


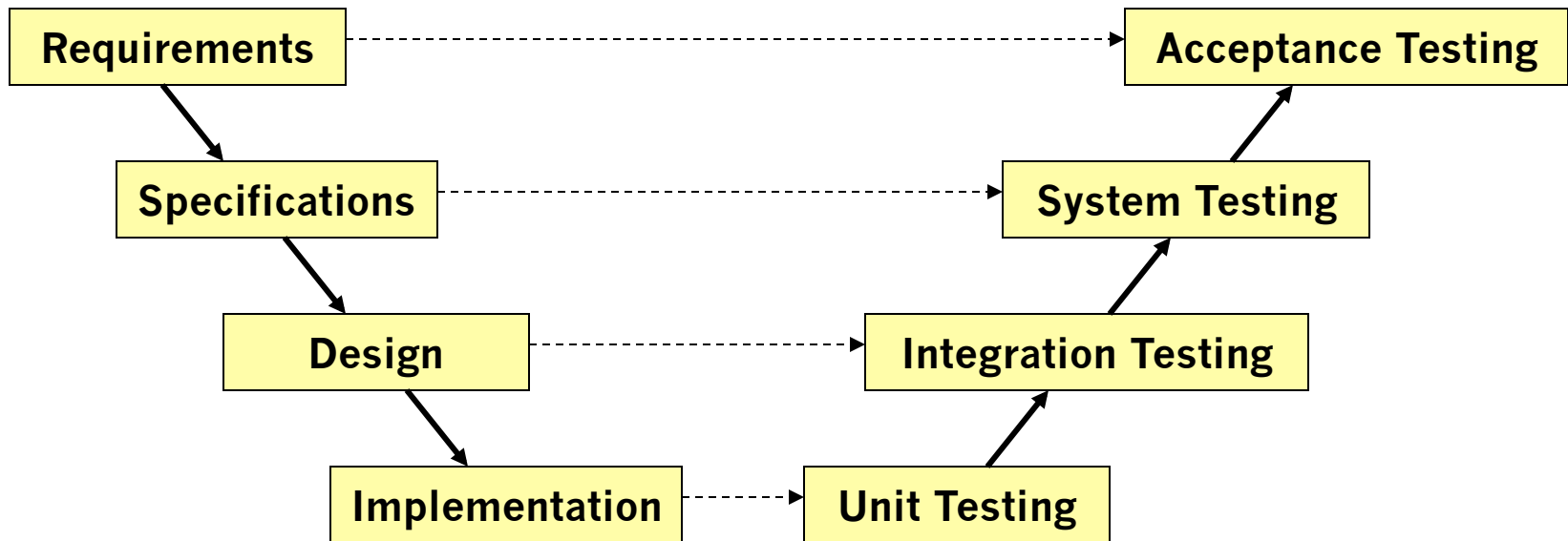


Software Testing



Some Material adapted from Lethbridge & Laganier;
Some Material adapted from Pressman.

Testing phases: V model



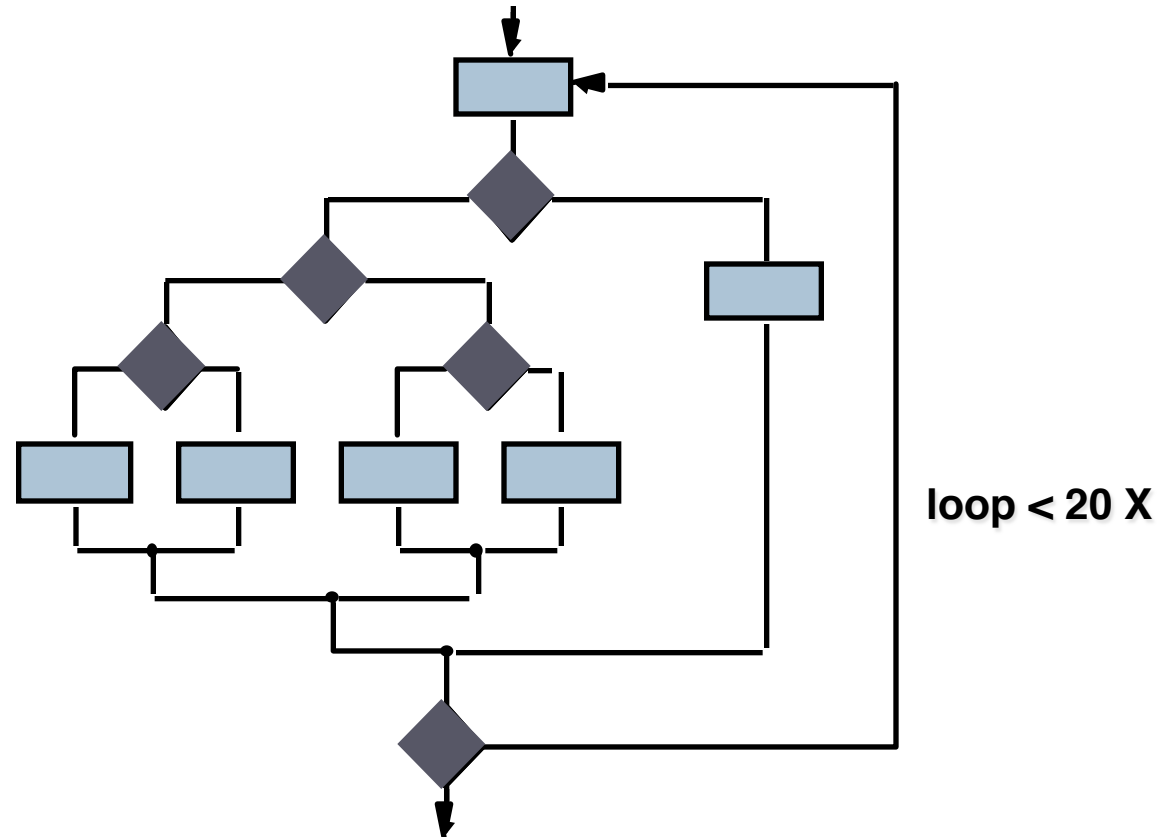
A lifecycle view that shows relationships between development and test phases

