Lecture 10

- Software Testing Techniques

"How to find bugs in the software?"

Topics

- Software Testing
- White-Box Testing
- Black-Box Testing
- Testing Object-Oriented Applications
- Debugging

1. Software Testing

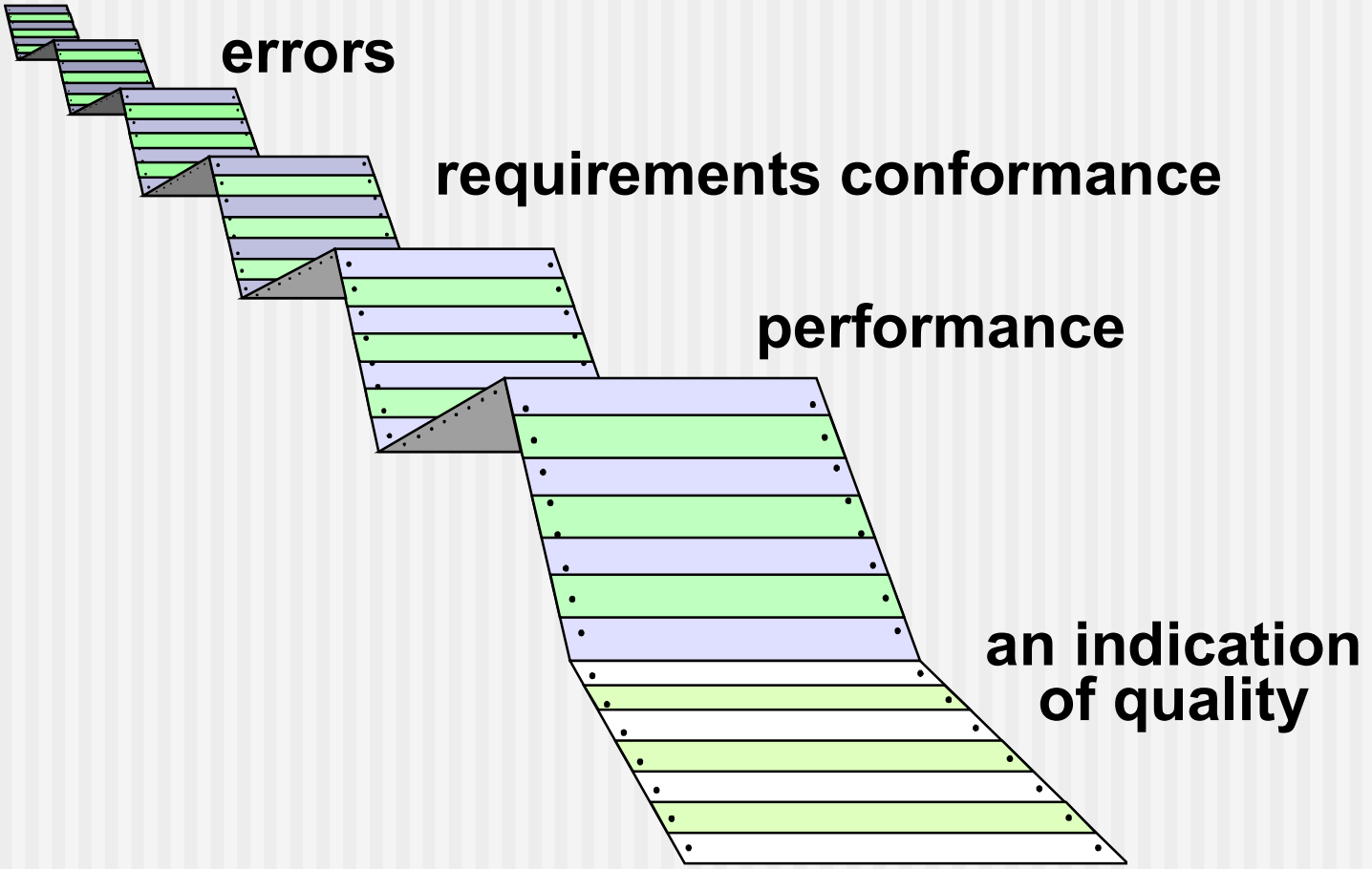
Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Testing

- Once source code has been generated, the software must be tested to uncover (and correct) as many errors as possible before delivery to customer
- The goal is to design a series of test cases that have a high likelihood of finding errors

What Testing Shows



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—of the design

What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

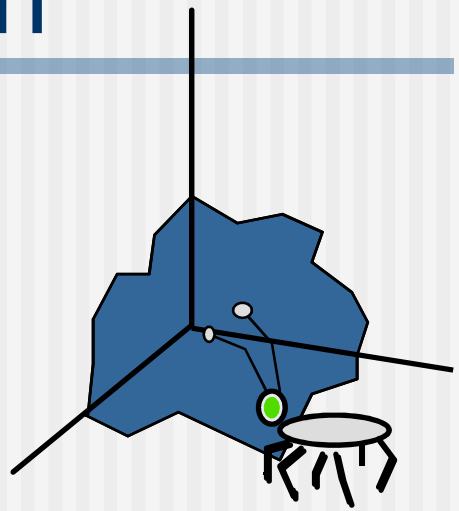
Internal and External Views

- Any engineered product (and most other things) can be tested in one of two ways:
  - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

