Software Testing

Software Testing

- Is often referred to as verification and validation (V&V).
- Verification
  - Activities ensure that implementation has a specific function
  - *Are we building the product right?*
- Validation
  - Activities ensure that software traceable to requirements
  - *Are we building the right product?*

What Testing Shows?

- The goal of defect testing is to discover defects in programs
- A successful defect test is a test which causes a program to behave in an anomalous way

Testing can only show the presence of errors, not their absence

E.W.Dijkstra, 1972

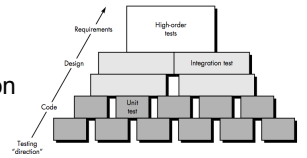
Testing Strategy

- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- Conventional Software
  - The module (component) is our initial focus
  - Integration of modules follows
- OO Software
  - our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

When is Testing Complete?

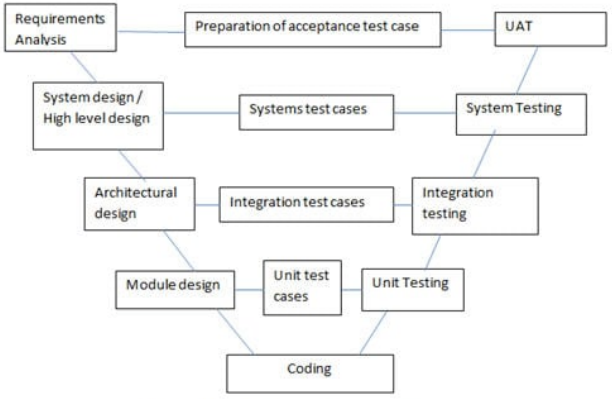
- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
  - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

Four Testing Steps



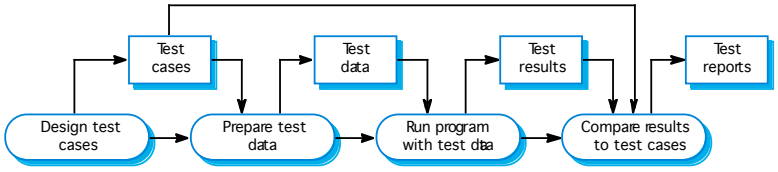
- **Step1**: Focus on each component in isolation
  - unit testing
  - heavy use of white-box testing techniques
    - exercise specific paths in the control structure for complete coverage
- **Step2**: Assemble components (integration)
  - mainly uses black-box testing techniques (may also uses white-box tests)
    - reason in terms of inputs and expected outputs
- **Step3**: Conduct higher-order tests
  - validation criteria must be set (related to requirements)
    - related to behavior and performance
  - uses exclusively black-box techniques
- **Step4**: Combine with other systems
  - goes beyond SWE ... more systems engineering

V-Model of Software Testing



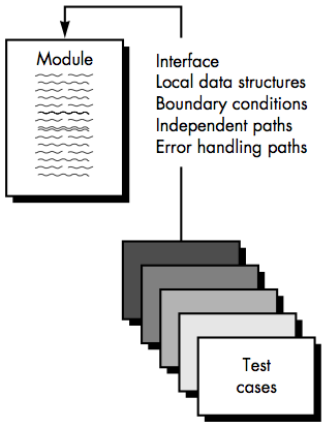
STLC - V Model - © www.SoftwareTestingHelp.com

The software testing process



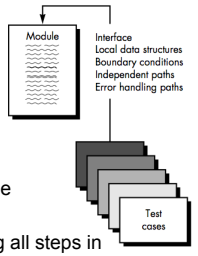
from I.Sommerville, SE8

Unit Testing



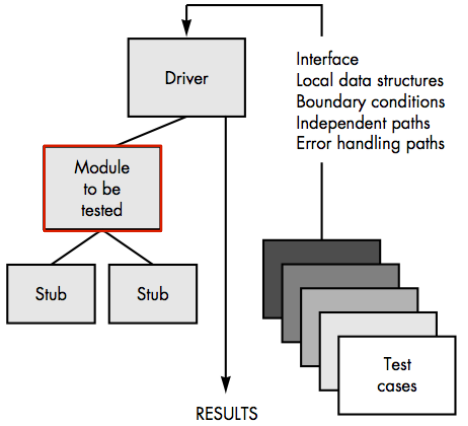
from R.S.Pressman, 2005

Targets for Unit Test Cases



- Module interface
  - Ensure that information flows properly into and out of the module
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific error conditions

