These occur in two stages.
  - Alpha testing
    - Is conducted at the developer's site by a customer
    - The developer would supervise
    - Is conducted in a controlled environment
  - Beta testing
    - Is conducted at one or more customer sites by the end user of the software
    - The developer is generally not present
    - Is conducted in a "live" environment

System Testing

- System testing is a series of different tests whose primary purpose is to fully exercise the computer-based system.
- System testing focuses on those from system engineering.
- Test the entire computer-based system.
- One main concern is the interfaces between software, hardware and human components.
- Kind of System Testing
    - Recovery
    - Security
    - Stress
    - Performance

Recovery Testing

- A system test that forces software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic, re-initialization, check-pointing mechanisms, data recovery, and restart are each evaluated for correctness.
- If recovery is manual, the mean time to repair is evaluated to determine whether it is within acceptable limits.

Security Testing

- Security testing attempts to verify that protection mechanisms built into a system will in fact protect it from improper penetration.
- Particularly important to a computer-based system that manages sensitive information or is capable of causing actions that can improperly harm individuals when targeted.

Stress Testing

- Stress Testing is designed to confront programs with abnormal situations where unusual quantity frequency, or volume of resources are demanded
- A variation is called sensitivity testing;
  - Attempts to uncover data combinations within valid input classes that may cause instability or improper processing

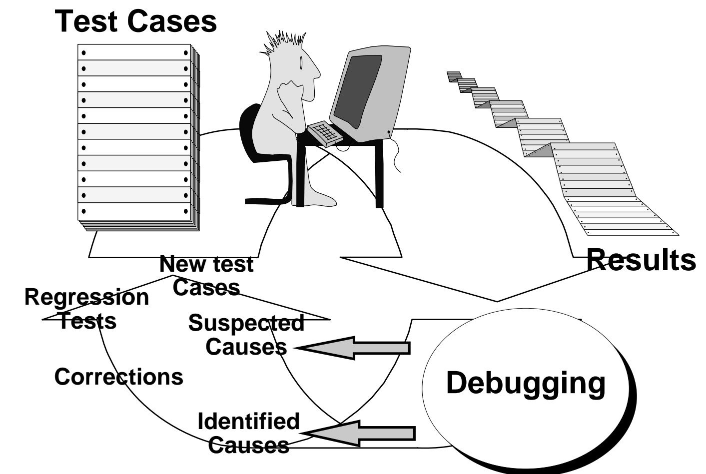
Performance Testing

- Test the run-time performance of software within the context of an integrated system.
- Extra instrumentation can monitor execution intervals, log events as they occur, and sample machine states on a regular basis
- Use of instrumentation can uncover situations that lead to degradation and possible system failure

Debugging

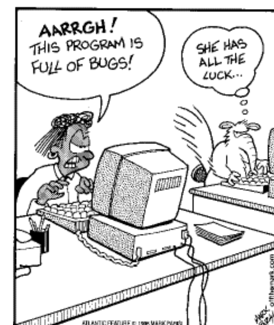
- Debugging will be the process that results in the removal of the error after the execution of a test case.
- Its objective is to remove the defects uncovered by tests.
- Because the cause may not be directly linked to the symptom, it may be necessary to enumerate hypotheses explicitly and then design new test cases to allow confirmation or rejection of the hypothesized cause.

The Debugging Process



What is Bug?

- A bug is a part of a program that, if executed in the right state, will cause the system to deviate from its specification (or cause the system to deviate from the behavior desired by the user).



Characteristics of Bugs

- The symptom and the cause may be geographically remote
- The symptom may disappear when another error is corrected
- The symptom may actually be caused by non-errors
- The symptom may be caused by a human error that is not easily traced
- It may be difficult to accurately reproduce input conditions
- The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably
- The symptom may be due to causes that are distributed across a number of tasks running on different processors

Debugging Techniques

- ❑ **Brute Force / Testing**
- ❑ **Backtracking**
- ❑ **Cause Elimination**

Debugging Approaches – Brute Force

- Probably the most common and least efficient method for isolating the cause of a software error.
- The program is loaded with run-time traces, and WRITE statements, and hopefully some information will be produced that will indicated a clue to the cause of an error.

Debugging Approaches – Backtracking

- Fairly common in small programs.
- Starting from where the symptom has been uncovered, backtrack manually until the site of the cause is found.
- Unfortunately, as the number of source code lines increases, the number of potential backward paths may become unmanageably large.

Debugging Approaches – Cause Elimination

- Data related to the error occurrence is organized to isolate potential causes.
- A "cause hypothesis" is devised and the above data are used to prove or disapprove the hypothesis.
- Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.
- If the initial tests indicate that a particular cause hypothesis shows promise, the data are refined in a attempt to isolate the bug.

The Debugging Outcome

- There are two possible outcomes of the debugging process:
  - The cause of the error will be found, corrected, and removed.
  - The cause of the error will not be found.
- In the case of the latter, the person performing debugging may suspect a cause, design a test case to help validate his or her suspicion, and work toward error correction in an iterative fashion.

