Software Testing

No issue is meaningful unless it can be put to the test of decisive verification.

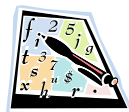
C.S. Lewis, 1934

Outline of the Lecture

- Some Notations
- Testing Level
  - Integration Testing
  - Component/Unit/Module/Basic Testing
  - System Testing Steps
    - Function testing / Thread testing
    - Performance testing
    - Acceptance testing
    - Installation testing
- Test Automation
- Termination Problem

Testing a ballpoint pen

- Does the pen write in the right color, with the right line thickness?
- Is the logo on the pen according to company standards?
- Is it safe to chew on the pen?
- Does the click-mechanism still work after 100 000 clicks?
- Does it still write after a car has run over it?



What is expected from this pen?
Intended use!!

Goal: develop software to meet its intended use!
But: human beings make mistake!

bridge

automobile

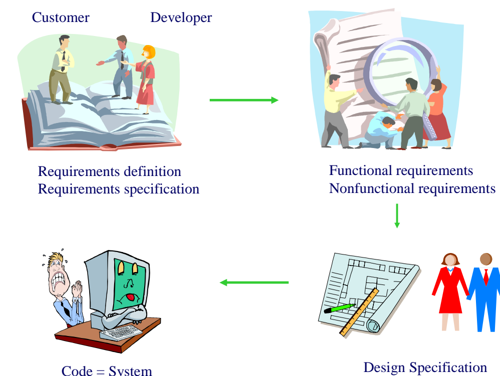
television

word processor

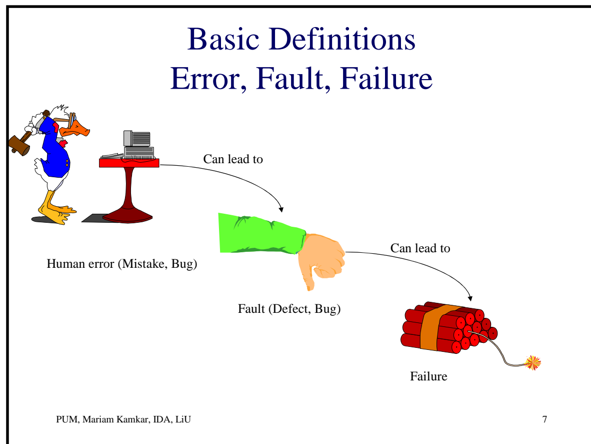
⇒ Product of any engineering activity must be verified against its requirements throughout its development.

- Verifying bridge = verifying design, construction, process,…
- Software must be verified in much the same spirit. In this lecture, however, we shall learn that verifying software is perhaps <u>more difficult</u> than verifying other engineering products.

We shall try to clarify why this is so.



Basic Definitions
Error, Fault, Failure



Debugging vs. Testing

- **Debugging**: to find the bug
- **Testing**: to demonstrate the existence of a fault
  - fault identification
  - fault correction / removal

Types of Faults
(dep. on org. IBM, HP)

- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str. size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: speed in req.

Types of Faults (cont.)

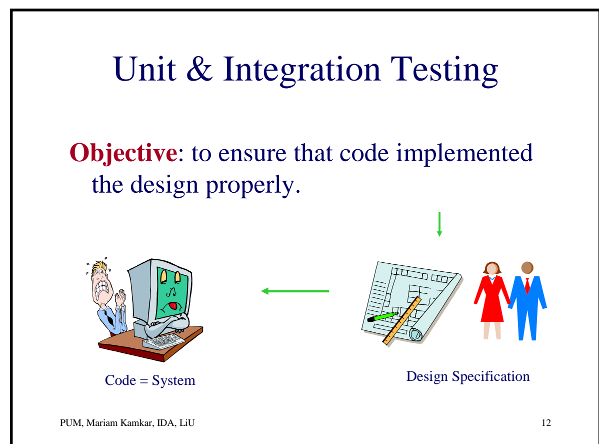
- Recovery: power failure
- Hardware & System Software: modem
- Standards & Procedure: organizational standard; difficult for programmers to follow each other

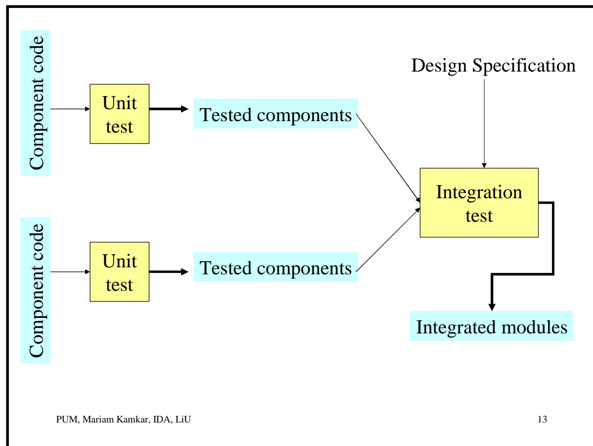
Integration Testing

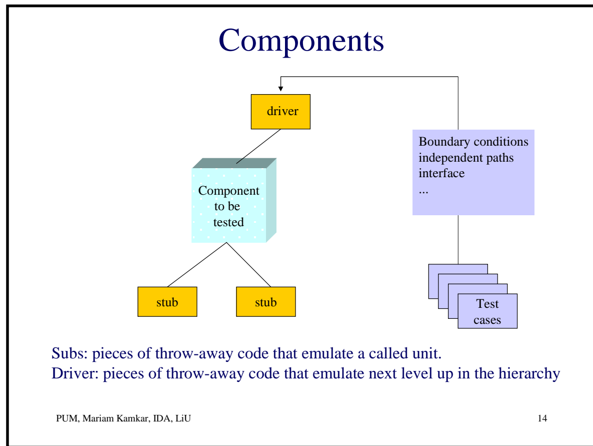
- Top-down
- Bottom-up
- Big bang
- Sandwich