Unit Test Environment



from R.S.Pressman, 2005

Drivers and Stubs for Unit Testing

- Driver
  - A simple main program that accepts test case data
  - passes such data to the component being tested
  - prints the returned results
- Stubs
  - replace modules that are subordinate to (called by) the tested component
  - uses the module's exact interface
    - may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent overhead
  - must be written but don't constitute part of the installed software product

Integration Testing

- Why is unit testing not enough?
  - interfacing causes problems
- Examples of Interfacing Problems:
  - data loss across an interface
  - adverse effects of one module on another
  - individually acceptable imprecisions, may not be globally acceptable
- Two Options:
  - "Big Bang" approach - **NO!**
  - **Incremental Integration**

"Big-Bang" Integration Testing

- non-incremental integration
- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
  - Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur
  - testing appears to enter an endless loop

Incremental Integration Testing

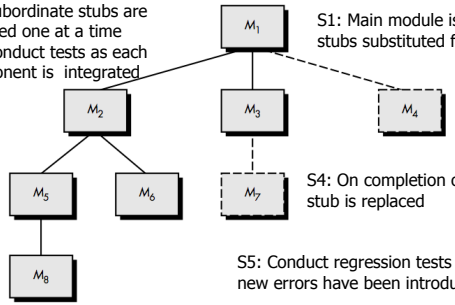
- The program is constructed and tested in small increments
- **Advantages**
  - Errors are easier to isolate and correct
  - Interfaces are more likely to be tested completely
  - A systematic test approach is applied
- Two Strategies
  - Top-down integration
  - Bottom-up integration

Top-Down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - Depth-First: All modules on a major control path are integrated
  - Breadth-First: All modules directly subordinate at each level are integrated
- **Advantages**
  - verifies major control or decision points **early** in the test process
- **Disadvantages**
  - Stubs need to be created for modules that have not been built or tested yet
  - No significant data flow can occur until much later in the integration process
    - because stubs are used to replace lower level modules

Top-Down Integration

S2: Subordinate stubs are replaced one at a time
S3: Conduct tests as each component is integrated



S1: Main module is used as test driver, stubs substituted for all direct subordinate components

S4: On completion of each sets of tests, another stub is replaced

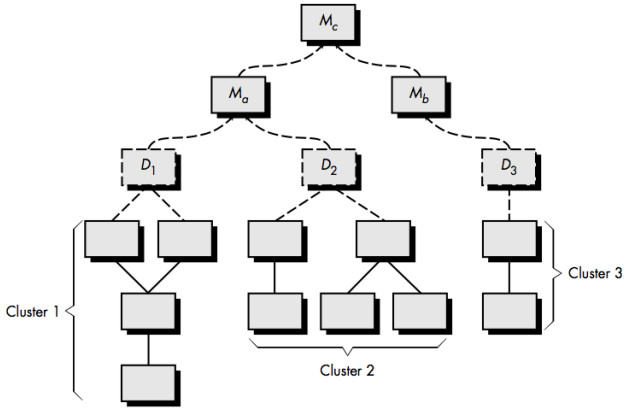
S5: Conduct regression tests to make sure that no new errors have been introduced

from **R.S.Pressman, 2005**

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- **Advantages**
  - verifies low-level data processing early in the testing process
  - No need for stubs
- **Disadvantages**
  - Driver modules needed
    - to test the lower-level modules
    - this code is later discarded or expanded into a full-featured version
  - Drivers are inherently incomplete
    - do not contain the complete algorithms that will eventually use the services of the lower-level modules
    - testing may be incomplete or more testing may be needed later when the upper level modules are available

Bottom-Up Integration



from **R.S.Pressman, 2005**

Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
    - changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test contains 3 classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests on software functions likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

Smoke Testing

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule
  - Build is integrated with other builds and the entire product is **smoke tested daily**
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

Benefits of Smoke Testing

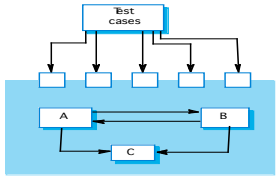
- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

Test Case 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"
  - reveals more than one error... an entire class of errors
- A good test should be neither too simple nor too complex
  - if too simple, might not reveal anything
  - if too complicated danger of side effects
  - Tests should be run individually

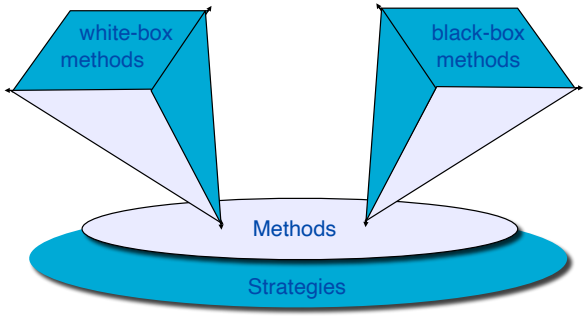
Testing guidelines



- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.