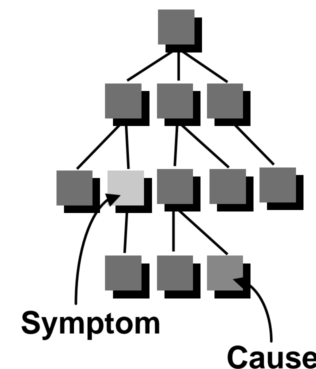
Finding and Fixing Bugs

- Keep an old version of the code which contains the bug and the input-output relation that demonstrates its presence. It allows you to review process of bug trapping and fixing. It also allows to go back to previous version if new version has problem.
- Record the time and date the bug was found, the author of the change as a comment in the new version.
- Record the location of the file containing the previous version of the code.

Difficulties in Debugging

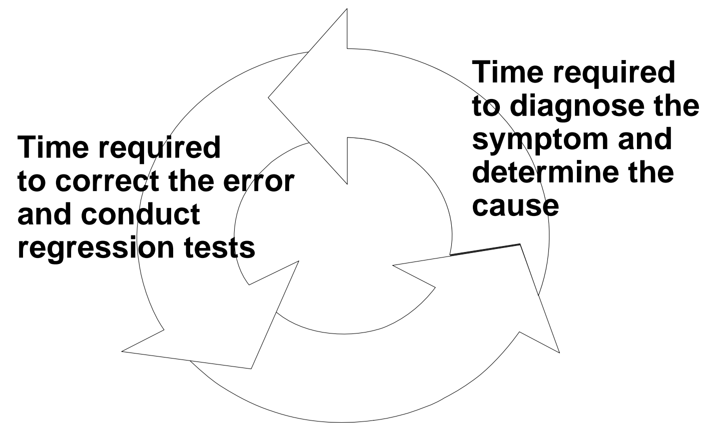
- Symptom and Cause may lie at locations in the code that are quite remote from one another.
- Symptom may disappear by the removal of another error, but only temporarily.
- Symptom may be caused not by an error but by misjudgment (e.g. round-off inaccuracies).
- Symptom may be caused by human error (e.g. incorrect test plans, misinterpretation of specification).
- Symptom may be difficult to reproduce.

Symptoms and Causes

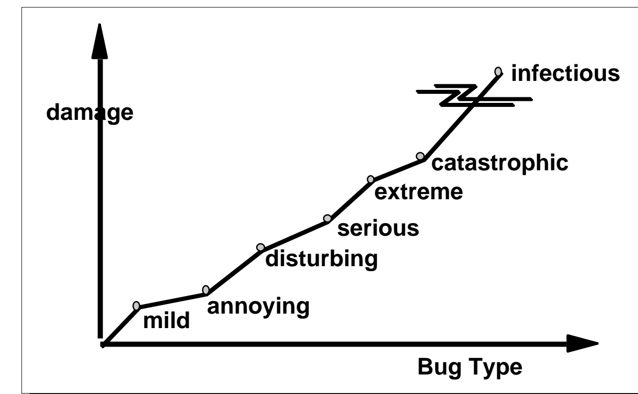


- Symptom and cause may be geographically separated
- Symptom may disappear when another problem is fixed
- Cause may be due to a combination of non-errors
- Cause may be due to a system or compiler error
- Cause may be due to assumptions that everyone believes
- Symptom may be intermittent

Debugging Effort



Consequences of Bugs



Bug Categories: **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

Debugging Tools

- Debugging compilers
- Dynamic debugging aides ("tracers")
- Automatic test case generators
- Memory dumps

Debugging: Final Thoughts

- Don't run off half-cocked, think about the symptom you're seeing.
- Use tools (e.g., dynamic debugger) to gain more insight.
- If at an impasse, get help from someone else.
- Be absolutely sure to conduct regression tests when you do "fix" the bug.

Software Maintenance

- Maintenance account for up to 60% effort put into software.
- Most Software has a long operational life.
- Many software systems were built under different technological constraints than hold now (e.g. scarce primary and secondary memory).
- Most software systems have had to migrate to new hardware and software environment.
- Not enough thought is put into re-analysis and re-design.

Maintenance Category

- **Corrective** – Remove defects.
- **Adaptive** – Adapt to new hardware or software environment as well as changes in requirements.
- **Perfective** - Incorporate new functions or improve existing ones.
- **Preventive** – Basic Preventive Maintenance Activities
  - Reverse Engineering
  - Re-Engineering

Reverse Engineering

- It is often that all the higher level knowledge of the software becomes unavailable.
- Current team of software engineers loses understanding of the high-level aspects of the old software and can only work with the code.
- Reverse Engineering tries to extract the architectural, data and procedural knowledge buried in code.
- Its goal is to produce a design-level representation from a code-level representation of the software.

Re-Engineering

- The main motivation of re-engineering is to improve maintainability.
- It redesign the processes in order to achieve dramatic improvements in performance and quality.