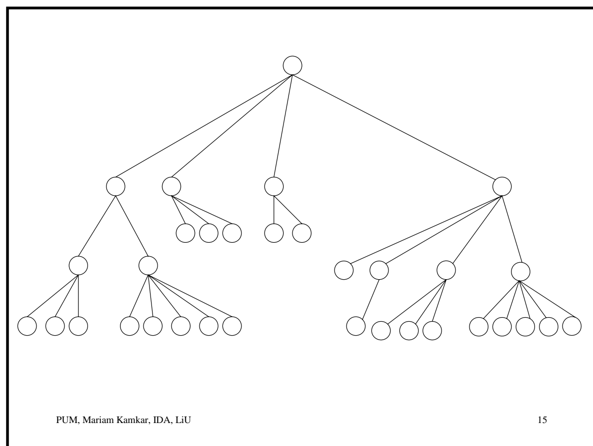
Unit & Integration Testing

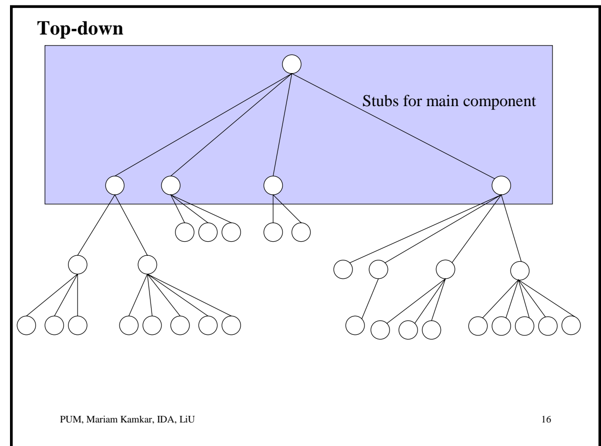
Objective: to ensure that code implemented the design properly.

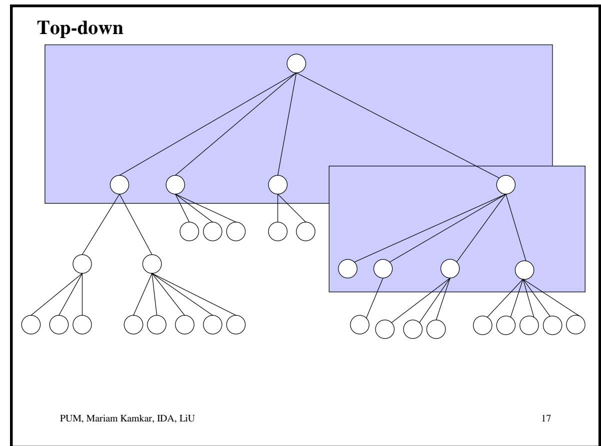


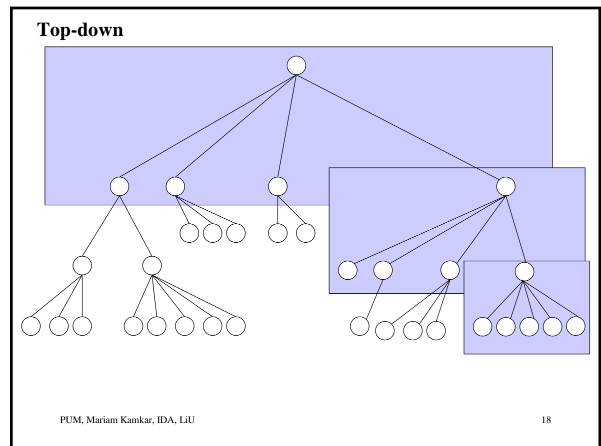

Component code
Unit test
Tested components
Component code
Unit test
Tested components
Design Specification
Integration test
Integrated modules
PUM, Mariam Kamkar, IDA, LiU
13

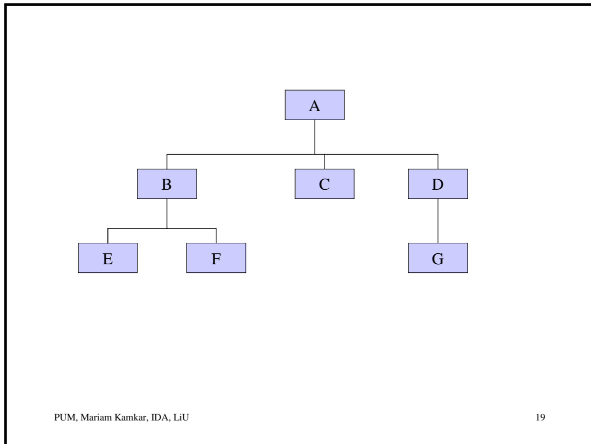

Components
driver
Component to be tested
stub
stub
Boundary conditions
independent paths
interface
...
Test cases
Subs: pieces of throw-away code that emulate a called unit.
Driver: pieces of throw-away code that emulate next level up in the hierarchy
PUM, Mariam Kamkar, IDA, LiU
14