Test Case Design

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

Boris Beizer

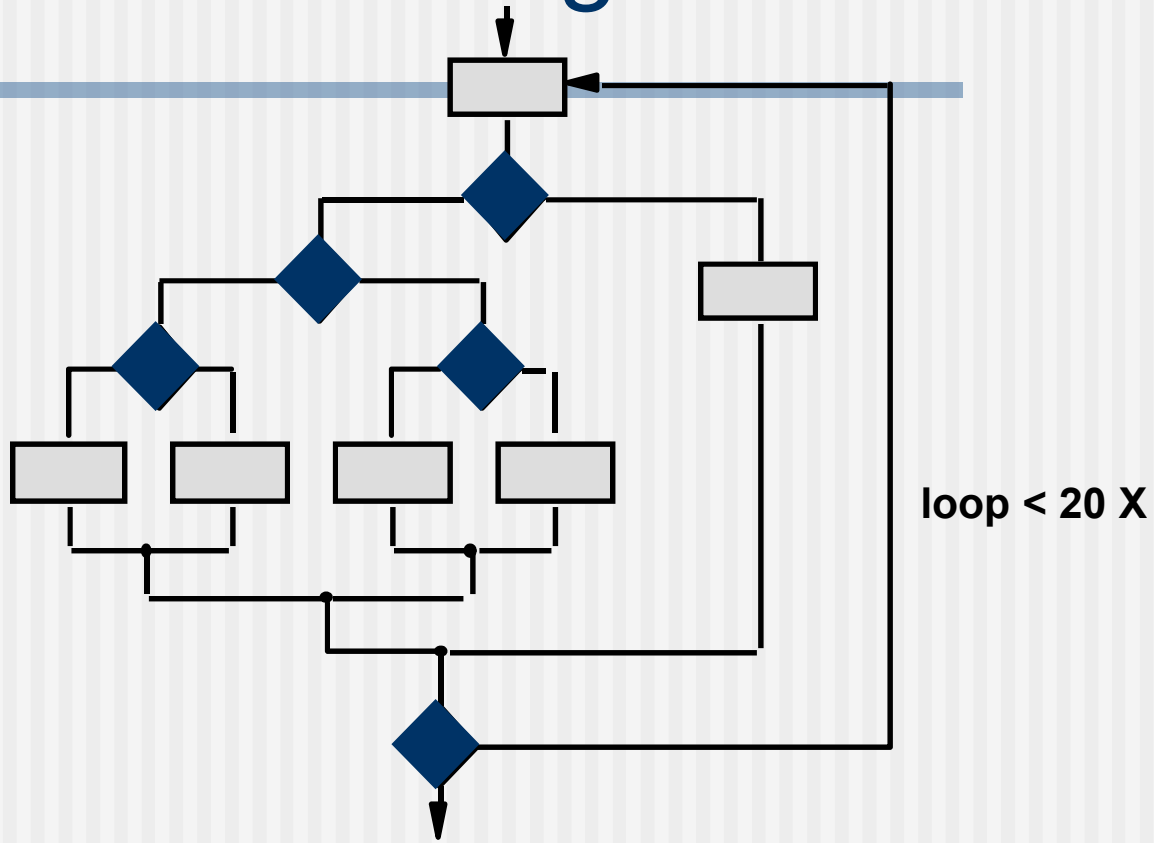


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

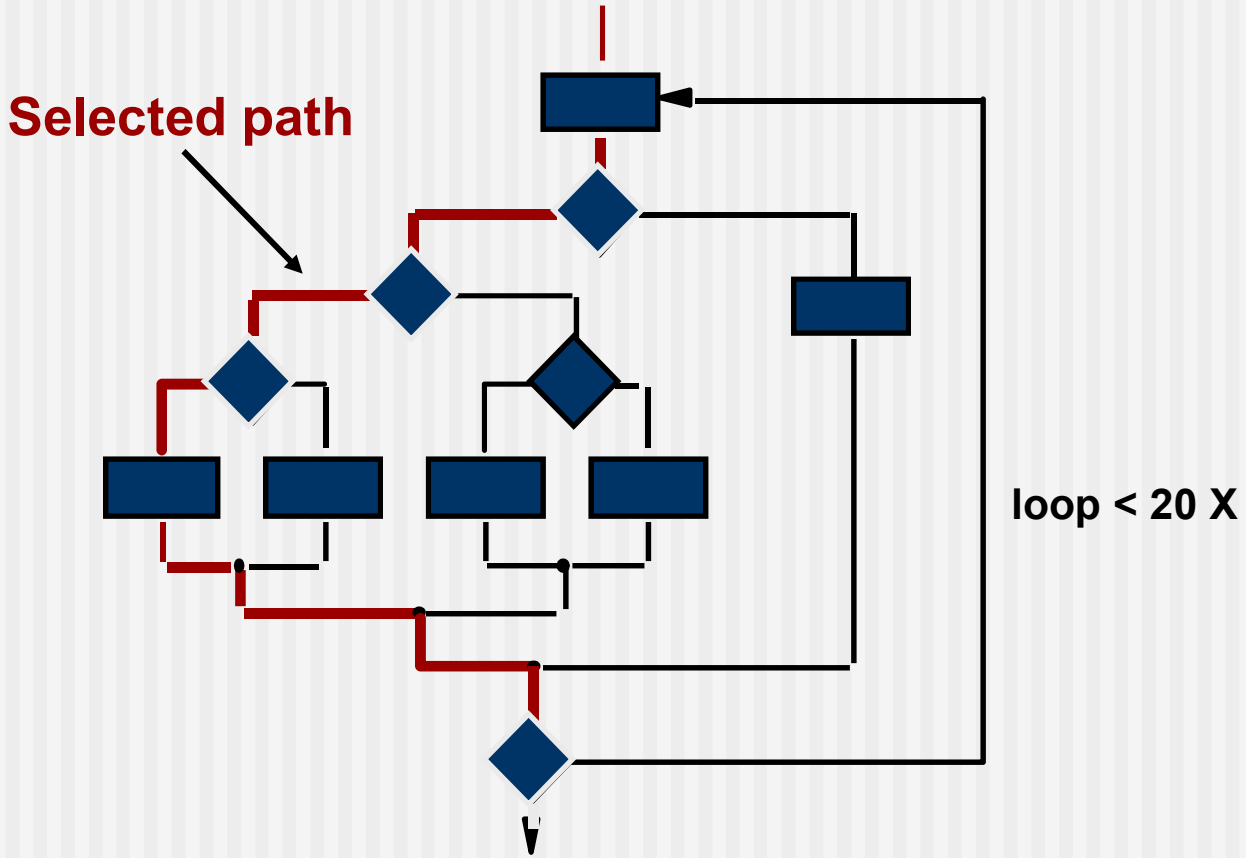
CONSTRAINT **with a minimum of effort and time**