Testing Phases

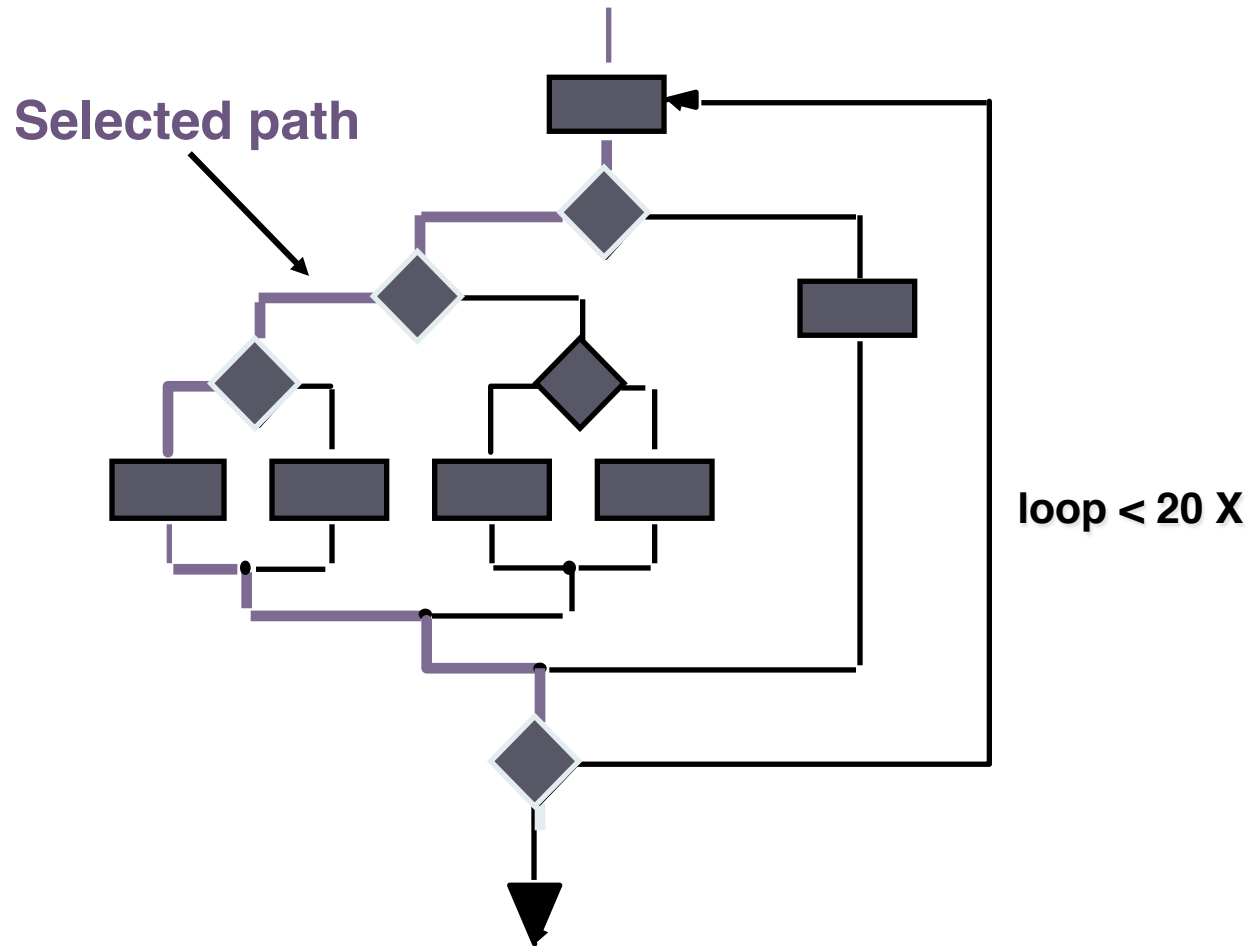
- ▶ **Unit Testing**
 - ▶ Developer tests individual modules
- ▶ **Integration testing**
 - ▶ Put modules together, try to get them working together
 - ▶ Integration testing is complete when the different pieces are able to work together
- ▶ **System testing**
 - ▶ Black-box testing of entire deliverable against specs
- ▶ **Acceptance testing**
 - ▶ Testing against user needs, often by the user