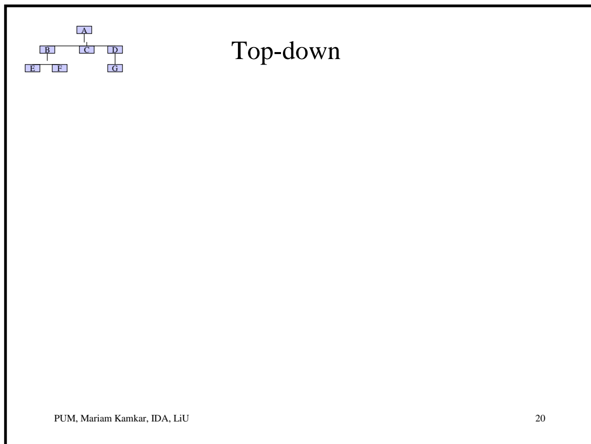

PUM, Mariam Kamkar, IDA, LiU
15

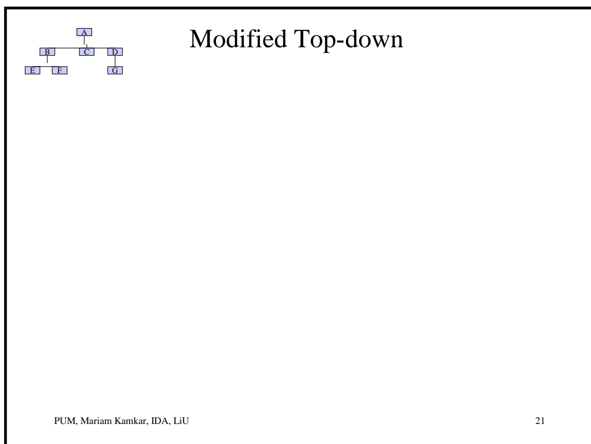

Top-down
Stubs for main component
PUM, Mariam Kamkar, IDA, LiU
16

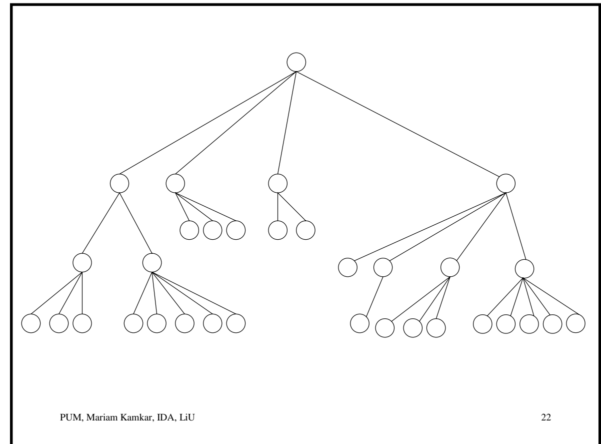

Top-down
PUM, Mariam Kamkar, IDA, LiU
17

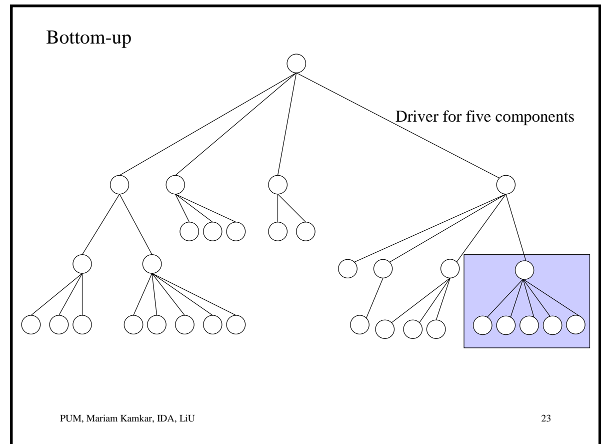

Top-down
PUM, Mariam Kamkar, IDA, LiU
18

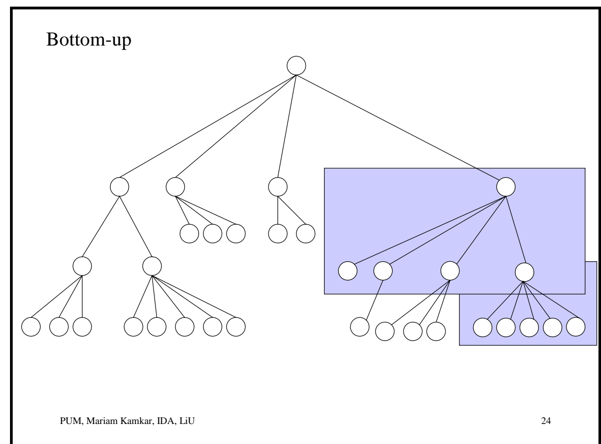

A
B
C
D
E
F
G
PUM, Mariam Kamkar, IDA, LiU
19

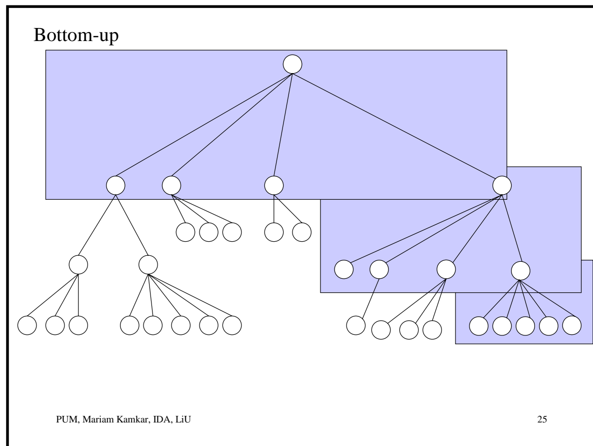

Top-down
PUM, Mariam Kamkar, IDA, LiU
20

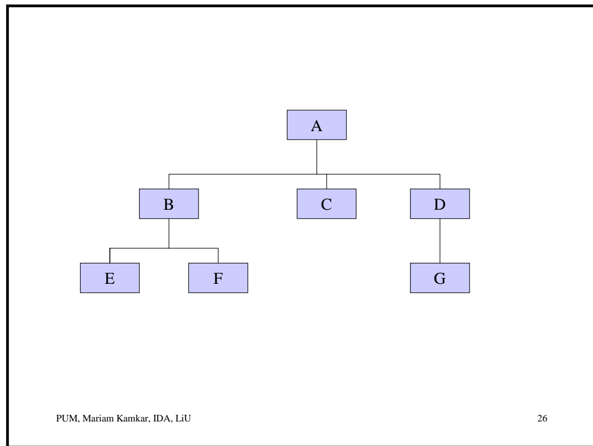

Modified Top-down
PUM, Mariam Kamkar, IDA, LiU
21

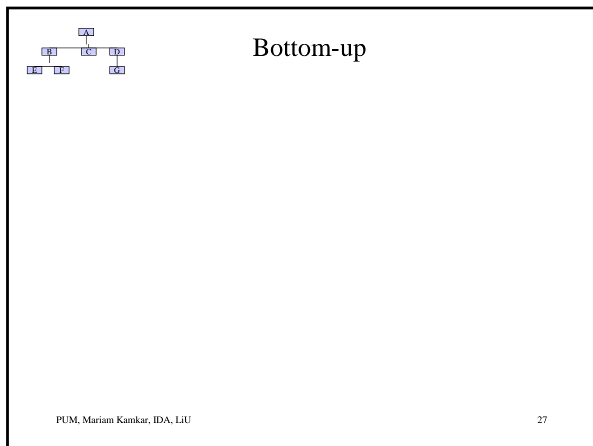

PUM, Mariam Kamkar, IDA, LiU
22

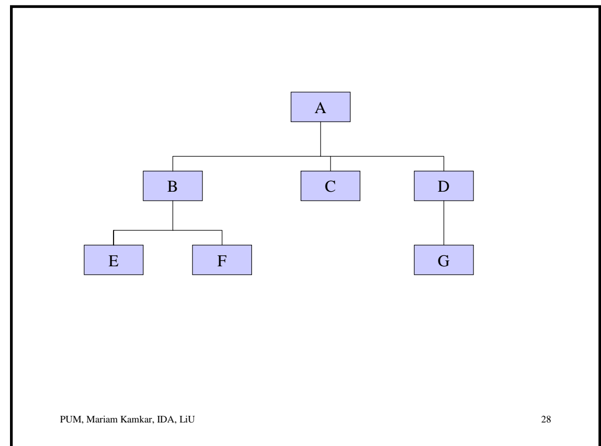

Bottom-up
Driver for five components
PUM, Mariam Kamkar, IDA, LiU
23

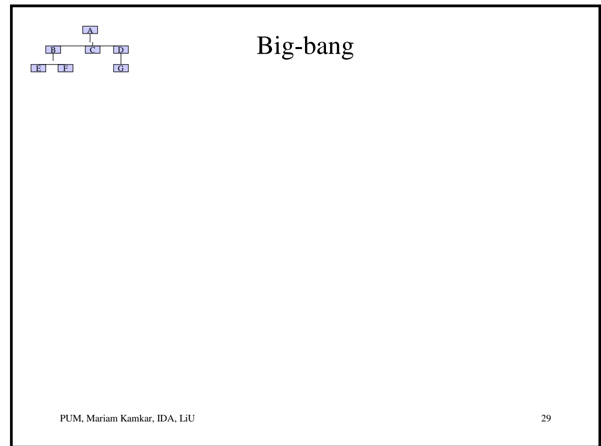

Bottom-up
PUM, Mariam Kamkar, IDA, LiU
24

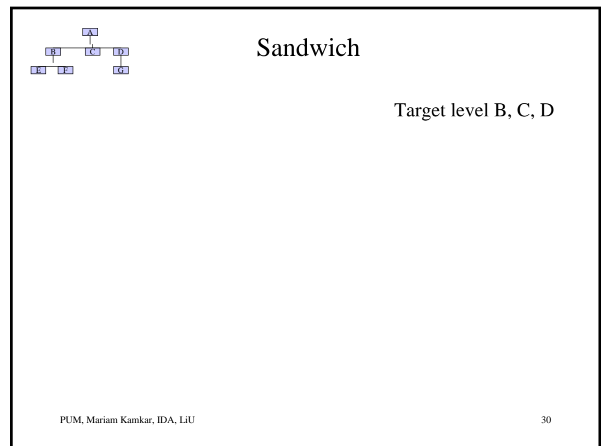

Bottom-up
PUM, Mariam Kamkar, IDA, LiU
25

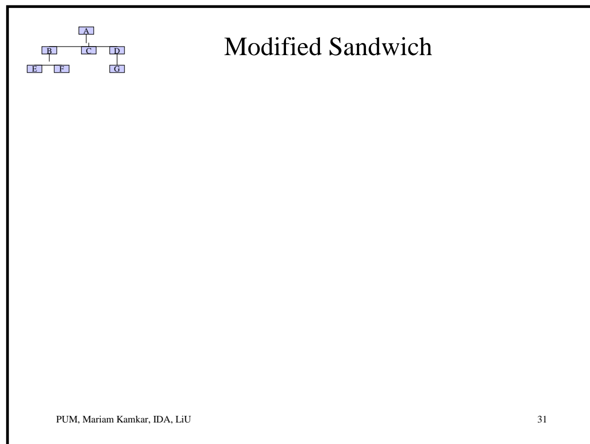

A
B
C
D
E
F
G
PUM, Mariam Kamkar, IDA, LiU
26


Bottom-up
PUM, Mariam Kamkar, IDA, LiU
27


A
B
C
D
E
F
G
PUM, Mariam Kamkar, IDA, LiU
28


Big-bang
PUM, Mariam Kamkar, IDA, LiU
29


Sandwich
Target level B, C, D
PUM, Mariam Kamkar, IDA, LiU
30

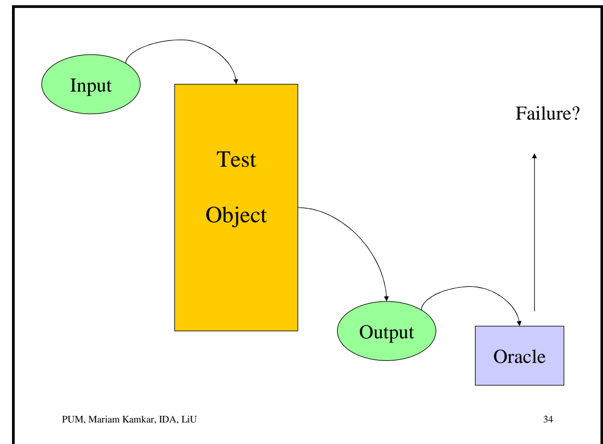


Comparison of Integration Strategies

	Top-down	Modified Top-Down	Bottom-up	Big-bang	Sandwich	Modified Sandwich
Integration	Early	Early	Early	Late	Early	Early
Time to basic working program	Early	Early	Late	Late	Early	Early
Driver needed	No	Yes	Yes	Yes	Yes	Yes
