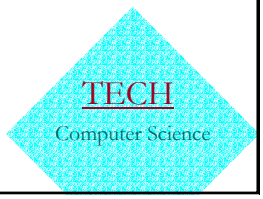
CH08: Testing the Programs

- Testing components individually and
- then integrating them to check the interfaces
- * Software Faults and Failures
- * Testing Issues
- * Unit Testing
- * Integration Testing



More Tests

- Testing Object-Orientated Systems
- Testing Planning
- Automated Testing Tools
- When to Stop Testing

Software Faults and Failures

- Software has failed?
  - **Software does not do what the requirements describe.**
- Reasons for software failure:
  - **Specifications do not state exactly what the customer wants or needs: wrong or missing requirements**
  - **Requirement is impossible to implement**
  - **Faults in system design, program design, and/or program code**

Testing to:

- demonstrate the existence of a fault
- The goal is to discover faults,
- Testing is successful only when a fault is discovered or
- a failure occurs as a result of our testing procedures

Fault Identification and Correction

- **Fault Identification** is the process of determining what fault or faults cased the failure.
- **Fault Correction or Removal** is the process of making changes to the system so that the faults are removed.

Types of Faults

- Knowing what kind of faults we are seeking
- Anything can go wrong:
  - **algorithmic, syntax, precision, documentation, stress or overload, capacity or boundary**
  - **timing or coordination, throughput or performance, recovery,**
  - **hardware and system software, standards and procedures**

Algorithmic fault:
logic is wrong, wrong processing steps

- Program review (reading through the program) to find this type of fault
- Typical algorithmic faults:
  - **branching too soon, or too late**
  - **testing for the wrong condition**
  - **forgetting to initialize variables or set loop invariant**
  - **forgetting to test for a particular condition**
  - **comparing variables of inappropriate data types**

Syntax faults

- a missing comma, or a O or 0.
- Compilers will catch most of the syntax faults
- But don't count on it
- Know the syntax of your programming language

Computation and precision faults

- formula is not correctly implemented.
- Do not understand the order of operations
  - **Q: y = a*x^2*b + c**
  - **A: y = a*(x^2)*b + c**
  - **B: y = ((a*x)^2)*b + c**
  - **C: y = a*(x^(2*b)) + c**
  - **D: y = a*x^2*(b+c)**
- Precision faults: unexpected truncated

More Faults

- Documentation faults: documentation does not match what the program actually does
- Stress or overload faults: exceeding maximum buffer size
- Capacity faults: more then the systems can handle
- Timing or Coordination faults: does not meet real-time requirement, out of sequence

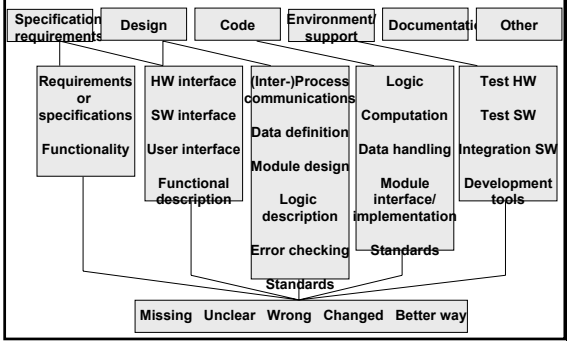
Still more faults

- Throughput or performance faults: could not finish the job within time limit
  - **total throughput - in and out, or response time to users**
- Recovery faults: could not get back to normal after system failure
- hardware and system faults: hardware malfunction, OS crashes.
- Standard and procedure faults: did not follow standard or processs.