Exhaustive Testing



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



Coverage

- ▶ Function coverage: Each function/method executed by at least one test case
- ▶ Statement coverage: Each line of code covered by at least one test case (need more test cases than above)
- ▶ Path coverage: Every possible path through code covered by at least one test case (need lots of test cases)



Equivalence classes

- ▶ It is inappropriate to test by *brute force*, using every *possible* input value
 - ▶ Takes a huge amount of time
 - ▶ Is impractical
 - ▶ Is pointless!
- ▶ You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
 - ▶ Such groups are called *equivalence classes*.
 - ▶ A tester needs only to run one test per equivalence class
 - ▶ The tester has to
 - understand the required input,
 - appreciate how the software may have been designed



Example of equivalence classes

Valid input is a month number (1-12)

- ▶ Equivalence classes are: $[-\infty..0]$, $[1..12]$, $[13..\infty]$



Combinations of equivalence classes

- ▶ Combinatorial explosion means that you cannot realistically test every possible system-wide equivalence class.
 - ▶ If there are 4 inputs with 5 possible values there are 5^4 (i.e.625) possible system-wide equivalence classes.
- ▶ You should first make sure that at least one test is run with every equivalence class of every individual input.
- ▶ You should also test all combinations where an input is likely to *affect the interpretation* of another.
- ▶ You should test a few other random combinations of equivalence classes.



Example equivalence class combinations

- ▶ One valid input is either 'Metric' or 'US/Imperial'
 - ▶ Equivalence classes are:
 - ▶ Metric, US/Imperial, Other
- ▶ Another valid input is maximum speed: 1 to 750 km/h or 1 to 500 mph
 - ▶ Validity depends on whether metric or US/imperial
 - ▶ Equivalence classes are:
 - ▶ $[-\infty..0]$, $[1..500]$, $[501..750]$, $[751.. \infty]$
- ▶ Some test combinations

▶ Metric, $[1..500]$	valid
▶ US/Imperial, $[501..750]$	invalid
▶ Metric, $[501..750]$	valid
▶ Metric, $[501..750]$	valid