Stubs needed	Yes	Yes	No	Yes	Yes	Yes

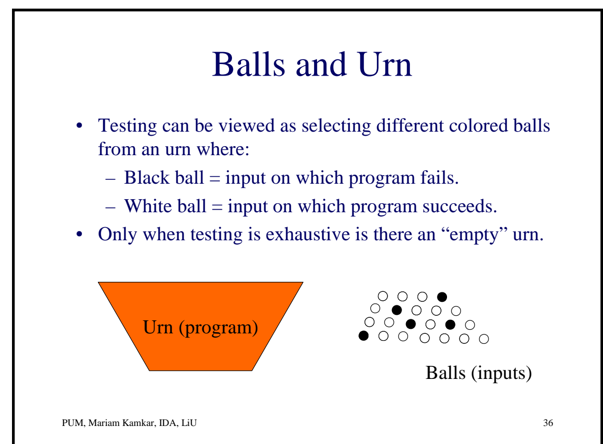
Unit Testing

- Code Reviews:
  - Walkthroughs
  - Inspections
- White/Open box testing
- Black/Close box testing



Two Types of Oracles

- Human: an expert that can examine an input and its associated output and determine whether the program delivered the correct output for this particular input.
- Automated: a system capable of performing the above task.



A program that always fails

A correct program

A typical program

Walkthroughs

(you lead the discussion)

Design, Code, Chapter of user's guide,…

- presenter
- coordinator
- secretary
- maintenance oracle
- standards bearer
- user representative

Inspections

(originally introduced by Fagan 1976)
(the review team leads the discussion)

- overview (code, inspection goal)
- preparation (individually)
- reporting
- rework
- follow-up

Inspections (cont.)

some classical programming errors

- Use of un-initialized variables
- Jumps into loops
- Non-terminating loops
- Incompatible assignments
- Array indexes out of bounds
- Off-by-one errors
- Improper storage allocation or de-allocation
- Mismatches between actual and formal parameters in procedure calls

Discovery activity	Faults found per thousand lines of code
Requirements review	2.5
Design review	5.0
Code inspection	10.0
Integration test	3.0
Acceptance test	2.0

Jons, S et al, Developing international user information. Bedford, MA: Digital Press, 1991.

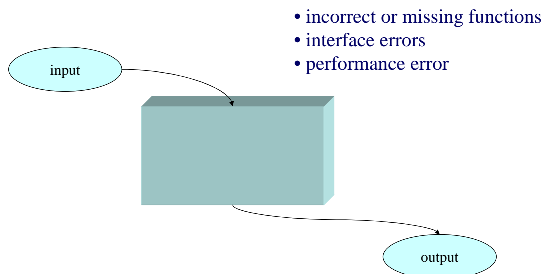
Experiments