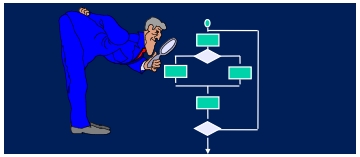
Design tests which cause the component to fail.

Software Testing



from **R.S.Pressman, 2005**

White-Box Testing



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

from **R.S.Pressman, 2005**

- The starting point for path testing is a program flow graph
  - nodes = program decisions
  - arcs = flow of control.
  - Statements with conditions are therefore nodes in the flow graph.

Why Cover?

- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
  - more errors in rarely executed paths
- We often **believe** that a path is not likely to be executed
  - reality is often counter-intuitive
- typographical errors are random
  - likely that untested paths will contain some

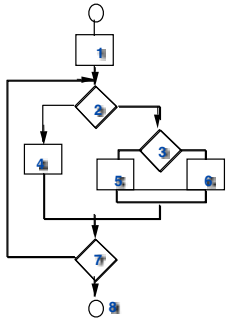
Cyclomatic Complexity [McCabe, 1976]

- Definition:

Let $G(N, E)$ be a control flow graph (CFG).
Cyclomatic complexity is defined as

$$v(G) = e - n + 2$$

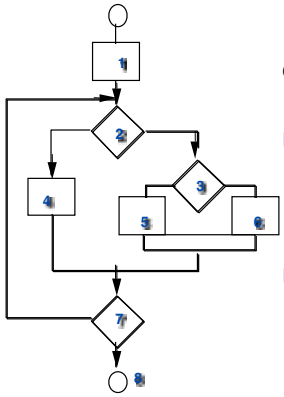
where $e = |E|$ and $n = |N|$.



- Interpretation:

more cyclomatic complexity = more branching =
=more testing needed

Basis Path Testing



First, we compute the cyclomatic complexity:

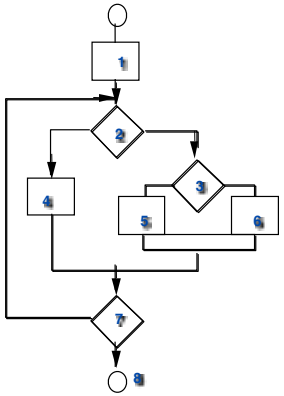
number of simple decisions + 1

or

number of enclosed areas + 1

In this case, V(G) = 4

Basis Path Testing



Next, we derive the independent paths:

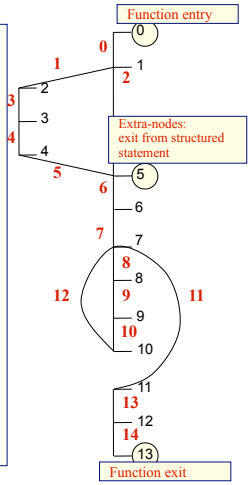
Since V(G) = 4,
there are four paths

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.

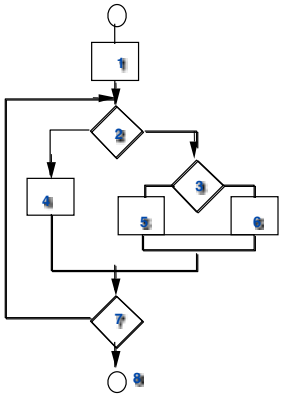
Cyclomatic Complexity for Euclid algorithm

```
int euclid(int m, int n)
{                                       /*0*/
    int r;
    if(n > m)                           /*1*/
    {
        r = m; m = n; n = r;            /*2,3,4:*/
    }                                   /*5*/
    r = m % n;                          /*6*/
    while(r != 0)                       /*7*/ {
        m = n; n = r; r = m % n;        /*8,9,10:*/
    }                                   /*11*/
    return r;                           /*12*/
}                                       /*13*/
```