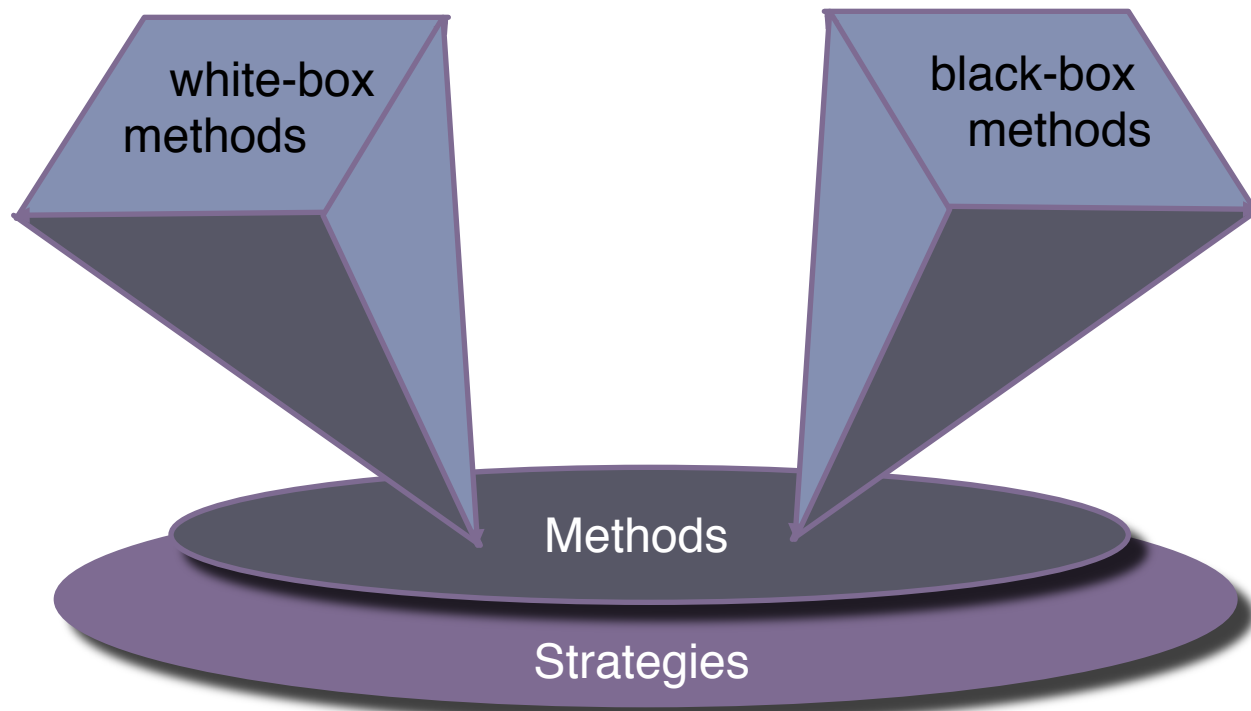


Testing at boundaries of equivalence classes

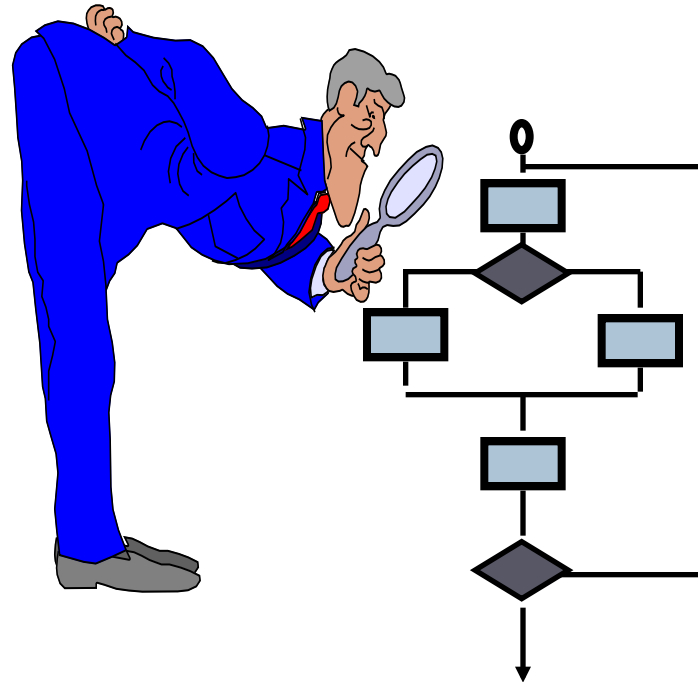
- ▶ More errors in software occur at the boundaries of equivalence classes
- ▶ The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
 - ▶ E.g. The number 0 often causes problems
- ▶ E.g.: If the valid input is a month number (1-12)
 - ▶ Test equivalence classes as before
 - ▶ Test 0, 1, 12 and 13 as well as very large positive and negative values



Software Testing



White-Box Testing



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

White-box testing

- ▶ Also called 'glass-box' or 'structural' testing
- ▶ Testers have access to the system design
 - ▶ They can
 - ▶ Examine the design documents
 - ▶ View the code
 - ▶ Observe at run time the steps taken by algorithms and their internal data
- ▶ Individual programmers often informally employ glass-box testing to verify their own code



White Box Methods

- ▶ Can be applied at all levels of system development – unit, integration, and system
- ▶ Coverage (Control flow)
 - ▶ statement
 - ▶ branch
- ▶ Dataflow
- ▶ Mutation

Statement Coverage

- ▶ Execute each statement in the program
- ▶ Considered minimum criterion for most unit testing
- ▶ May be difficult to achieve for error cases

Example Program

```
1:  if (a < 0) {  
2:      return 0 }  
3:  r = 0;  
4:  c = a;  
5:  while (c > 0) {  
6:      r = r + b;  
7:      c = c - 1; }  
8:  return r;
```

Statement tests

$a = 3, b = 4$

executes 1, 3, 4, 5, 6, 7,
5, 6, 7, 5, 6, 7, 5, 8

$a = -3, b = 2$

executes 1, 2

Branch Coverage

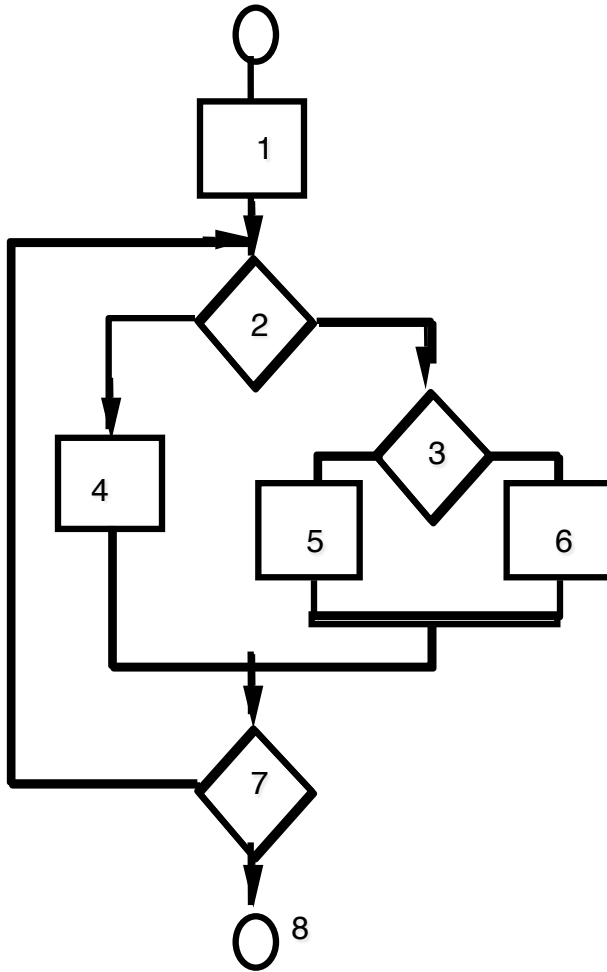
- ▶ Execute each branch of the program at least once
- ▶ Differs from statement coverage only for "if" statements without "else"s and case statements without default cases.