- 82% of faults discovered during design & code inspection (Fagan)
- 93% of all faults in a 6000-lines application were found by inspections (Ackerman, et al 1986)
- 85% of all faults removed by inspections from examining history of 10 million lines of code (Jones 1977)
- Inspections : finding code faults
- Prototyping: requirements problem

Proving code correct

- Formal proof techniques
- Symbolic execution
- Automated theorem proving

Black box / Closed box testing



Block-Box Testing

- Definition: a strategy in which testing is based on requirements and specifications.
- Applicability: all levels of system development
  - Unit
  - Integration
  - System
- Disadvantages: never be sure of how much of the system under test has been tested.
- Advantages: directs tester to choose subsets to tests that are both efficient and effective in finding defects.

Black-box Testing Techniques

- Exhaustive testing
- Equivalence class testing (Equivalence Partitioning)
- Boundary value analysis

Exhaustive testing

- **Definition**: testing with every member of the input value space.
- Input value space: the set of all possible input values to the program.

Equivalence Class Testing

- Equivalence Class (EC) testing is a technique used to reduce the number of test cases to a manageable level while still maintaining reasonable test coverage.
- Each EC consists of a set of data that is treated the same by the module or that should produce the same result. Any data value within a class is ***equivalent***, in terms of testing, to any other value.