Exhaustive Testing

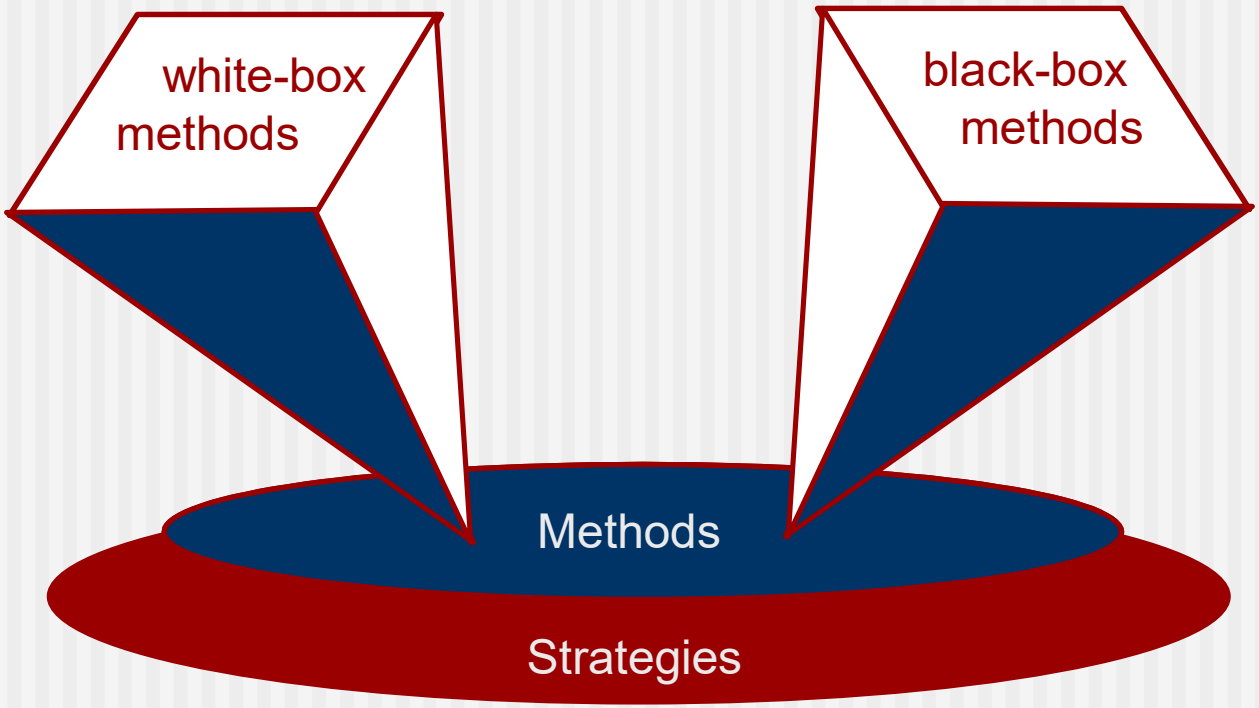


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing

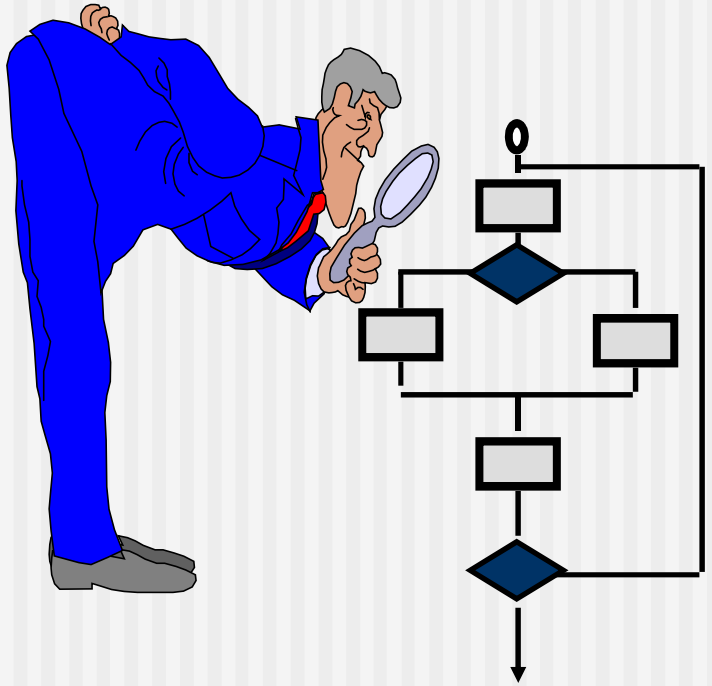


Software Testing



2. White-Box Testing

White-Box Testing

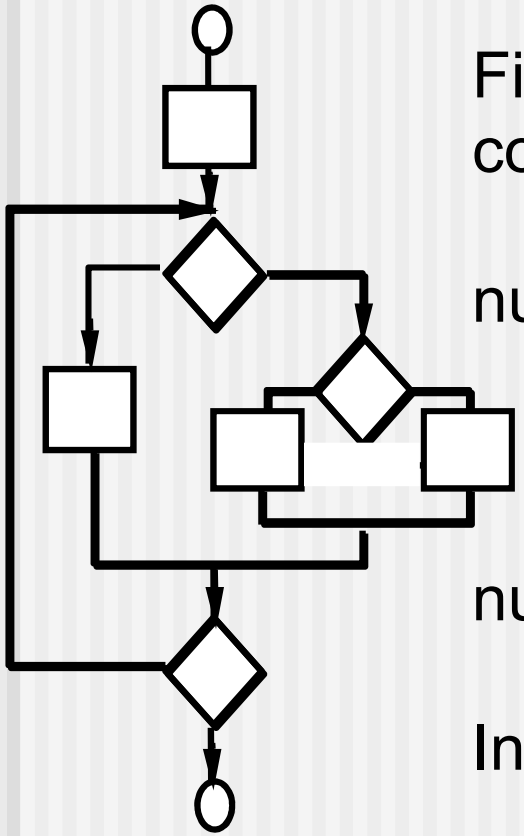


... our goal is to ensure that all statements and conditions have been executed at least once ...

Why Cover?