Levels of Coverage

- ▶ Level 1: 100% statement coverage
- ▶ Level 2: 100% decision coverage or branch coverage
- ▶ Level 3: 100% condition coverage
- ▶ Level 4: 100% decision/condition coverage
- ▶ Level 5: 100% multiple condition coverage
- ▶ Level 6: Limited path coverage
- ▶ Level 7: 100% path coverage



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

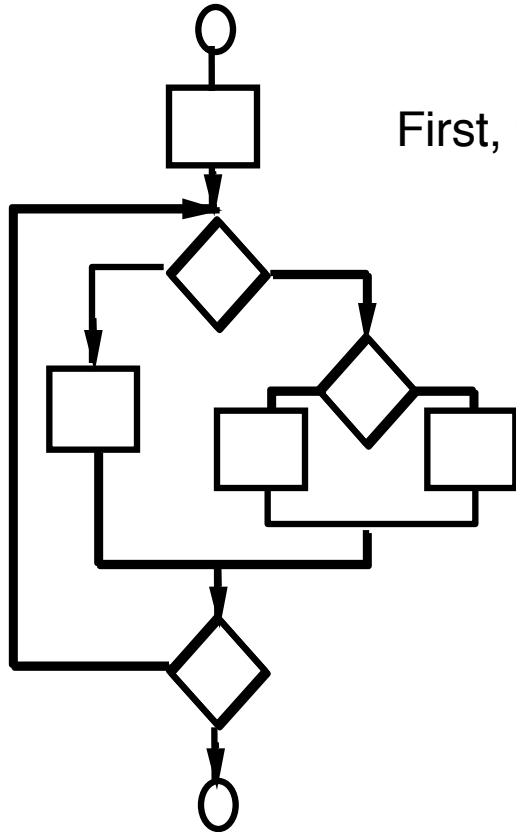
Path 2: 1,2,3,5,7,8

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

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

Finally, we derive test cases to exercise these paths.

Basis Path Testing



First, we compute the **McCabe's cyclomatic complexity**:

number of simple decisions + 1

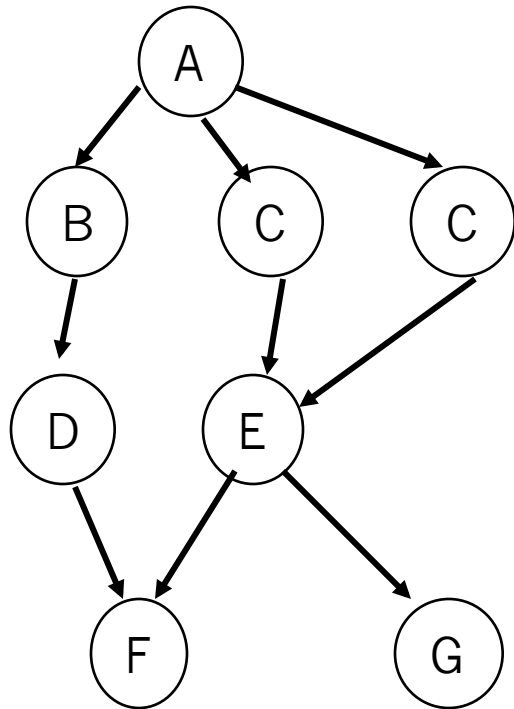
or

number of enclosed areas + 1

In this case, $V(G) = 4$

If decisions are not binary, this can't be used for computation of cyclomatic complexity

Cyclomatic complexity



- Cyclomatic complexity of a graph is

$$C = \text{edges} - \text{nodes} + 2$$

Why Data flow testing?

- ▶ Look at this simple C program

```
#include <stdio.h>
main() {
    int x;
    printf ("%d",x);
}
```

What value for x will be printed ?

Dataflow Testing

Definitions and Uses

- ▶ Defining node
 - ▶ input statement
 - ▶ lhs of assignment
- ▶ Usage node
 - ▶ output statement
 - ▶ rhs of assignment
 - ▶ control statement
- ▶ **DU testing:** execute each du-path of each variable

Example Program

```
1:  if (a < 0) {  
2:      return 0 }  
3:  r = 0;  
4:  c = a;  
5:  while (c > 0) {  
6:      r = r + b;  
7:      c = c - 1; }  
8:  return r;
```

Example DU Paths

Def (c) = {4, 7}

Use (c) = {5, 7}

Def (r) = {3, 6}

Use (r) = {6, 8}

du-paths for c:

4 - 5, 4 - 5 - 6 - 7, 7 - 5, 7 - 5 - 6 - 7

du-paths for r:

3 - 4 - 5 - 6, 3 - 4 - 5 - 8,
6 - 7 - 5 - 6, 6 - 7 - 5 - 8

Test Cases for DU Paths

$$a = 2$$

1 - 3 - 4 - 5 - 6 - 7 - 5 - 6 - 7 - 5 - 8

Covers du-paths:

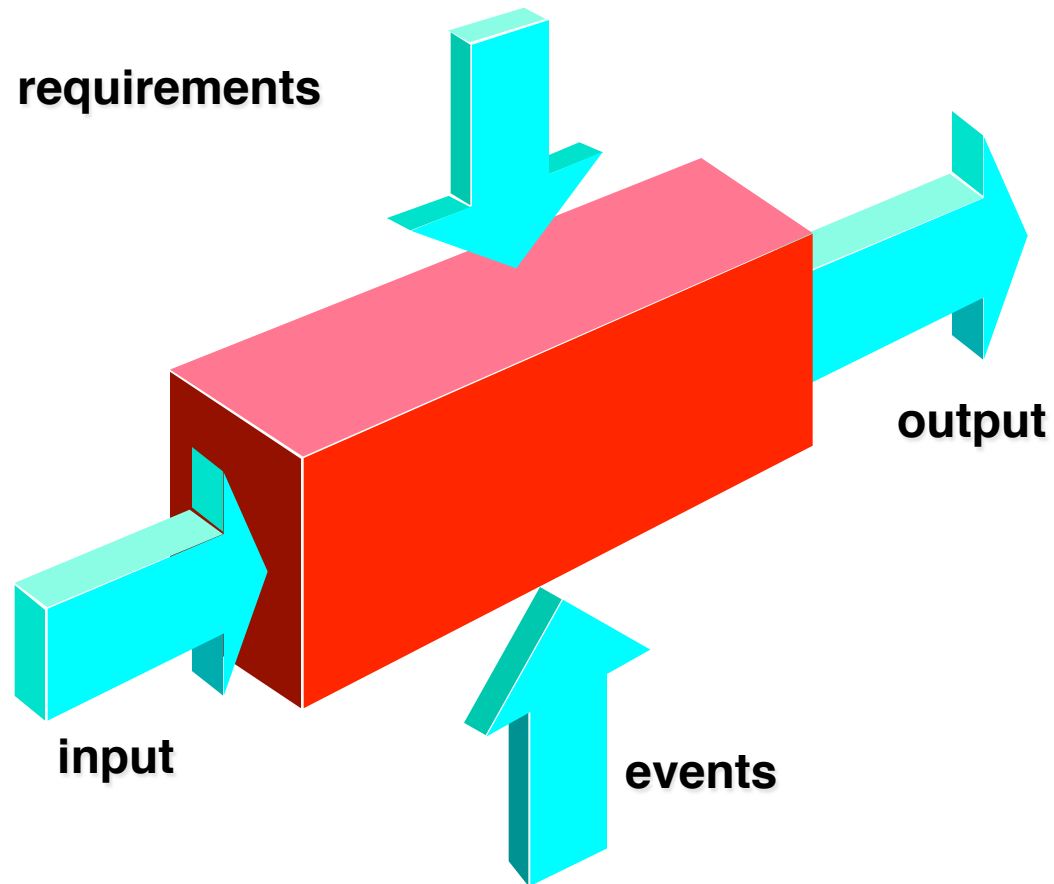
4 - 5, 4 - 5 - 6 - 7, 7 - 5, 7 - 5 - 6 - 7

3 - 4 - 5 - 6, 6 - 7 - 5 - 6, 6 - 7 - 5 - 8

Suspicious Paths

- ▶ Variable is defined (set to a new value) but never referenced
- ▶ Variable is referenced but never defined
- ▶ Variable is defined twice before it is used

Black-Box Testing



Testing

- ▶ Software products are tested at four levels:
 - ▶ Unit testing
 - ▶ Integration testing
 - ▶ System testing
 - ▶ Acceptance testing



Unit testing

- ▶ During unit testing, modules are tested in isolation:
 - ▶ If all modules were to be tested together:
 - ▶ it may not be easy to determine which module has the error.
- ▶ Unit testing reduces debugging effort several folds.
 - ▶ Programmers carry out unit testing immediately after they complete the coding of a module.



Role of Unit Testing

- ▶ Assure minimum quality of units before integration into system
- ▶ Focus attention on relatively small units
- ▶ Testing forces us to read our own code – spend more time reading than writing
- ▶ Automated tests support maintainability and extendibility
- ▶ Marks end of development step



Integration testing

- ▶ After different modules of a system have been coded and unit tested:
 - ▶ modules are integrated in steps according to an integration plan
 - ▶ partially integrated system is tested at each integration step.

Objectives:

- Gain confidence in the integrity of overall system design
- Ensure proper interaction of components
- Run simple system-level tests



Integration Testing Strategies

- ▶ Big-bang
- ▶ Top-down
- ▶ Bottom-up
- ▶ Critical-first
- ▶ Function-at-a-time
- ▶ As-delivered
- ▶ Sandwich



Big Bang Integration Testing

- ▶ Big bang approach is the simplest integration testing approach:
 - ▶ all the modules are simply put together and tested.
 - ▶ this technique is used only for very small systems.