Identifying the Equivalence Classes

Taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups:

- Input condition
  - range of values x: 1-50
- Valid equivalence class
  - 1< x < 50
- Invalid equivalence classes
  - x < 1
    x > 50

Guidelines

1. If an input condition specifies a *range* of values; identify one valid EC and two invalid EC.
2. If an input condition specifies the *number* (e.g., one through 6 owners can be listed for the automobile); identify one valid EC and two invalid EC (- no owners; - more than 6 owners).
3. If an input condition specifies a set of input values and there is reason to believe that each is handled differently by the program; identify a valid EC for each and one invalid EC.
4. If an input condition specifies a "must be" situation (e.g., first character of the identifier must be a letter); identify one valid EC (it is a letter) and one invalid EC (it is not a letter)
5. If there is any reason to believe that elements in an EC are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

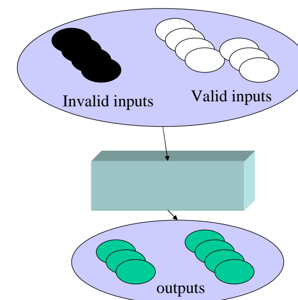
Identifying the Test Cases

1. Assign a unique number to each EC.
2. Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered valid ECs as possible.
3. Until all invalid ECs have been covered by test cases, write a test case that cover one, and only one, of the uncovered invalid ECs

Applicability and Limitations

- Most suited to systems in which much of the input data takes on values within ranges or within sets.
- It makes the assumption that data in the same EC is, in fact, processed in the same way by the system. The simplest way to validate this assumption is to ask the programmer about their implementation.
- EC testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs or outputs that can be partitioned based on the system's requirements.

Equivalence partitioning



Specification: the program accepts four to eight inputs which are 5 digit integers greater than 10000.

Boundary Value Testing

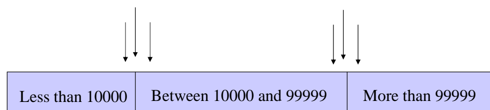
Boundary value testing focuses on the boundaries simply because that is where so many defects hide. The defects can be in the requirements or in the code.

The most efficient way of finding such defects, either in the requirements or the code, is through inspection (Software Inspection, Gilb and Graham’s book).

Technique

1. Identify the ECs.
2. Identify the boundaries of each EC.
3. Create test cases for each boundary value by choosing one point on the boundary, one point just below the boundary, and one point just above the boundary.

Boundary value analysis



Applicability and Limitations

Boundary value testing is equally applicable at the unit, integration, system, and acceptance test levels. All it requires are inputs that can be partitioned and boundaries that can be identified based on the system’s requirements.

White box testing

- logical decision
- loops
- internal data structure
- paths
- ...

Coverage!!

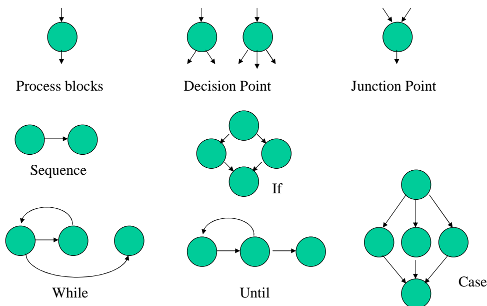
White box Techniques

- Control flow testing
- Data flow testing

White-box Testing

- Definition: a strategy in which testing is based on the internal paths, structure, and implementation of the software under test (SUT)
- Applicability: all levels of system development (path testing!)
  - Unit
  - Integration
  - System
  - Acceptance
- Disadvantages: 1) number of execution paths may be so large; 2) test cases may not detect data sensitivity; 3) assumes that control flow is correct (nonexistent paths!); 4) tester must have programming skills.
- Advantages: tester can be sure that every path have been identified and tested.

Control Flow Graphs



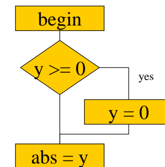
Definition: Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are statement fragments, and edges represent flow of control (a complete statement is a "default" statement fragment).

Levels of Coverage (test coverage metrics)

- Statement (Line) coverage
- Decision (Branch) coverage
- Condition coverage
- Decision/Condition coverage
- Multiple Condition coverage
- Path coverage

Statement Coverage

```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```

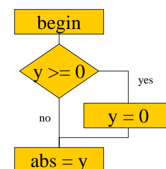


<u>test case-1:</u>

input: y = ?
expected result: ?
actual result: ?

Branch Coverage

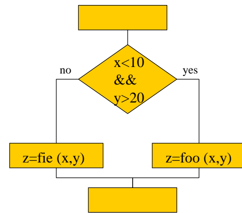
```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```



<u>test case-1(yes):</u>	<u>test case-2 (no) :</u>
input: y = 0	*input:* y = ?
expected result: 0	*expected result:* ?
actual result: 0	*actual result:* ?

Condition Coverage

Begin
if (x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;

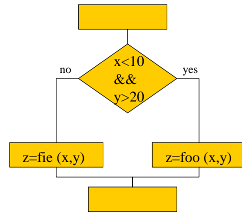


<u>test case-1:</u>
input: x = ?, y = ?
expected result: ?
actual result: ?

<u>test case-2 :</u>
input: x = ?, y = ?
expected result: ?
actual result: ?

Decision/Condition Coverage

Begin
if (x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;



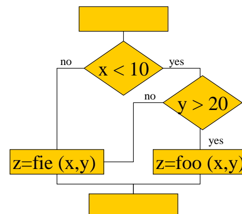
<u>test case-1 (T,T,yes):</u>
input: x = ?, y = ?
expected result: ?
actual result: ?

<u>test case-2 (F,F,no):</u>
input: x = ?, y = ?
expected result: ?
actual result: ?

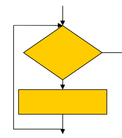
Multiple Condition Coverage

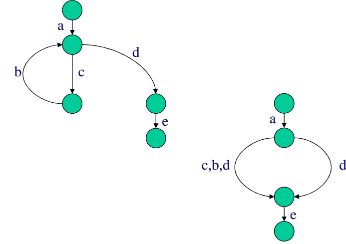
Begin
if (x < 10 && y > 20) {
z = foo (x,y); else z =fie (x,y);
}
end;

	x < ?	y > ?
test-case-1:	t	t
test-case:2	t	f
test-case-3:	f	t
test-case-4	f	f

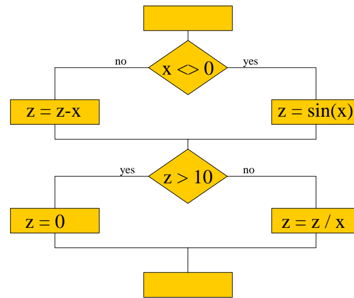


Path with loops





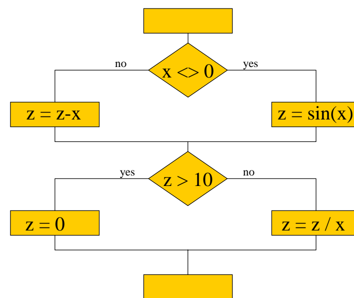
Path Coverage



(n, y) x = ?, z = ?
(y, n) x = ?, z = ?

(n, n) x = ?, z = ?
(n, y) x = ?, z = ?
(y, n) x = ?, z = ?
(y, y) x = ?, z = ?

Path Coverage



(n, y) x = 0, z = 12
(y, n) x = 2, z = 6

(n, n) x = 0, z = 7
(n, y) x = 0, z = 13
(y, n) x = 1, z = 5
(y, y) x = 2, z = 15

Path Coverage

- All possible execution paths
- Question: How do we know how many paths to look for?
- Answer: The computation of cyclomatic complexity

Computation of cyclomatic complexity

Cyclomatic complexity has a foundation in graph theory and is computed in the following ways:

1. Cyclomatic complexity V(G), for a flow graph, G, is defined as:

   V(G) = E – N + 2
   E: number of edges
   N: number of nodes

2. Cyclomatic complexity V(G), for a flow graph, G, with only binary decisions, is defined as:

   V(G) = P + 1
   P: number of binary decision

Data Flow Testing

DEF(S) = {x | statement S contains a definition of variable x}
USE(S) = {x | statement S contains a use of variable x}
DEF-USE-Chain (du chain) = [x, S, S']

S_1: i = 1;

S_2: while (i <= n)

Data Flow testing

```
s = 0;
i = 1;          ?
s = 1;
while (i <= n)
        {
                s + = i;
                i ++
        }
print (s);
print (i);
print (n);
```

du: def-use
dk: def-kill
...

Program Slicing

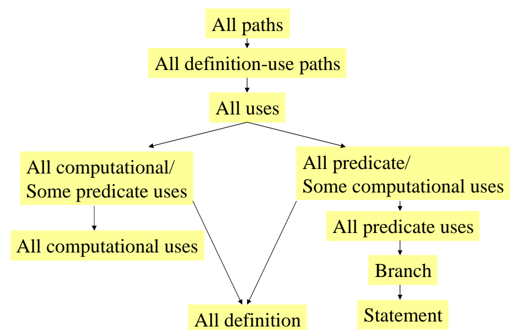
```
s = 0;
i = 1;
while (i <= n)
    {
        s + = i;
        i ++
    }
print (s);
print (i);
print (n);
```

```
i = 1;
while (i <= n)
    {

        i ++
    }

print (i);
```

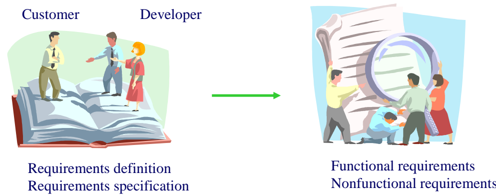

Relative strengths of test strategies (B. Beizer 1990)

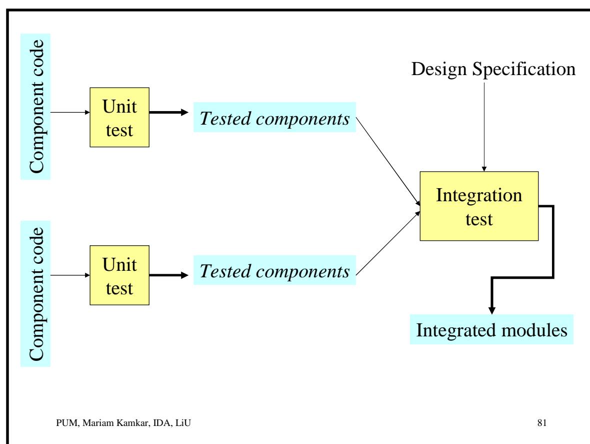


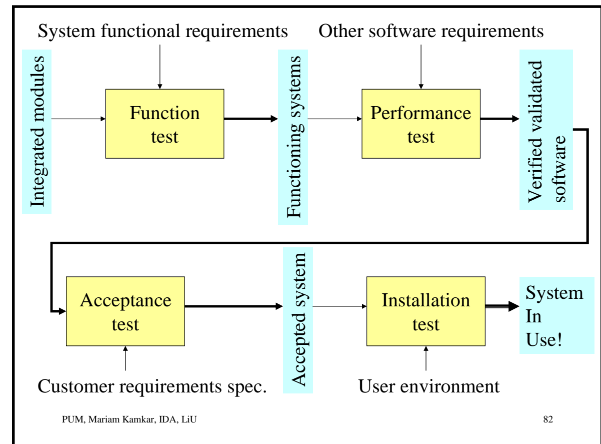
System Testing

- Function testing
- Performance testing
- Acceptance testing
- Installation testing

Objective: to ensure that the system does what the customer wants it to do.







Function testing

(testing one function at a time)

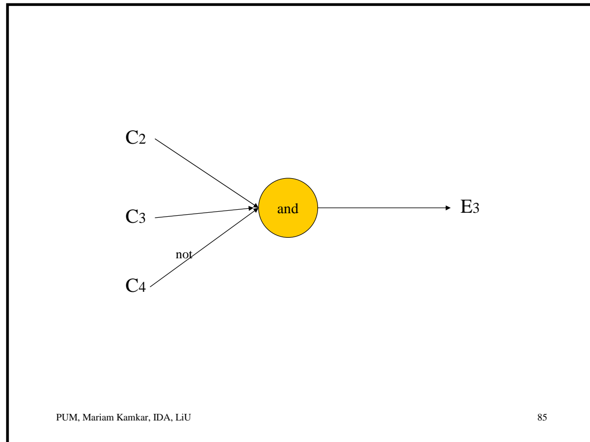
functional requirements

- have a high probability of detecting a fault
- use a test team independent of the designers and programmers
- know the expected actions and output
- test both valid and invalid input
- never modify the system just to make testing easier
- have stopping criteria

Cause-Effect

(test case generation from req.)

Causes	***Effects***
C_1: command is credit	E_1: print "invalid command"
C_2: command is debit	E_2: print "invalid account number"
C_3: account number is valid	E_3: print "debit amount not valid"
C_4: transaction amount is valid	E_4: debit account print
	E_5: credit account print



Performance Testing
nonfunctional requirements

- Stress tests
- Volume tests
- Configuration tests
- Compatibility tests
- Regression tests
- Security tests
- Timing tests
- Environment tests
- Quality tests
- Recovery tests
- Maintenance tests
- Documentation tests
- Human factors tests / usability tests

Acceptance testing
customers, users need

- Benchmark test: a set of special test cases
- Pilot test: everyday working
  - Alpha test: at the developer's site, controlled environment
  - Beta test: at one or more customer site.
- Parallel test: new system in parallel with previous one

Installation testing
users site

Acceptance test at developers site
→ installation test at users site,
otherwise may not be needed!!

Test Planing

- Establishing test objectives
- Designing test cases
- Writing test cases
- Testing test cases
- Executing tests
- Evaluating test results

Automated Testing Tools

- Code Analysis tools
  - Static, Dynamic
- Test execution tools
  - Capture-and-Replay
  - Stubs & Drivers
  - Comparators
- Test case generator

Termination Problem
How decide when to stop testing

- The main problem for managers!
- Termination takes place when
  - resources (time & budget) are over
  - found the seeded faults
  - some coverage is reached

PUM, Mariam Kamkar, IDA, LiU 91

Oracle

Scaffolding

What can be automated?

Test case generation

Termination

PUM, Mariam Kamkar, IDA, LiU 92

Real life examples

- First U.S. space mission to Venus failed. (reason: missing comma in a Fortran do loop)
- December 1995: AA, Boeing 575, mountain crash in Colombia, 159 killed. Incorrect one-letter computer command (Cali, Bogota 132 miles in opposite direction, have same coordinate code)
- June 1996: Ariane-5 space rocket, self-destruction, $500 million. (reason: reuse of software from Ariane-4 without recommended testing).

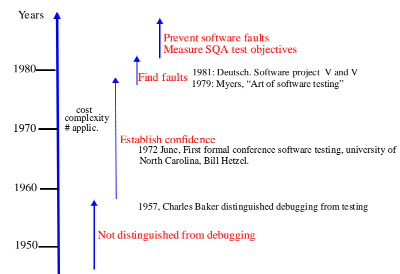
PUM, Mariam Kamkar, IDA, LiU 93

Real life examples

- Australia: Man jailed because of computer glitch. He was jailed for traffic fine although he had actually paid it for 5 years ago.
- Dallas Prisoner released due to program design flaw: He was temporary transferred from one prison to another (witness). Computer gave him "temporary assignment".

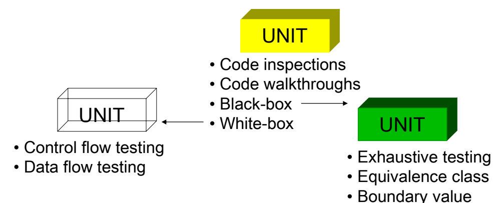
PUM, Mariam Kamkar, IDA, LiU 94

Goals of software testing: Historical Evolution

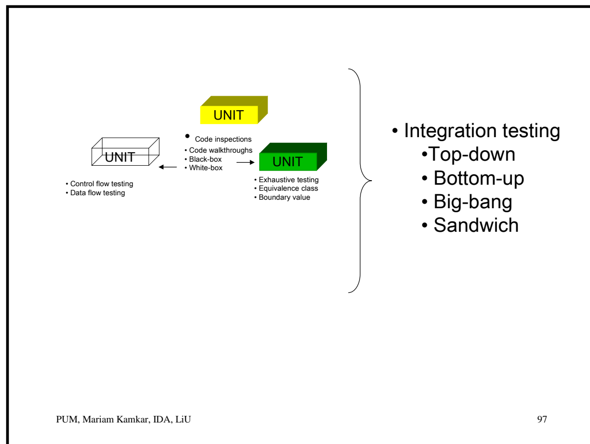


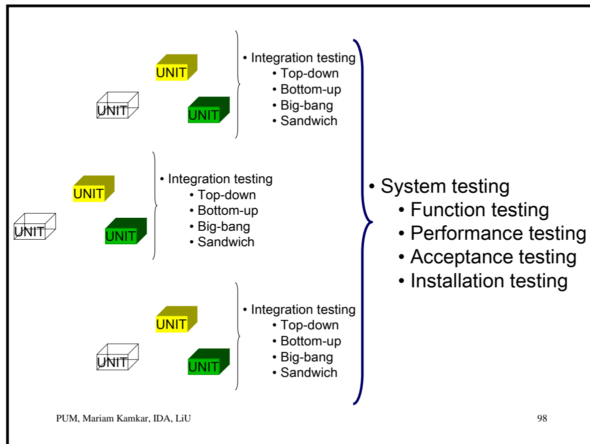
PUM, Mariam Kamkar, IDA, LiU 95

Summary



PUM, Mariam Kamkar, IDA, LiU 96





And …

Testing can show the presence, but never the absence of errors in software.

E. Dijkstra, 1969