- **logic errors and incorrect assumptions are inversely proportional to a path's execution probability**
- **we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive**
- **typographical errors are random; it's likely that untested paths will contain some**

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

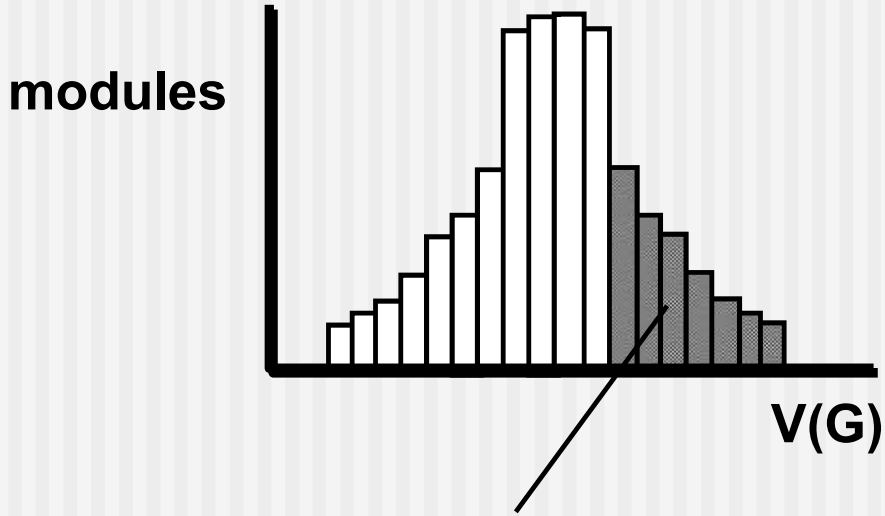
or

number of enclosed areas + 1

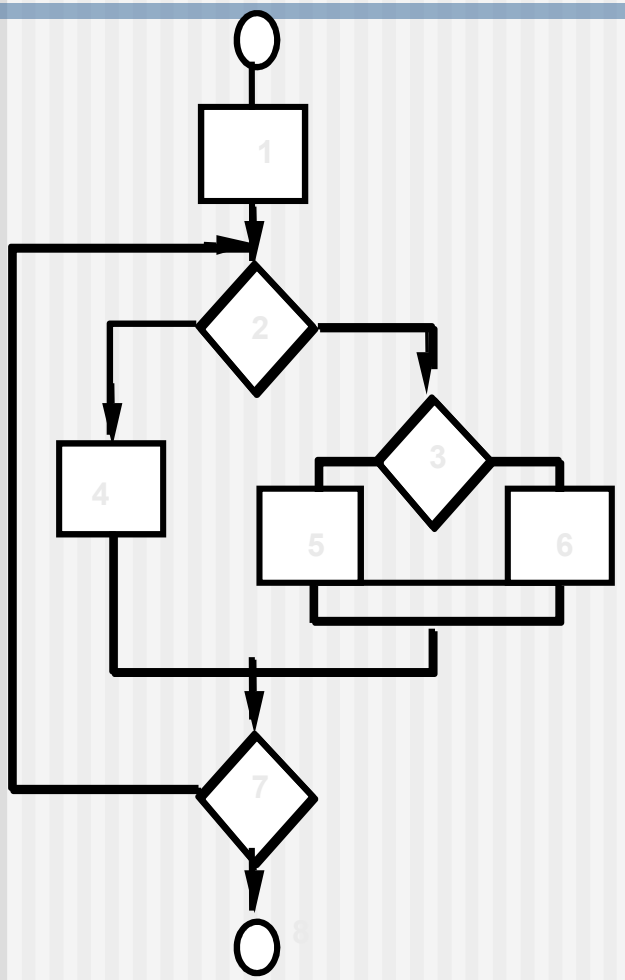
In this case, V(G) = 4

Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

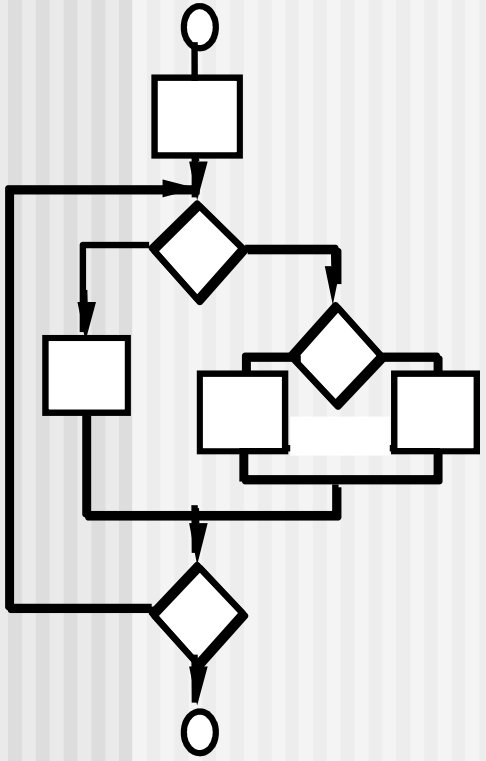
Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