Issues:

- ▶ avoids cost of scaffolding (stubs or drivers)
- ▶ does not provide any locality for finding faults



Top-down integration testing

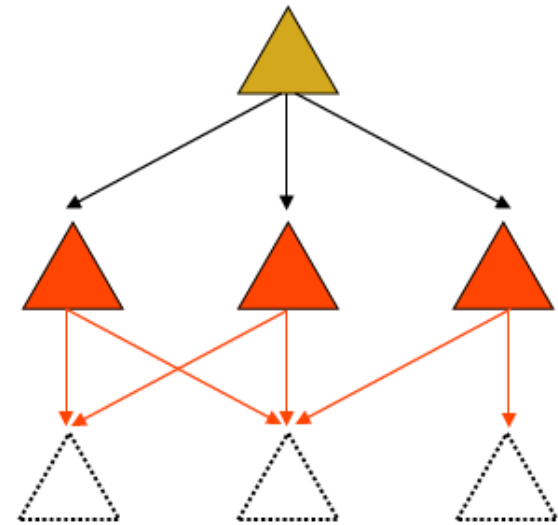
- ▶ Start with top-level modules
- ▶ Use stubs for lower-level modules
- ▶ As each level is completed, replace stubs with next level of modules

Pros:

- ▶ Always have a top-level system
- ▶ Stubs can be written from interface specifications

Cons:

- ▶ May delay performance problems until too late
- ▶ Stubs can be expensive



Bottom-up Integration Testing

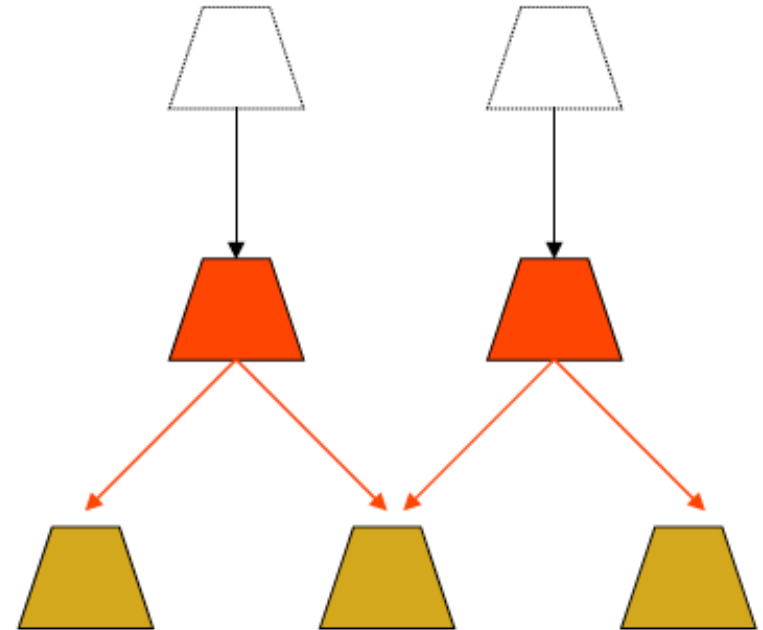
- Start with bottom-level modules
- Use drivers for upper-level modules
- As each level is completed, replace drivers with next level of modules

Pros:

- Primitive functions get most testing
- Drivers are usually cheap

Cons:

- Only have a complete system at the end



Critical-first Integration

- ▶ Integrate the most critical components first
- ▶ Add the remaining pieces later

Issues:

- Guarantees that the most important components work
- May be difficult to integrate



Function-at-a-time Integration

- ▶ Integrate all modules needed to perform a particular function
- ▶ For each function, add another set of modules

Issues

- ▶ Makes for easier test generation
- ▶ May postpone function interaction for too long
 - ▶ Dependencies may create a problem



As-Delivered Integration

- ▶ Integrate the modules as and when they become available

Issues

- ▶ Efficient – Just-in-time Integration
- ▶ Lazy – may lead to missed schedules



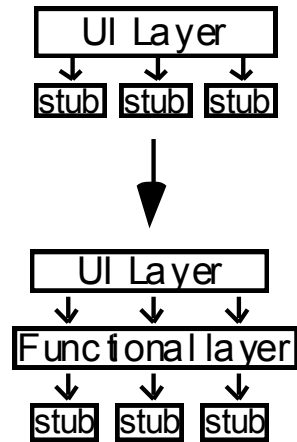
Sandwich Integration Testing

- ▶ **Mixed (or sandwiched) integration testing:**
 - ▶ uses both top-down and bottom-up testing approaches.
 - ▶ Most common approach