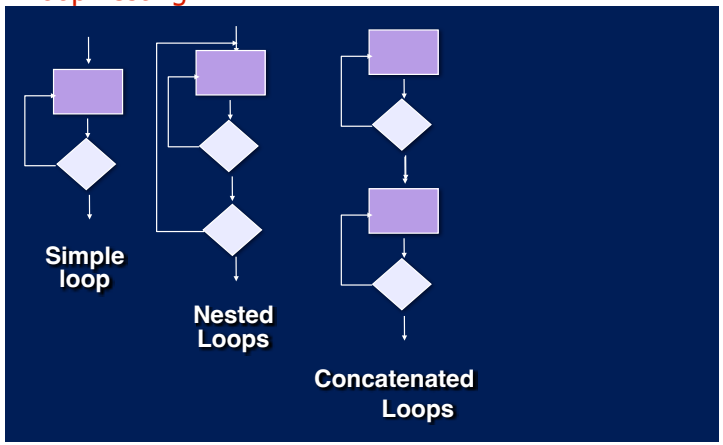
$v(G) = 15 - 14 + 2 = 3$

Remarks on Basis Path Testing



- You don't need a flow chart, but the picture will help when tracing the program paths
- Count each simple logical test!
  - **Compound tests count as >= 2**
- Basis Path Testing should be applied to critical modules

Loop Testing



from **R.S.Pressman, 2005**

Loop Testing: Simple Loops

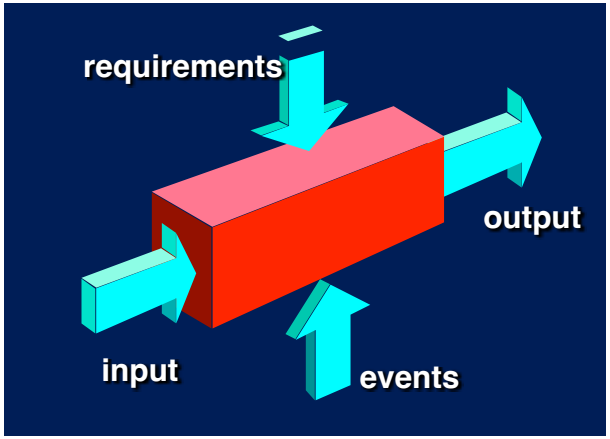
- Minimum Conditions to Test:
  1. Skip the loop entirely
  2. Only one pass through the loop
  3. Two passes through the loop
  4. N-1 and N passes through the loop

where N is the maximum number of allowable passes

Loop Testing: Nested Loops

- **Nested Loops**
  1. Start at the innermost loop. Set all outer loops to their minimum iteration parameter values
  2. apply **simple loop testing rules** to the innermost loop, while holding the outer loops at their minimum values
  3. Move out one loop and set it up as in step 2, holding all outer loops at minimum values, and all inner loops at typical values.
  4. Continue step 3 until the outermost loop has been tested
- **Concatenated Loops**
  6. If the loops are independent of one another, treat each as a simple loop
  7. Else treat as a nested loop (e.g. the final loop counter value of loop 1 is used to initialize loop 2

Black-Box Testing

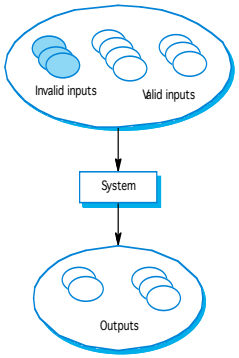


from R.S.Pressman, 2005

Partition testing

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition
  - i.e a domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

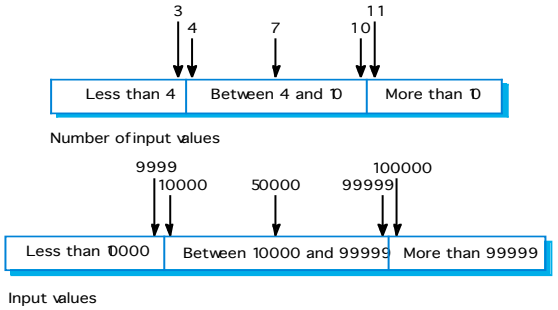
Equivalence partitioning



- Identify partitions from program specs or user documentation
- For each partition choose test cases:
  - **boundaries**
  - **mid-point**

Equivalence partitions

- Example: program accept **4 to 10 inputs** that are **five-digit integers > 10.000**



Guidelines for testing **sequences (collections)**

- Test software with sequences which have only **a single value**.
- Test with sequences of **zero length**.
- Use **sequences of different sizes** in different tests.
- Derive tests so that the **first**, **middle** and **last elements** of the sequence are accessed.