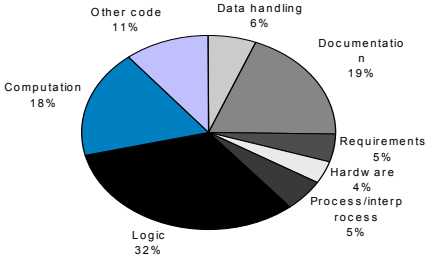
Orthogonal Defect Classification

- Categorize and track the types of faults we find to help direct our testing efforts
- (classification scheme is orthogonal if any item being classified belongs to exactly one category.)
- IBM Orthogonal defect classification (Fault type):
  - **Function, Interface, Checking, Assignment, Timing/serialization, Build/package/merge, Documentation, and Algorithm**

Hewlett-Packard Fault Classification



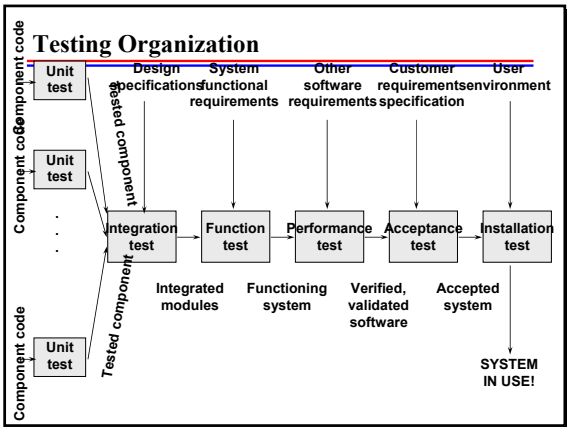
Faults for one HP division



Testing Issues

- Test Organization
- Attitudes Toward Testing
- Who Performs the Test?
- Views of the Test Objects

Testing Organization



Tests

- **Module, Component, or unit testing**: verifies functions of components, under controlled environment.
- **Integration test**: verifying components work together as an integrated system
- **Functional test**: determine if the integrated system perform the functions as stated in the requirements.

More tests

- **Performance test**: determine if the integrated system running under customer's actual working environment meet the response time, throughput, and capacity requirements.
- **Acceptance test**: conduct at under customer's supervision, verify if the system meet the customer's requirements (or satisfactions).

Last test

- **Installation test**: the system is installed in the customer's working environment, and test the installation is successful.

Attitudes Toward Testing

- New programmers are not accustomed to viewing testing as a discovery process.
- We take pride on what we developed. We defend ourselves "it is not my fault!"
- Conflict between tester and developer.
- Testers try to find faults, developers try to take pride.
- "Hurt feelings and bruised egos have no place in the development process as faults are discovered."

Who Performs the Test?

- Unit testing and integration testing are usually done by development teams.
- The rest of the testings called "system testings" are usually done be test teams.

Views of the Test Objects

- **Closed box or black box** view: provide the input to a box whose contents are unknown, see what output is produced.
  - **Could not choose representative test cases because we do not know enough about the processing to choose wisely.**
- Open box: we know the processing of the inside the box, we can choose test cases to test all paths.
  - **Usually can not test all paths**

Unit Testing

- Examining the Code
- Proving Code Correct
- Testing Program Components
- Comparing Techniques

Examining the Code

- **Code review**: Review both your code and its documentation for misunderstanding, inconsistencies, and other faults.
- Two types:
  - Code Walk-through
  - Code Inspection

Code review

- **Code Walk-through**: you present your code and accompanying documentation to the review team, you lead the discussion, they comments on correctness of your code.
- **Code Inspection**: a review team checks the code and documentation against a prepared list of concerns.

Success of Code Reviews

- You may feel uncomfortable with the idea of having a team examine your code.
- But, reviews have been shown to be extraordinarily successful at detecting faults.
- Code review becomes mandatory or best practices for most organization!

Success stories about Code Reviews

- One study by Fagan shows that 67% detected faults were found by code inspections.
- Code Inspection process produces 38% fewer failures (during the first 7 months of operation) than code walk-through process.

More stories about Code reviews

- Another study by Ackerman et. al. shows that 93% of all faults in a 6000-line business application were found by code inspections !
- Another study by Jones shows that code inspections removed as many as 85% of the total faults found.

Faults Found During Discovery Activities

- Discovery activities (Faults founded per 1000 lines of code)
- Requirements review (2.5)
- Design review (5.0)
- Code inspection (10.0)
- Integration test (3.0)
- Acceptance test (2.0)

Proving Code Correct

- Correct: a program is correct if it implements the functions and data property as indicated in the design, and if it interfaces properly with other components.
- Prove: view program code as statements of logical flow:
  - **Formal Proof Techniques**
  - **Advantages and Disadvantages of Correctness Proofs**
  - **Other Proof Techniques**
  - **Automated Theorem Proving**

Formal Proof Techniques

- Prove if A1 than A2.
- Prove if A2 than A3, ...
- Prove All paths.
- Prove the program terminates!!!
- [Never ending proofs.]

Advantages and Disadvantages of Correctness Proofs

- It takes a lot more time to prove the code correct than to write the code itself.
- The proof is more complex then the program itself.
- The proof may not be correct.

Other Proof Techniques: Symbolic Execution

- Simulated execution of the code using symbols instead of data variables.
- The program execution is viewed as a series of state changes.
  - **input state (determined by input data and conditions)**
  - **next state (caused by line of coded is executed)**
  - **next state ...**
  - **output state. (correct result?)**

Symbolic Execution: paths

- Each logical paths through the program corresponds to an ordered series of state changes. ((like an activity graph))
- Divide large sets of data into equivalence classes, work on the classes instead of the individual data: e.g.
  - **if (a > d)**
  - **doTaskx();**
  - **else**
  - **doTasky();**

Automated Theorem Proving

- Let the machine prove it.
- Develop tools to read as input
  - **input data and conditions**
  - **output data and conditions**
  - **lines of code for the component to be tested**
- Tells the user,
  - **(1) the component is correct, or**
  - **(2) counterexample showing the input is not correctly transformed to output by the component.**

Computational Limitation on Theorem Proving

- A theorem prover that read in any program and produce as its output either a statement confirming the code's correctness or the location of a fault, can never be built!!
- Limitation of computation. (Halting Problem)

Testing Program Components

- Testing vs. Proving
- Choosing Test Cases
- Test Thoroughness

Testing vs. Proving

- Testing is a series of experiments which gives us information about how program works in its actual operating environment.
- Proving tells us how a program will work in a hypothetical environment described by the design and requirements

Choosing Test Cases

- To test a component, we
- choose input data and conditions,
- allow the component to manipulate the data, and
- observe the output
- We select the input so that the output demonstrates something about the behavior of the code.

How to choose test cases

- select test cases and design a test to meet a specific objective:
  - **demonstrate all statements execute properly**
  - **every function performed correctly**
- Use closed-box or open-box view to help us choose test cases
  - **closed-box: supply all possible input and compare the output according to requirements**
  - **open-box: examine internal logic, and choose test to go through all paths.**

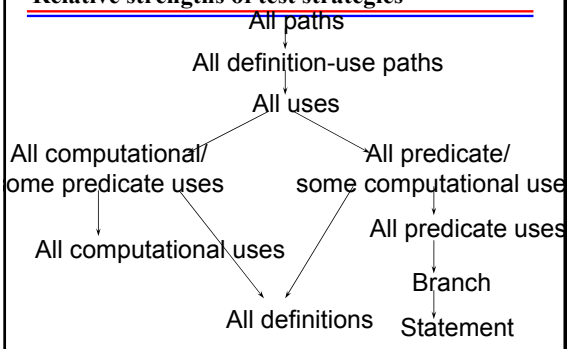
Test Thoroughness

- How to demonstrate in a convincing way that the test data exhibit all possible behaviors:
  - **Statement testing**
  - **Branch testing**
  - **Path testing**

Thoroughness based on data manipulated by the code

- definition-use path testing: all definition and their used are tested
- All-uses testing
- All-predicate-uses/some-computational-uses testing
- All-computational-uses/some-predicate-uses testing

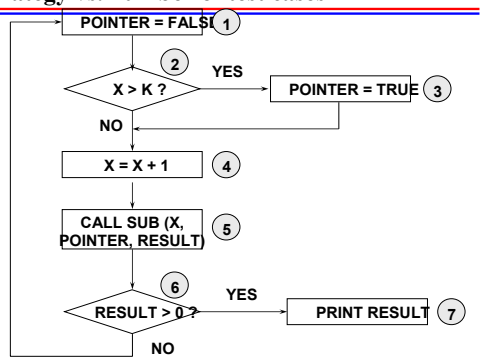
Relative strengths of test strategies



Relative strengths of test strategies comparison

- Study by Ntafos shows that
  - **random testing (not test strategies) found 79.5% faults**
  - **branch testing found 85.5%, and**
  - **all-uses testing found 90%**

Strategy vs. number of test cases



Number of test cases

- Statement testing: all statements
  - **(by choosing X and K) 1-2-3-4-5-6-7**
- Branch testing: all decision points
  - **1-2-3-4-5-6-7**
  - **1-2-4-5-6-1**
- Path testing: all paths
  - **1-2-3-4-5-6-7**
  - **1-2-4-5-6-1**
  - **1-2-4-5-6-7**
  - **1-2-3-4-5-6-1**

Comparing Techniques

- Fault discovery percentages by fault origin
  - **discovery technique, requirement, design, coding, documentation**
- Prototyping, 40, 35, 35, 15
- Requirement review, 40, 15, 0, 5
- design review, 15, 55, 0, 15
- code inspection, 20, 40, 65, 25
- unit testing, 1, 5, 20, 0

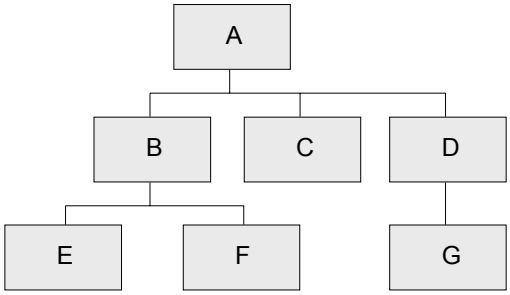
Integration Testing //

- Combine individual components into a working system:
- Bottom-up Integration
- Top-down Integration
- Big-bang Integration
- Sandwich Integration
- Comparison of Integration Strategies

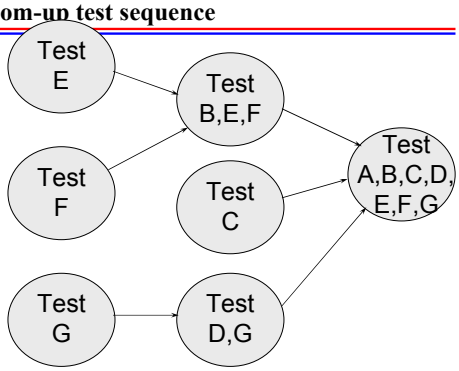
Bottom-up Integration

- Each component at the lowest level of the system hierarchy is tested individually first, then next components to be tested.
- Suitable for:
  - **object-oriented design**
  - **low-level components are general-purpose utility routines**

Component hierarchy



Bottom-up test sequence



Component Driver

- a special code to aid the integration.
- a routine that calls a particular component and passes a test case to it.
- take care of driver's interface with the test component

Bottom-up Integration pros and cons

- - top-level components are usually the most important but the last to be tested.
- + most sensible for object-oriented programs:
  - **Objects are combined one at a time with collections of objects that have been tested previously.**

Top-down Integration

- The top level, usually one controlling component, is tested by itself.
- Then, all components called by the tested component(s) are combined and tested as a larger unit.

Top-down testing

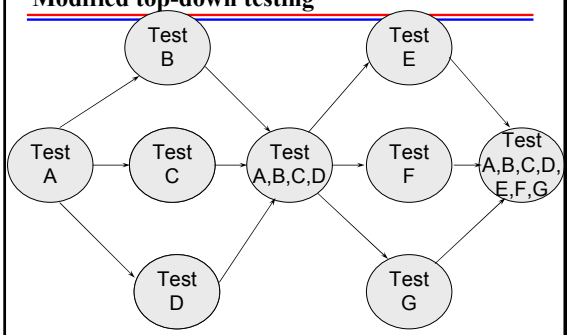
Stub

- Problem: A component being tested may call another that is not yet tested!
- Solution: Write a special-purpose program to simulate the activity of the missing component.
- The special-purpose program is called a stub.
  - **-If the lowest level of components performs the input and output operations, stubs for them may be almost identical to the actual components they replace.**

Top-down testing pros and cons

- + Any design faults or major questions about functional feasibility can be addressed at the beginning of testing instead of the end.
- - Writing stubs can be difficult, because they must allow all possible conditions to be tested.
- - Stub may itself needed to be tested to insure it is correct
- - Very large number of stubs may be required.

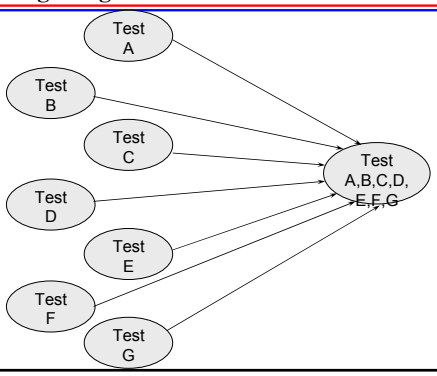
Modified top-down testing



Modified top-down (difficulty)

- Both stubs and drivers are needed for each component
- Much more coding and many potential problems.

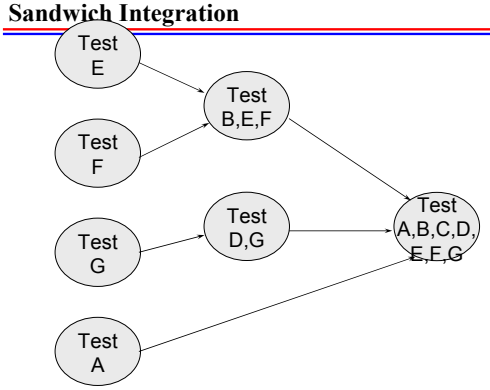
Big-bang Integration



Big-bang Integration (Not)

- Not recommended:
- - requires both stubs and drivers to test independent components
- - difficult to trace the cause of any failure since all components are merged all at once.
- interface faults cannot be distinguished easily from other types of faults.

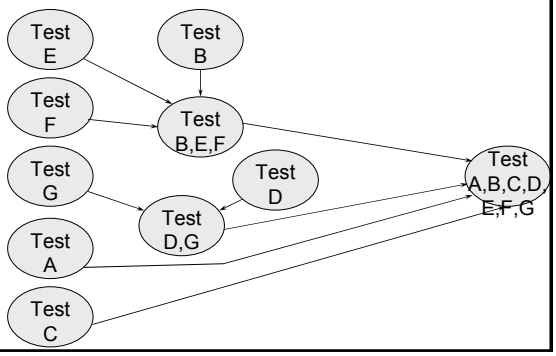
Sandwich Integration



Sandwich Up and Down

- combine both top-down and bottom-up
- three layers:
  - **bottom-up for the lower layer**
  - **top-down for the top layer**
  - **then, "big-bang" for mid layer**
- + combines advantages of top-down with bottom-up
- - individual components are not thoroughly tested before integration

Modified Sandwich testing: allows upper-level components to be tested before merging them



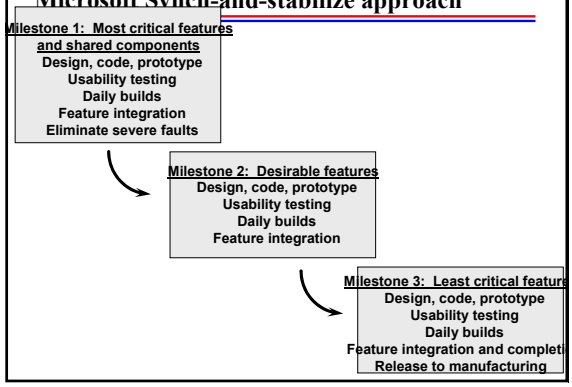
Comparison of Integration Strategies

	Botton-up	Top-down	Mod. Top-d	Big-bang	Sand-wich	Mod. Sand
Integrtion	Early	Early	Early	Late	Early	Early
Time to basic working program	Late	Early	Early	Late	Early	Early
Component Drivers needed	Yes	No	Yes	Yes	Yes	Yes
Stubs needed	No	Yes	Yes	Yes	Yes	Yes
Work parallelism at beginning	Medium	Low	Medium	High	Medium	High
Ability to test particular paths	Easy	Hard	Easy	Easy	Medium	Easy
Ability to plan and control sequence	Easy	Hard	Hard	Easy	Hard	Hard

Builds At Microsoft

- iterates: designing, building, testing -- involving customers in the testing process
- teams size: three to eight developers
- different teams are responsible for different features
- allows team to change the specification of features
- partitioning of features

Microsoft Synch-and-stabilize approach



Testing Object-Orientated Systems

- take several additional steps to make sure that your object-oriented programs' characteristics have been addressed by your testing techniques.
- Testing the code
- Differences between object-oriented and traditional testing

Testing the code

- you need more class definitions (missing class) if
  - **one class is playing two or more roles**
  - **an operation has no good target class**
- you have too many class definitions (unnecessary class) if
  - **a class has not attributes, operations, or associations.**
- Develop tests to track an object's state and changes to that state.

Differences between object-oriented and traditional testing

- Object-oriented does not always minimize testing
- adding or changing subclass requires re-testing of the methods inherited from each of its ancestor super-classes.
- need to develop new test case to test a method that is locally overrided by a subclass.
- >Use top-down testing: test base classes having no parents then next level ...

Object-oriented aspects make testing easier or harder

- objects tend to be small -- easier
- interface (inheritance) more complex -- harder
- = unit testing is less difficult
- = integration testing is more difficult
- = more source code analysis
- = more coverage analysis
- = more test case generation

Testing Planning

- Each step of the testing process must be planned:
  - **1. establishing test objectives**
  - **2. designing test cases**
  - **3. writing test cases**
  - **4. testing test cases**
  - **5. executing tests**
  - **6. evaluating test results**

Test Cases

- If test cases are not representative and do not thoroughly
- exercise the functions that demonstrate the correctness and validity of the system,
- then the remainder of the testing process is useless.
  - ✈ **"Testing" test case: to verify that they are correct, feasible, provide the desired degree of coverage, and demonstrate the desired functionality.**

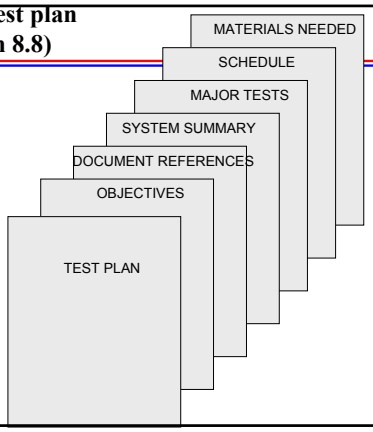
Test Plan

- describes the way in which we will show our customers that the software works correctly
- addresses unit testing, integration testing, and system testing.
- explains **who** does the testing, **why** the tests are performed, **how** the tests are conducted, and **when** the tests are scheduled.

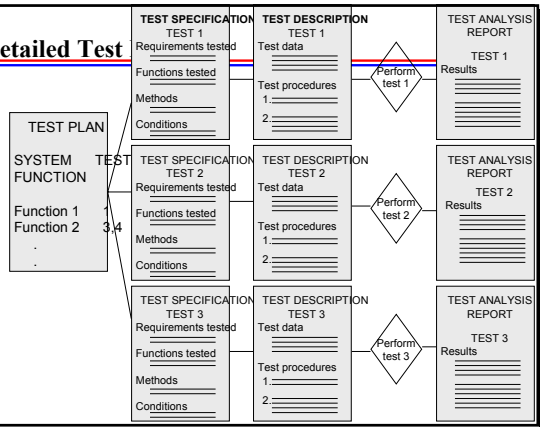
Contents of Test Plan

- test objectives,
- addressing each type of tests: unit --- to functional --- installation testing
- how test will be run
- what criteria will be used to determine when the testing is complete.
- testing tools needed.
- testing environment needed.

Parts of a test plan See (Section 8.8)



Detailed Test



Test specification and evaluation, test description

- Test specification and evaluation: details each test and defines the criteria for evaluating each feature addressed by the test.
- Test description: presents the test data and procedures for individual tests.
- > Use naming or numbering scheme that ties together all documents.

Test schedule includes

- 1. the overall testing period
- 2. the major subdivisions of testing, and their start and stop times
- 3. any pretest requirements (generation of test data, setting up test environments) and the time necessary for each
- 4. the time necessary for preparing and reviewing the test report

More on Test Specification and Evaluation

- (Test plan describes an overall breakdown of testing into individual tests)
- for each individual tests, we write a test specification and evaluation
- (keep track on the correspondence between requirements and tests)

Specification states test conditions

- Is the testing using actual input from user or devices, or are special cases generate by a program or surrogate device?
- What are the test coverage criteria?
- How will data be recorded?
- Are there timing, interface, equipment, personal, database, or other limitations on testing?
- What order are the test to be performed?

More on Test Description

- A test description is written for every test defined in the test specification.
- We use the test description as a guide in performing the test.
- It states clearly:
  - **the means of control**
  - **the data**
  - **the procedures**
- It provides procedure "test script" to guide us through the test.

Test Script

- gives a step-by-step description of how to perform the test
- provides rigidly defined set of steps to give us control over the test:
  - **allows us to duplicate conditions and recreate the failure if necessary.**
- steps are numbered and data associated with each step are referenced.

Automated Testing Tools

- testing tools are useful and often necessary
- Tools for:
  - **Code Analysis**
  - **Test Execution**
  - **Test Case Generation**

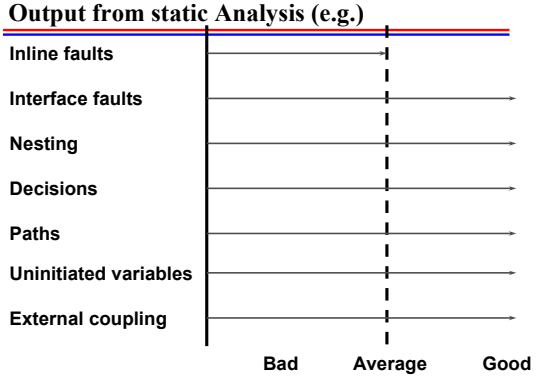
Code Analysis tools

- Static Analysis: is performed when the program is not actually executing
- Dynamic analysis: is done when the program is running.

Static Analysis

- Code analyzer: check proper syntax
- Structure checker: generates a graph from codes to depict the logic flow, also check structural flaws.
- Data analyzer: review the data structures and data declarations, check conflicting data definitions and illegal data usage.
- Sequence Checker: check sequences of events

Output from static Analysis (e.g.)



Dynamic analysis

- also called program monitors or debugger: they watch and report the program's behavior
- tracing the running of program
- Set break points to stop running, allow us to see a "snapshot" of program status
- examine the contents of memory or values of specific data items.

Test Execution tools

- tools for automating the test planing and even running the tests themselves
- Capture and Replay: record key-strokes, mouse movement, mouse clicks, and
  - **responses to whose inputs**
  - **while a test case is being execute by a human tester.**
- Playback or replay the recorded test cases.

Stubs and Drivers Tools

- Commercial tools are available to assist you in generating stubs and drivers automatically!

Test Case Generators

- Structural test case generators: base their test cases on the structure of the source code.
- Formal specifications of programs: using extend finite state machine, (like UML), can generate all possible paths
- user can choose coverage criteria

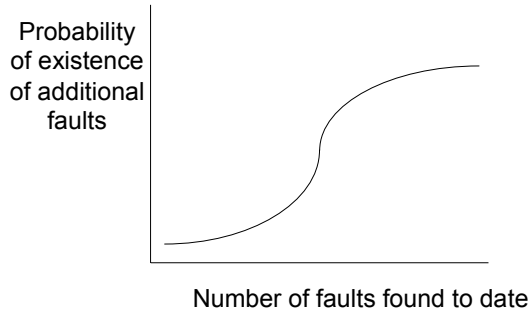
Automated Testing Environments

- Test planning,
- Test case Generations
- Test execution,
- Test result reporting,
- Re-test
- Testing result analysis
- ($$$)

When to Stop Testing

- NOW? or Never ending....
  - **large number of faults find -> more faults to be find**
- Faults Seeding
- Confidence in the software
- Other Stopping Criteria
- Identifying Fault-prone Code

Myers Study shows (more fault-> more to be found)



Faults Seeding

- intentionally inserts (or "seeds") a know number of faults in a program.
- (detected seeded faults)/(total seeded faults) = (detected nonseeded faults)/(total nonseeded faults)
- Pro and cons
  - **+ simple and useful**
  - **- assumes that seeded faults are of the same kind and complexity as the actual faults, but we don't know what kind of faults are where.**

Two groups testing the same program

- group A found 25 faults (A)
- group B found 30 faults (B)
- both groups found the Same 15 faults (S)
- Total number of faults in the program??? (**T**)
- In general: S <= A and S <= B

How many faults are where in the program?

- (Effectiveness of group A) = A/**T**
  - **Assume group A is just as effective at finding faults in any part of the program as in any other part. Thus that,**
- (Effectiveness of group A) = S/B
- (Effectiveness of group A) = A/**T** = S/B
- **T** = A*B/S
- **T** = 25*30/15 = 50 (faults in the program)

Confidence in the software

- Confidence, usually expressed as a percentage, tells us the liklihood that the software is faults-free.
- If we say a program is fault-free with a 95% level of confidence,
- then we mean that the probability that the program has no faults is 0.95

Confidence calculation using "Fault Seeding" method

- seeded a program with 10 faults (S)
- we claim a fault-free program there is 0 faults (N)
- We find all 10 seeded faults but not others faults (n)
- Confidence level = S/(S - N + 1) if n <= N
- Confidence level = 10/(10 - 0 + 1) = 10/11 = 91%

Higher Confidence more tests

- Contract or requirements mandate a confidence level of 98% that the program is fault-free
- That is: S/(S - 0 + 1) = 98/100
- Solve for S:

S/(S + 1) = 0.98
S = 0.98*S + 0.98
S = 0.98/(1-0.98) = 49

Better calculation Confidence

- **So far, can not predict confidence until all seeded faults are founded.**
- seeded a program with 10 faults (S)
- we claim a fault-free program there is 0 faults (T)
- We find only 8 seeded faults but not others faults (f)
- (Richards) Confidence = (S!*(f+T)!) / ((f-1)!*(S+T+1)!)

•=73%

Other Stopping Criteria

- determine our test progress in terms of the number of statements, paths, or branches left to test
- Use automated tool to calculate coverage values for us.

Identifying Fault-prone Code