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

Deriving Test Cases

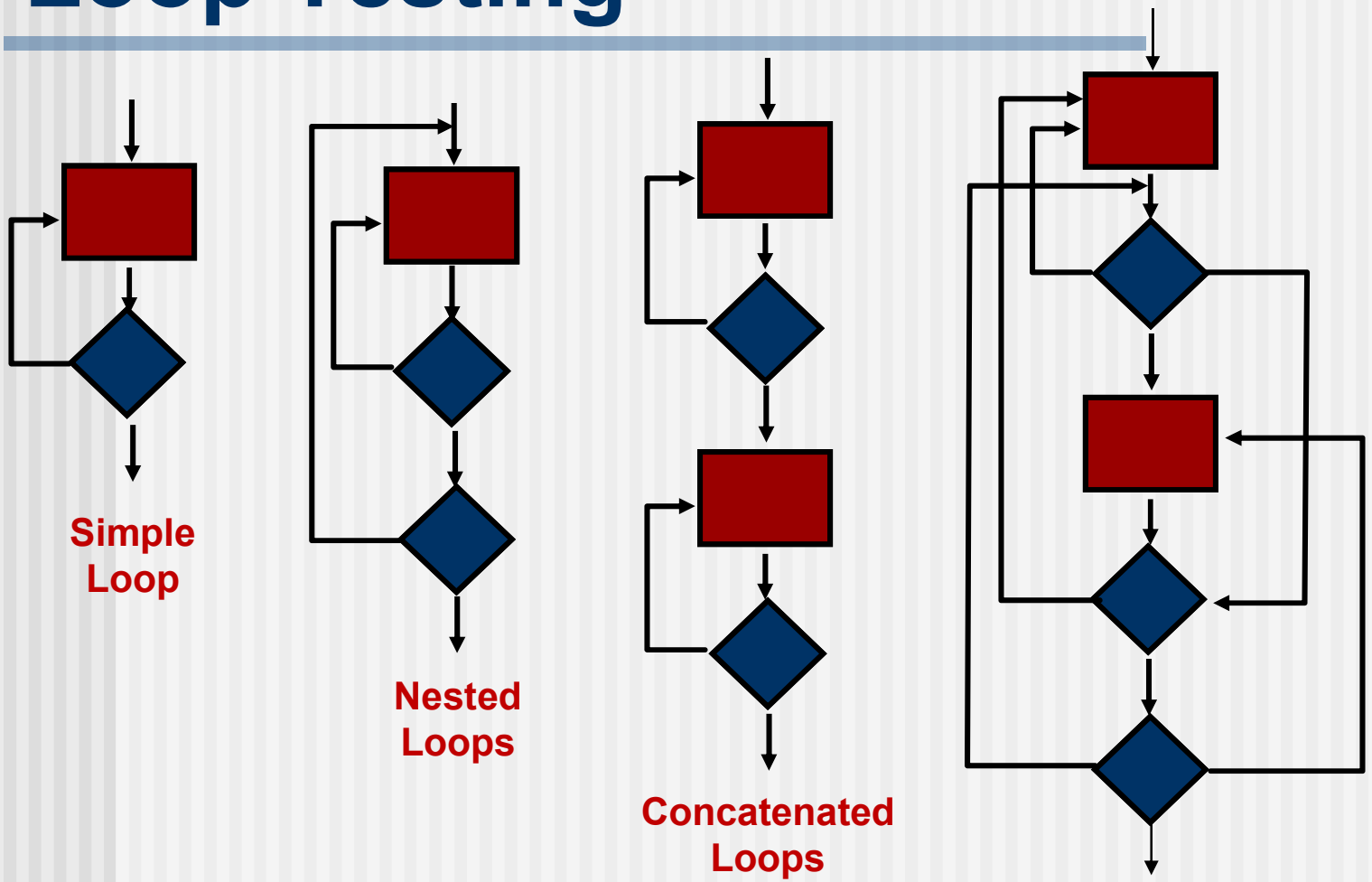
- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

Control Structure Testing

- **Condition Testing** — A test case design method that exercises the logical conditions contained in a program module.
- **Data Flow Testing** — Selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop Testing

This class of loop should be redesigned to reflect the use of the structured programming constructs.



Loop Testing: Simple Loops

- Skip the loop entirely.
- Only one pass through the loop.
- Two passes through the loop.
- ***m*** passes through the loop ***m*** < ***n***.
- (***n*** -1), ***n***, and (***n*** +1) passes through the loop, where ***n*** is the maximum number of allowable passes.

Loop Testing: Nested Loops

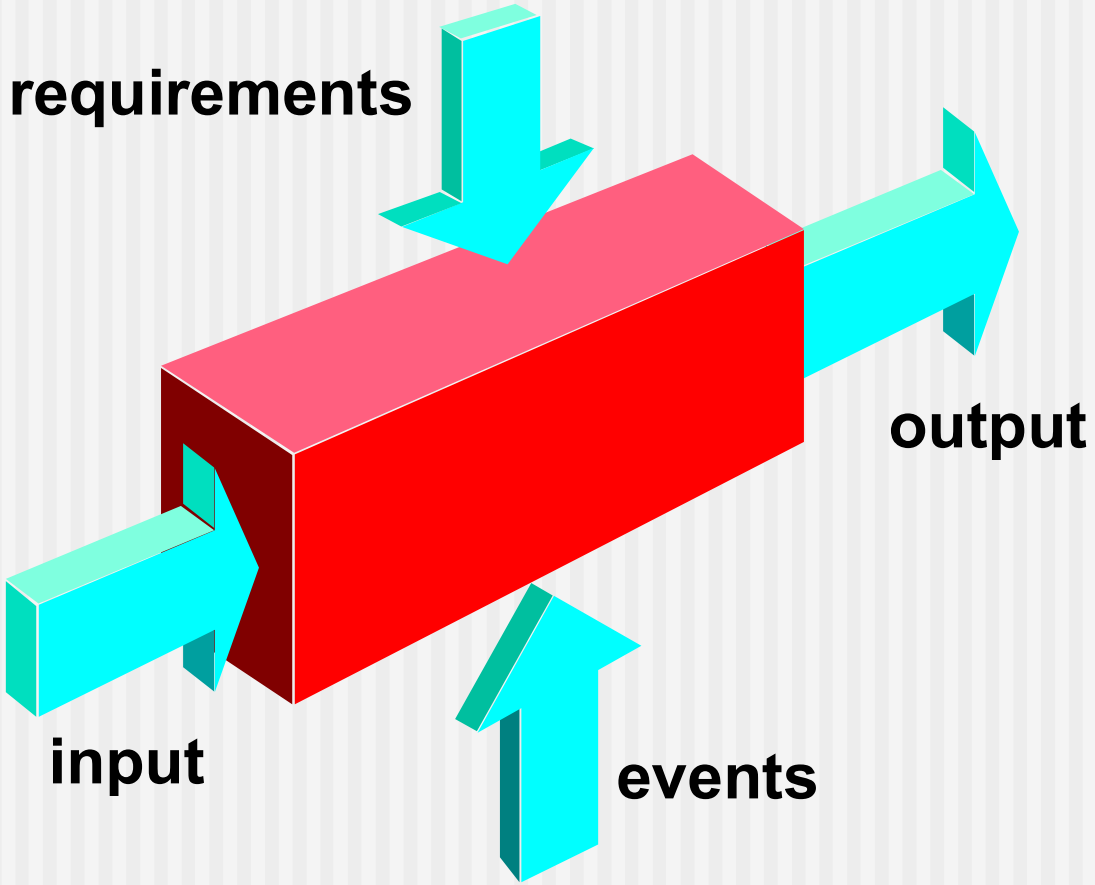
- Start at the innermost loop; set all outer loops to their minimum iteration parameter values.
- Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.
- Move out one loop and set it up as in step 2, holding all other loops at typical values; continue this step until the outermost loop has been tested.