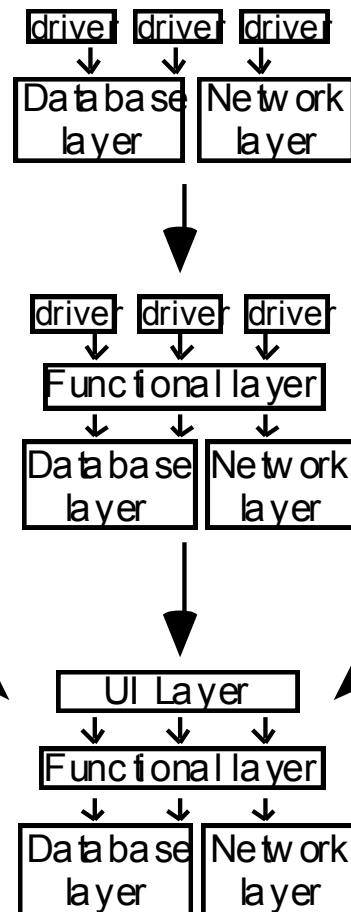


Example of different integration strategies

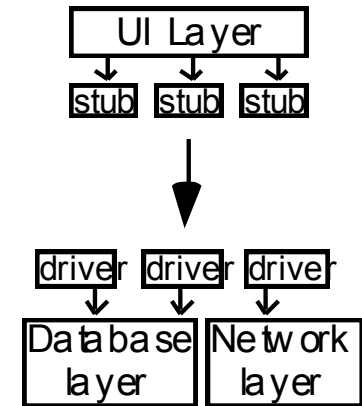
Top-down testing



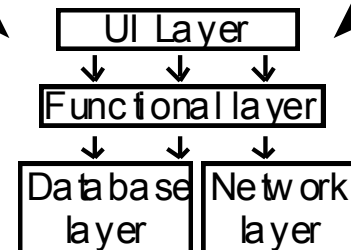
Bottom-up testing



Sandwich testing



Fully
integrated
system



System Testing

Objectives

- ▶ Gain confidence in the integrity of the system as a whole
 - ▶ Ensure compliance with functional requirements
 - ▶ Ensure compliance with performance requirements



Testing Functional Requirements

1. Prepare a test plan from the functional specification of the system
2. Prepare tests for all areas of functionality
3. Review test plan and tests
4. Execute tests
5. Monitor fault rate



Testing Performance Requirements

1. Identify stress points of system
2. Create or obtain load generators
 - ▶ might use existing system
 - ▶ might buy/make special purpose tools
3. Run stress tests
4. Monitor system performance
 - ▶ usually needs instrumentation



Acceptance Testing

- ▶ Testing performed by the customer or end-user himself:
 - ▶ to determine whether the system should be accepted or rejected.

Other paths to acceptance:

- ▶ Beta testing
 - ▶ Distribute system to volunteers
 - ▶ Collect change requests, fix, redistribute
 - ▶ Collect statistics on beta use
- ▶ Shadowing
 - ▶ Collect or redistribute real-time use of existing system
 - ▶ Compare results
 - ▶ Collect statistics



The test-fix-test cycle

- ▶ When a failure occurs during testing:
 - ▶ Each failure report is entered into a failure tracking system.
 - ▶ It is then screened and assigned a priority.
 - ▶ Low-priority failures might be put on a *known bugs list* that is included with the software's *release notes*.
 - ▶ Some failure reports might be merged if they appear to result from the same defects.
 - ▶ Somebody is assigned to investigate a failure.
 - ▶ That person tracks down the defect and fixes it.
 - ▶ Finally a new version of the system is created, ready to be tested again.



Deciding when to stop testing

- ▶ All of the level 1 (“critical”) test cases must have been successfully executed.
- ▶ Certain pre-defined percentages of level 2 and level 3 test cases must have been executed successfully.
- ▶ The targets must have been achieved and are maintained for at least two cycles of ‘builds’.
 - ▶ A *build* involves compiling and integrating all the components.
 - ▶ Failure rates can fluctuate from build to build as:
 - Different sets of regression tests are run.
 - New defects are introduced.



The roles of people involved in testing

- ▶ The first pass of unit and integration testing is called *developer testing*.
 - ▶ Preliminary testing performed by the software developers who do the design.
- ▶ *Independent testing* may be performed by separate group.
 - ▶ They do not have a vested interest in seeing as many test cases pass as possible.
 - ▶ They develop specific expertise in how to do good testing, and how to use testing tools.



Test planning

- ▶ Decide on overall test strategy
 - ▶ What type of integration
 - ▶ Whether to automate system tests
 - ▶ Whether there is an independent test team
- ▶ Decide on the coverage strategy for system tests
 - ▶ Compute the number of test cases needed
- ▶ Identify the test cases and implement them
 - ▶ The set of test cases constitutes a “test suite”
 - ▶ May categorize into critical, important, optional tests (level 1, 2, 3)
- ▶ Identify a subset of the tests as regression tests



Testing performed by users and clients

▶ *Alpha testing*

- ▶ Performed by the user or client, but under the supervision of the software development team.

▶ *Beta testing*

- ▶ Performed by the user or client in a normal work environment.
- ▶ Recruited from the potential user population.
- ▶ An *open beta release* is the release of low-quality software to the general population.

▶ *Acceptance testing*

- ▶ Performed by users and customers.
- ▶ However, the customers do it on their own initiative.



Inspections Vs Testing

- ▶ Both testing and inspection rely on different aspects of human intelligence.
- ▶ Testing can find defects whose consequences are obvious but which are buried in complex code.
- ▶ Inspecting can find defects that relate to maintainability or efficiency.
- ▶ The chances of mistakes are reduced if both activities are performed.



Testing or inspecting, which comes first?

- ▶ It is important to inspect software *before* extensively testing it.
- ▶ The reason for this is that inspecting allows you to quickly get rid of many defects.
- ▶ Even before developer testing



Packaging for Delivery

- ▶ Software we deliver to the user must include
 - ▶ Executable in a convenient format e.g. EXE, JAR file
 - ▶ Release notes
 - ▶ User documentation: instructions on usage
 - ▶ Tutorials, user manuals, “getting started” instructions
 - ▶ Installation instructions
- ▶ May create “installables”
 - ▶ Compressed packages e.g. zip files, tar files
 - ▶ Scripts that automate installation procedures