Loop Testing: Concatenated Loops

- If the loops are independent of one another, then treat each as a simple loop; else treat as nested loops.
- For example, the final loop counter value of loop 1 is used to initialize loop 2.

3. Black-Box Testing

Black-Box Testing



Black-Box Testing

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

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
- Behavior or performance errors
- Initialization or termination errors

Black-Box Testing Methods

- Graph-based testing methods
  - Based on the understanding of the objects that are modeled in software and the relationships that connect these objects
  - Tests verify that all objects have the expected relationship to one another
- Equivalence partitioning
  - Divides the input domain of a program into classes of data from which test cases can be derived
- Boundary value analysis
  - Selection of test cases that exercise bounding values

4. Testing Object-Oriented Applications

OO Testing

- To adequately test OO systems, three things must be done:
  - the definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models
  - the strategy for unit and integration testing must change significantly, and
  - the design of test cases must account for the unique characteristics of OO software.

‘Testing’ OO Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side affects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Correctness of OO Models

- During analysis and design, semantic correctness can be asesssed based on the model's conformance to the real world problem domain.
- If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is semantically correct.
- To determine whether the model does, in fact, reflect real world requirements, it should be presented to problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity.
- Class relationships (instance connections) are evaluated to determine whether they accurately reflect real-world object connections.

Class Model Consistency

- Revisit the CRC model and the object-relationship model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- Determine whether widely requested responsibilities might be combined into a single responsibility.

OO Testing Methods

Berard [Ber93] proposes the following approach:

1. **Each test case should be uniquely identified and should be explicitly associated with the class to be tested,**
2. **The purpose of the test should be stated,**
3. **A list of testing steps should be developed for each test and should contain [BER94]:**
    a. **a list of specified states for the object that is to be tested**
    b. **a list of messages and operations that will be exercised as a consequence of the test**
    c. **a list of exceptions that may occur as the object is tested**
    d. **a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)**
    e. **supplementary information that will aid in understanding or implementing the test.**

OO Testing Methods

- Fault-based testing
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

Random Testing

- Identify operations applicable to a class.
- Define constraints on their use.
- Identify a minimum test sequence: an operation sequence that defines the minimum life history of the class (object).
- Generate a variety of random (but valid) test sequences: exercise other (more complex) class instance life histories.

Partition Testing

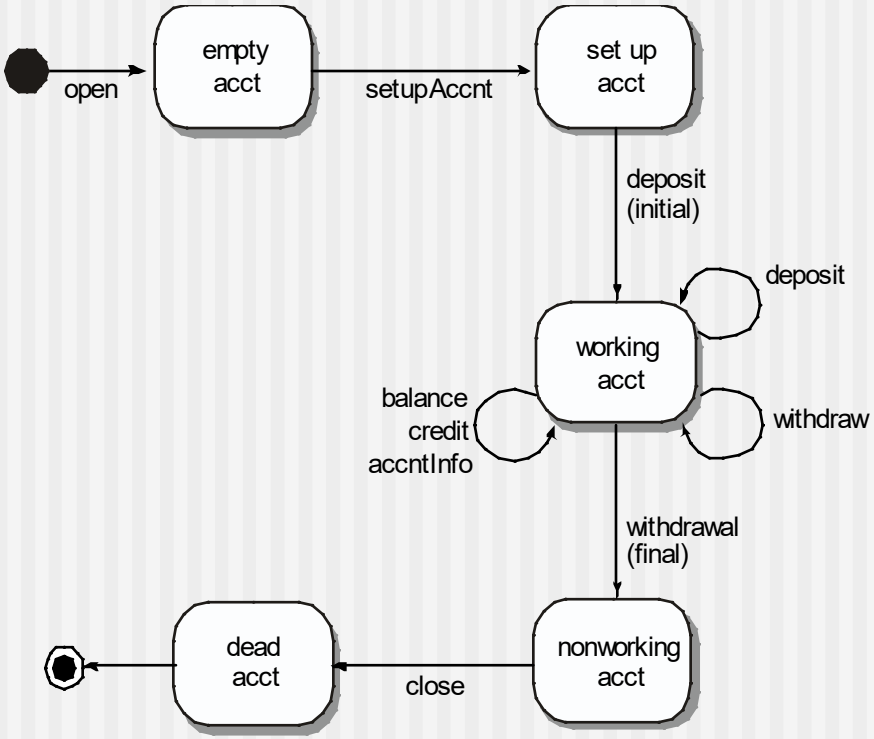
- Reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software:
  - **State-based Partitioning** categorizes and tests operations based on their ability to change the state of a class.
  - **Attribute-based Partitioning** categorizes and tests operations based on the attributes that they use.
  - **Category-based Partitioning** categorizes and tests operations based on the generic function each performs.

Inter-Class Testing

- For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
- For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
- For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
- For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence.

Behavior Testing

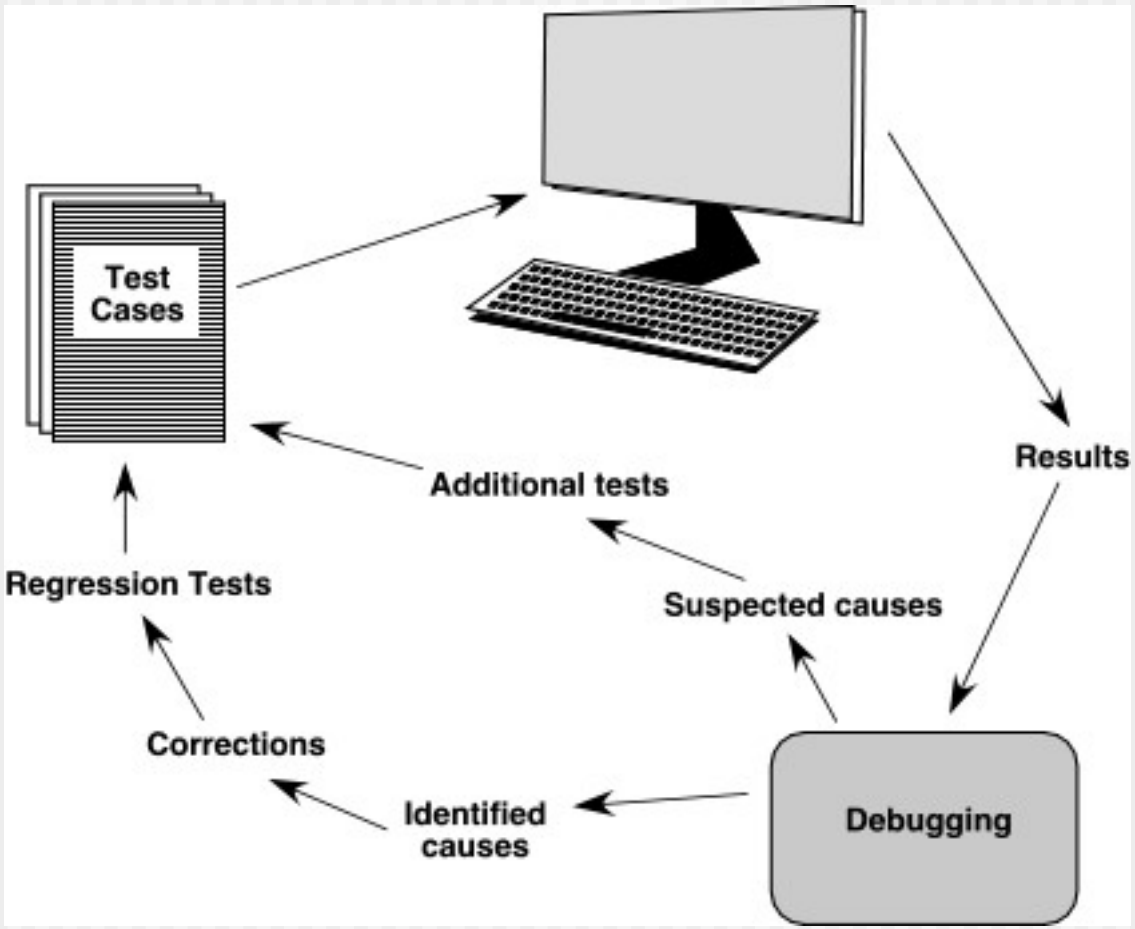
The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states



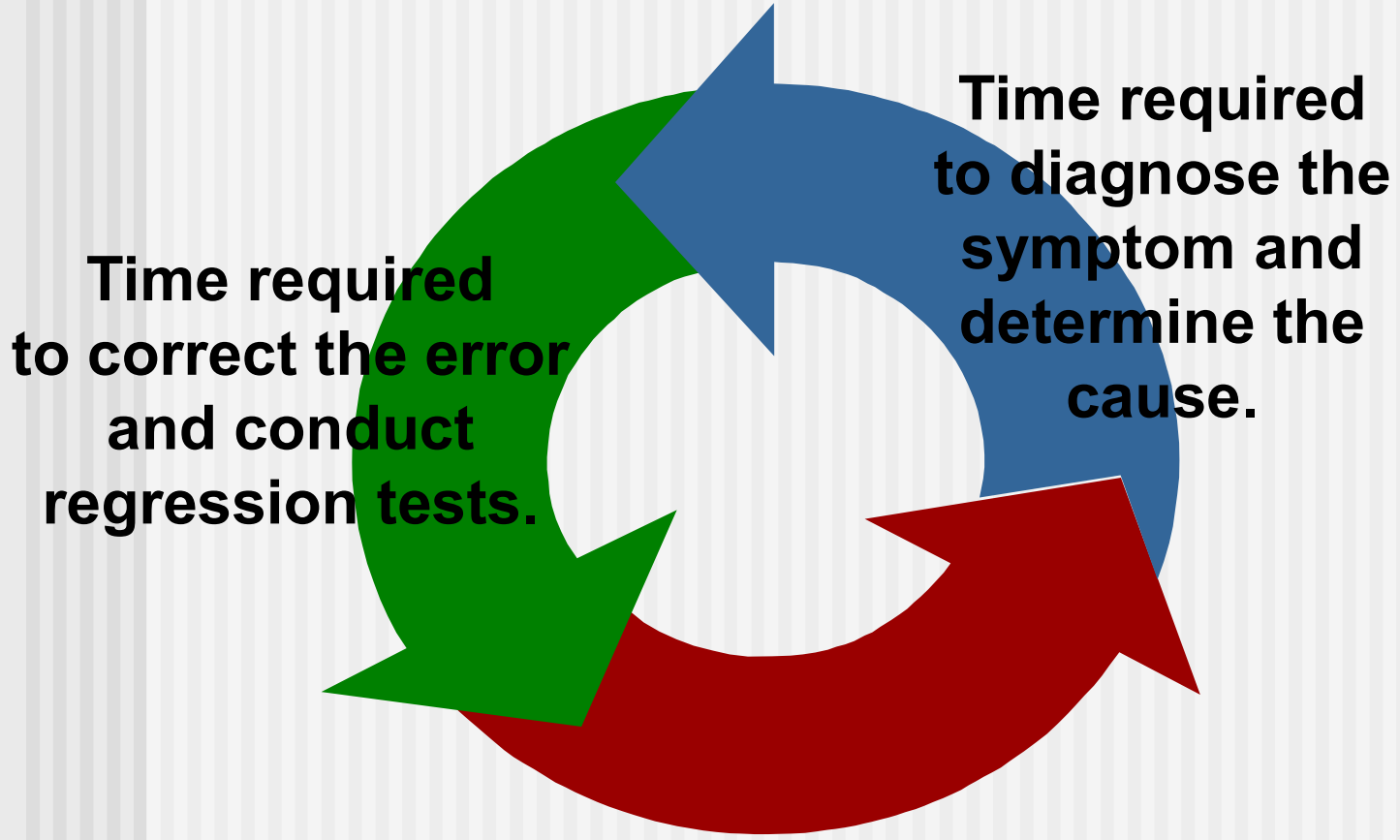
5. Debugging

Debugging: A Diagnostic Process

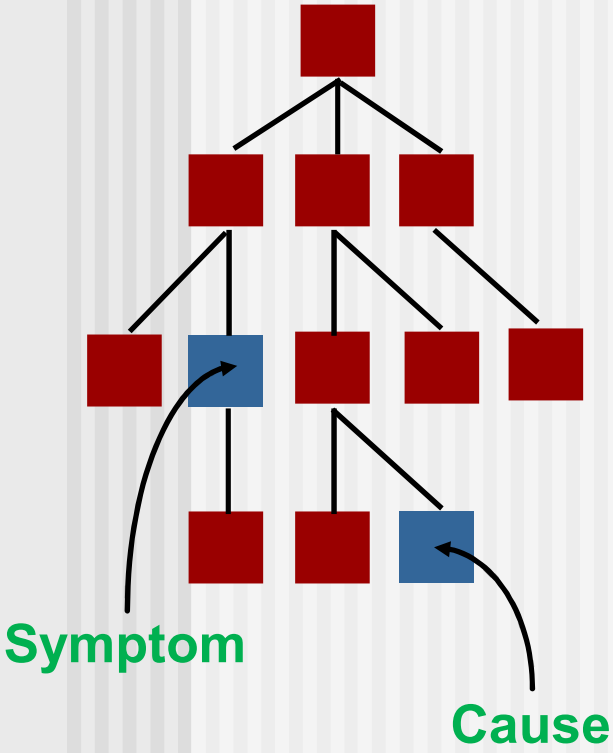
The Debugging Process



Debugging Effort

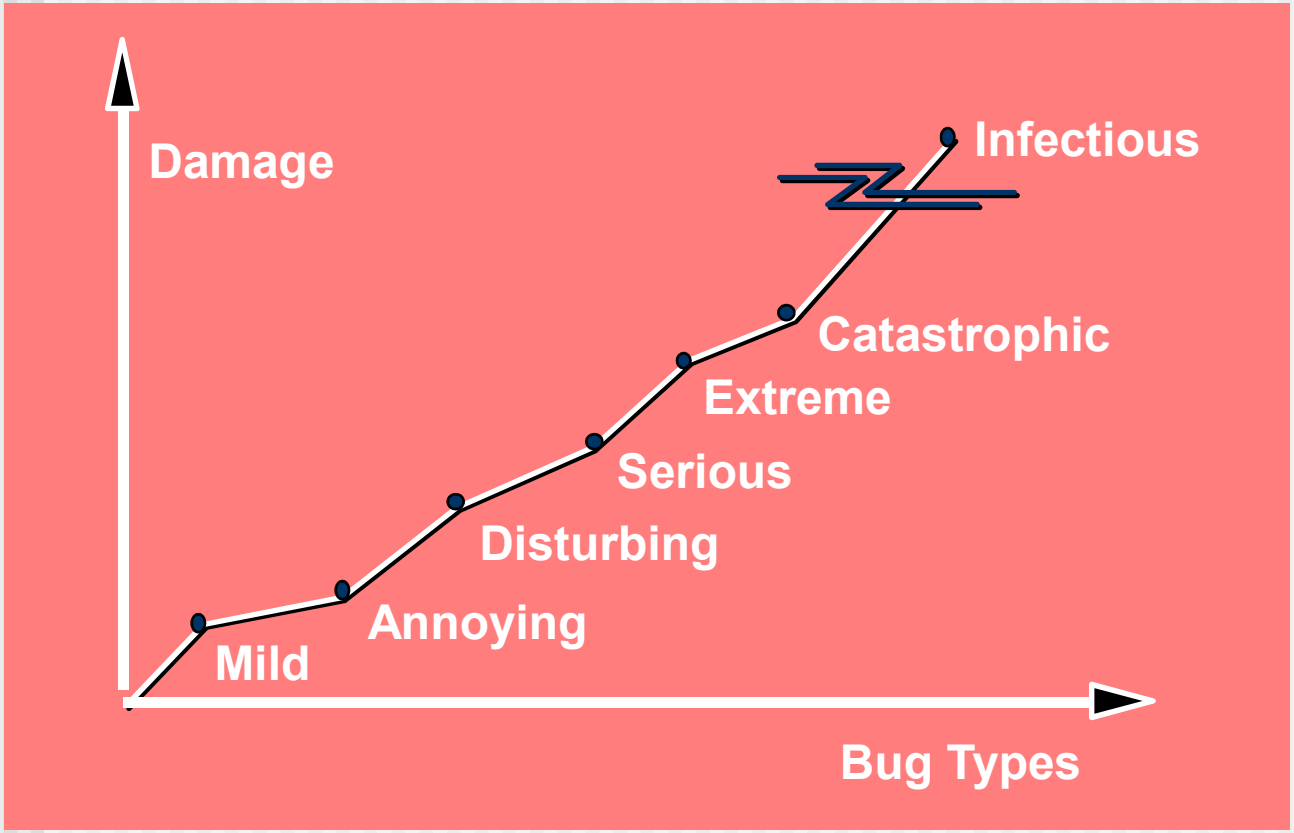


Symptoms and Causes



- 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.

Consequences of Bugs



Debugging Techniques

- brute force / testing
- backtracking
- induction
- deduction

Correcting the Error

- **Is the cause of the bug reproduced in another part of the program?** In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- **What "next bug" might be introduced by the fix I'm about to make?** Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
- **What could we have done to prevent this bug in the first place?** This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

Final Thoughts

- *Think* -- before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects