Software Testing

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

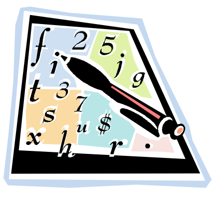
C.S. Lewis, 1934

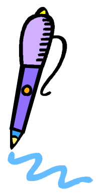
Testing a ballpoint pen

- 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?

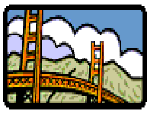
What is expected from this pen?

Intended use!!

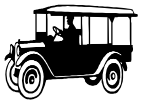




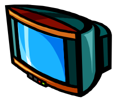
Goal: develop software to meet its intended use!
But: human beings make mistake!



bridge



automobile



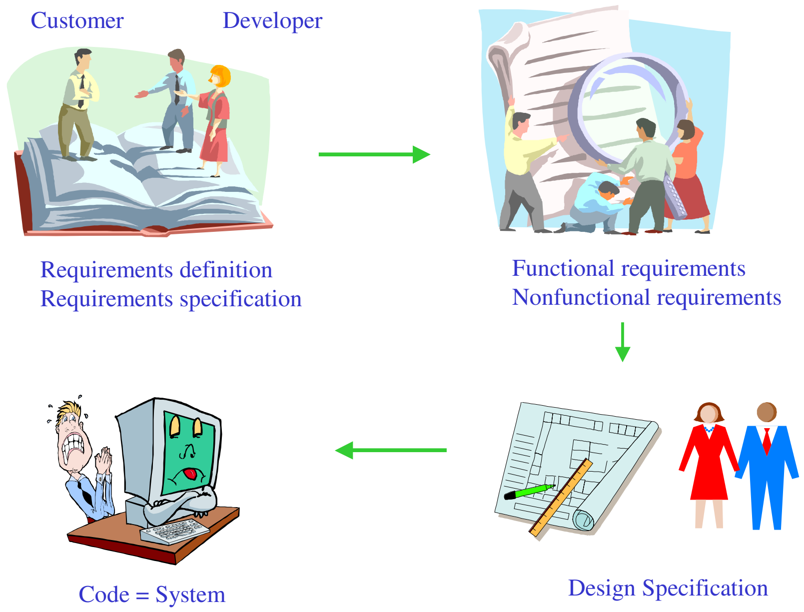
television

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.

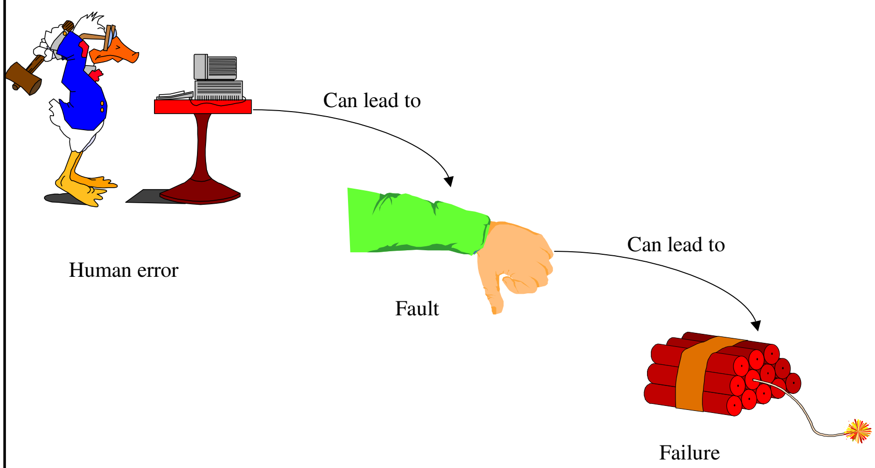
We shall try to clarify why this is so.



Outline

- Some notations
- Integration testing
- Component/Unit/Module/Basic testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing
- Real life examples

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)

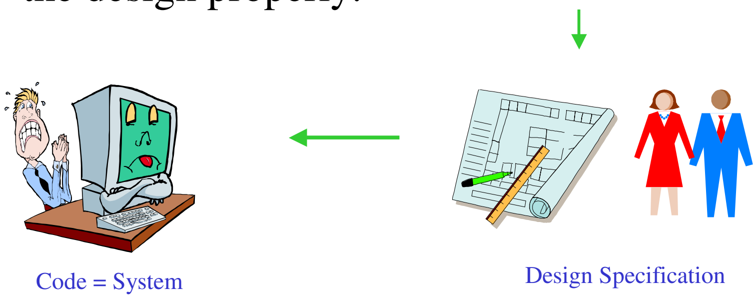
- Algorithmic: division by zero
- Computation & Precision: order of op
- Documentation: doc - code
- Stress/Overload: data-str size (dimensions of tables, size of buffers)
- Capacity/Boundary: x devices, y parallel tasks, z interrupts
- Timing/Coordination: real-time systems
- Throughout/Performance: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

Types of Faults

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

Unit & Integration Testing

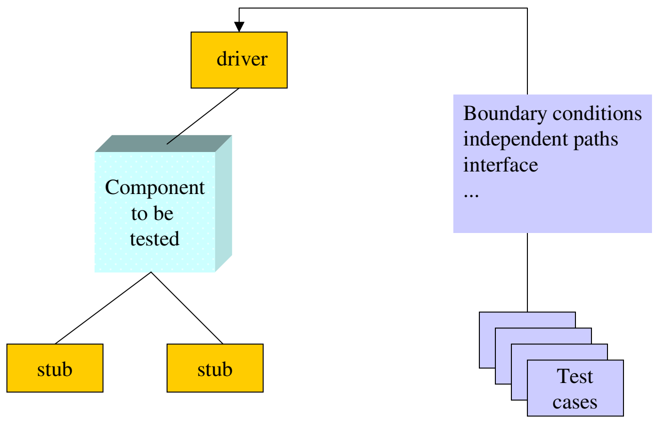
Objective: to ensure that code implemented the design properly.

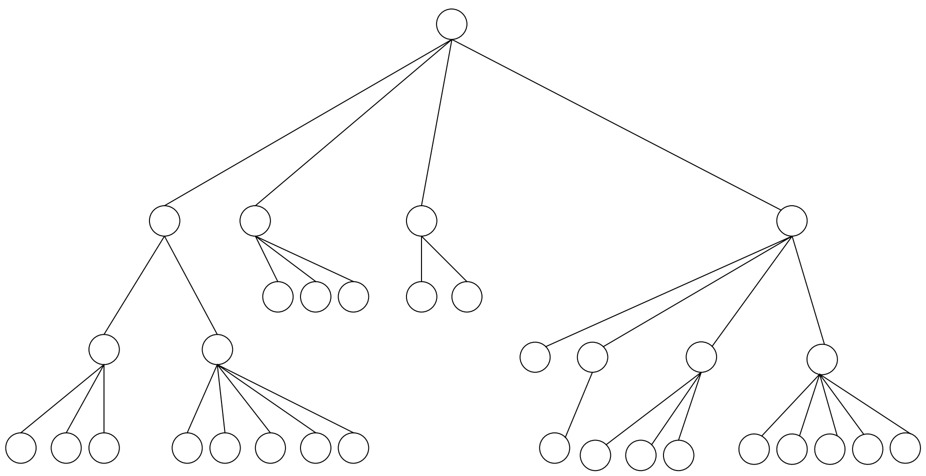


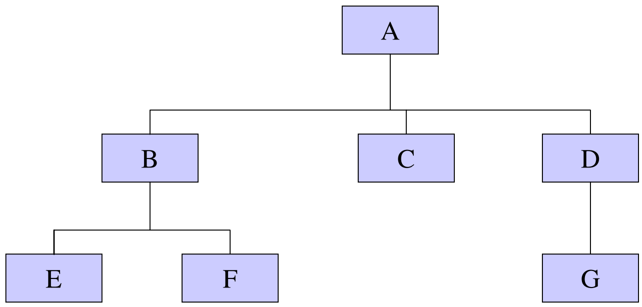
Classes of Integration Testing

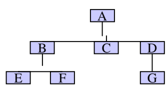
- Top-down
- Bottom-up
- Big bang
- Sandwich

Components

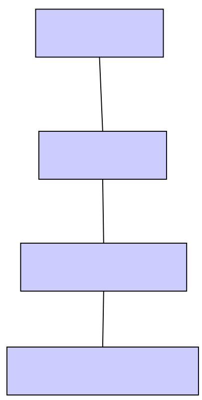

driver
Component
to be
tested
stub
stub
Boundary conditions
independent paths
interface
...
Test
cases



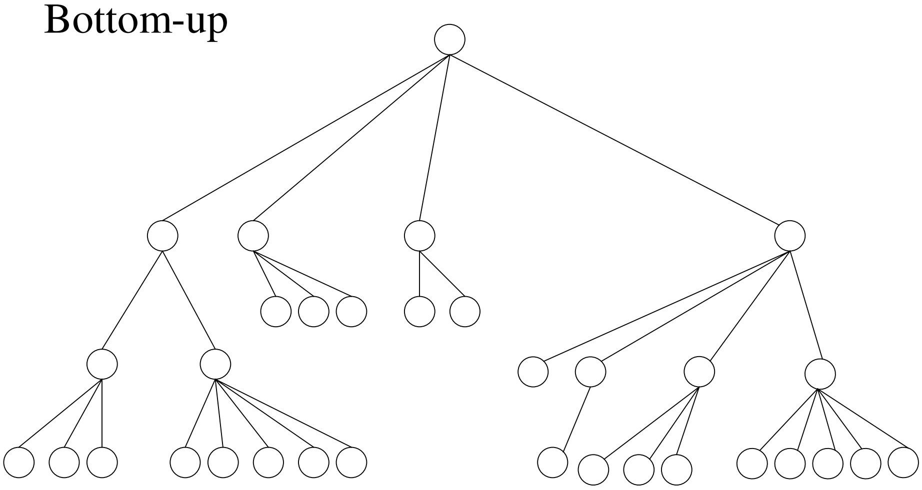




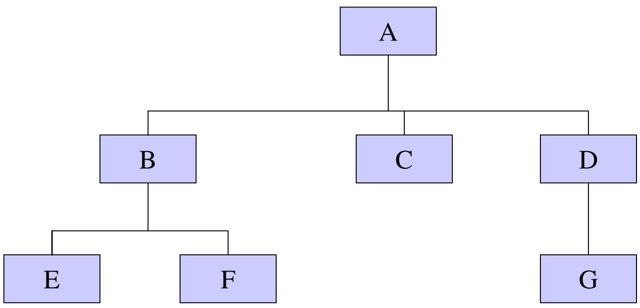
Top-down



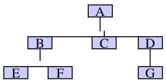
Bottom-up



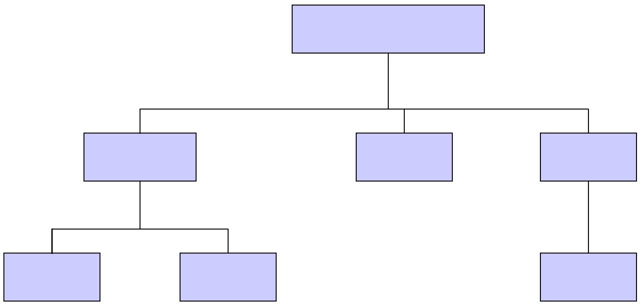
January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU 17



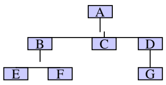
January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU 18



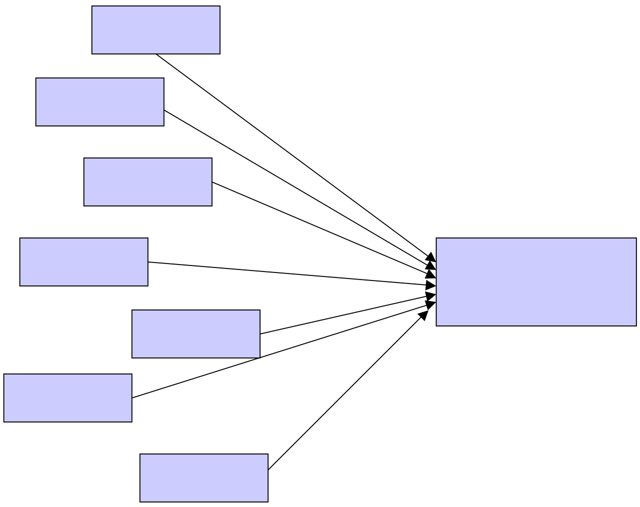
Bottom-up



January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU 19

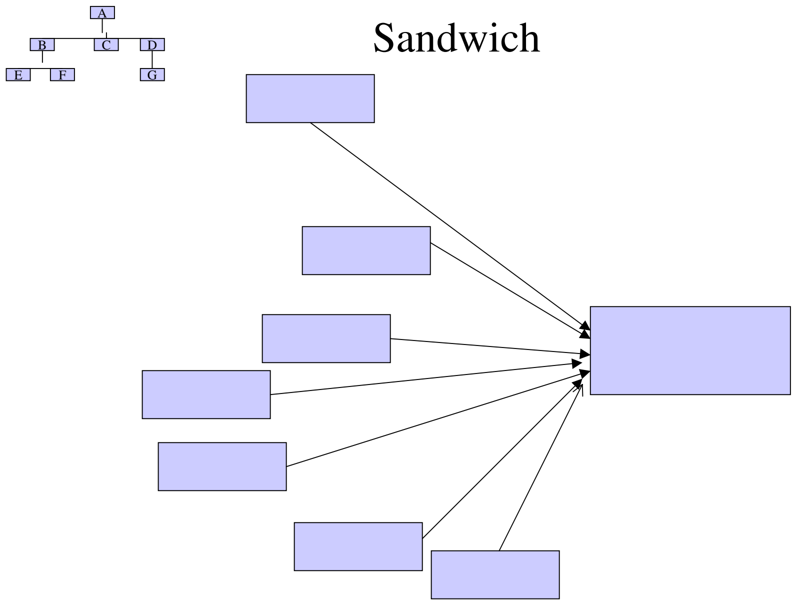


Big-bang

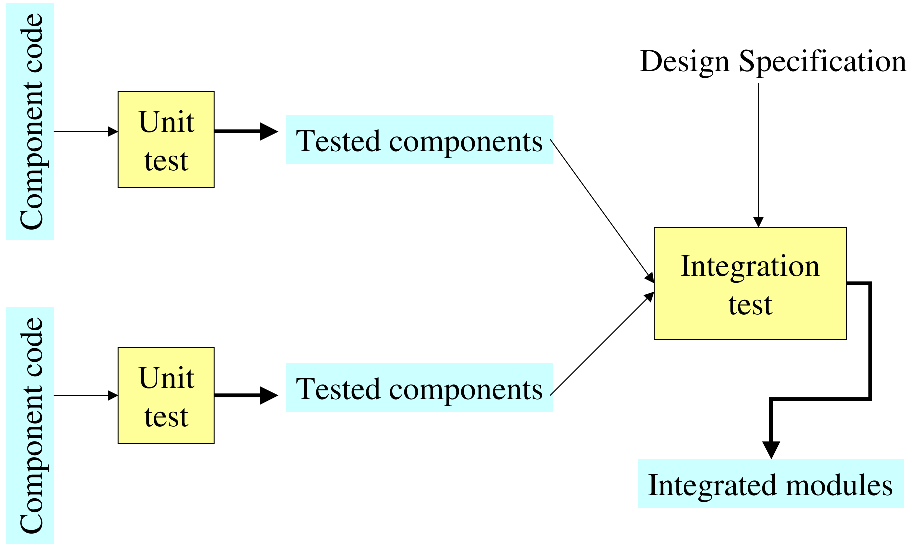


January 2006 CUGS, SE, Mariam Kamkar, IDA, LiU 20

Sandwich

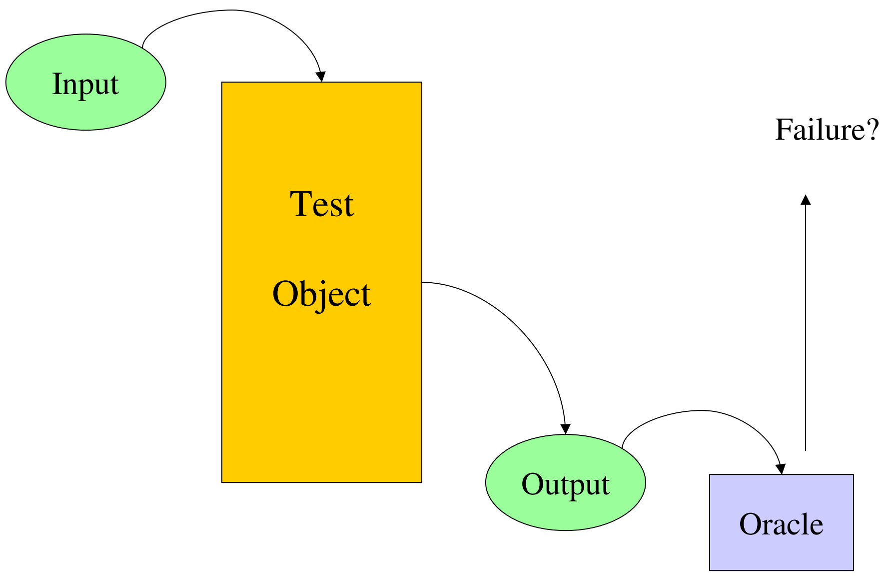


	Top-down	Bottom-up	Big-bang	Sandwich
Time to a basic working program	Early	Late	Late	Early
Driver needed	No	Yes	Yes	In part
Stubs needed	Yes	No	Yes	In part



Unit Testing

- Code Inspections
- Code Walkthroughs
- Open box testing
- Black box testing

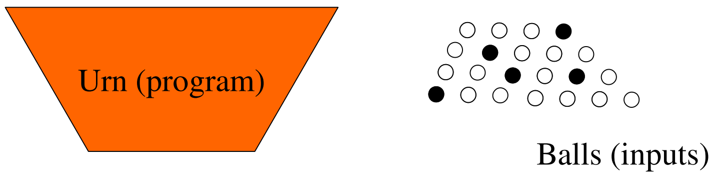


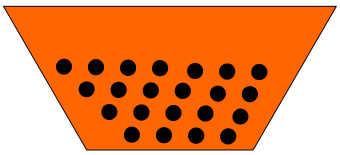
Two Types of Oracles

- 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.

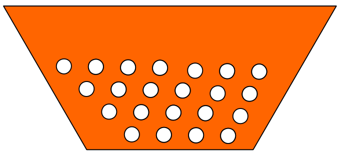
Balls and Urn

- Testing can be viewed as selecting different colored balls from an urn where:
  - Black ball = input on which program fails.
  - White ball = input on which program succeeds.
- Only when testing is exhaustive is there an “empty” urn.

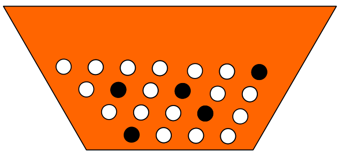




A program that always fails



A correct program



A typical program

Inspection
(originally introduced by Fagan 1976)

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

Inspection (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

Walkthroughs

design, code, chapter of user's guide,…

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

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

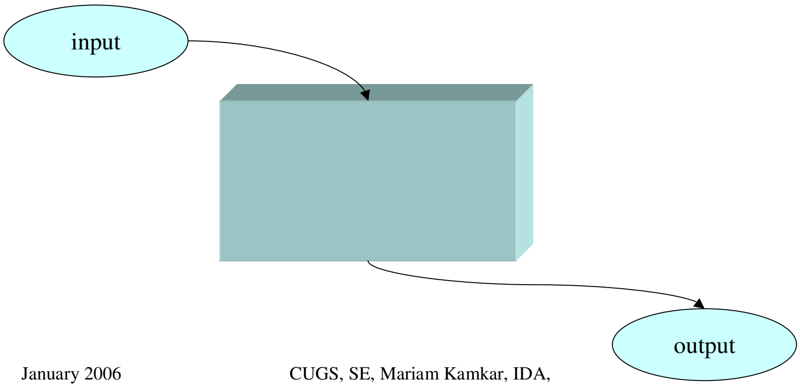
- 82% of faults discovered during design & code inspection (Fagan)
- 93% of all faults in a 6000-lines application were found by inspections (Ackerman, et al 1986)
- 85% of all faults removed by inspections from examining history of 10 million lines of code (Jones 1977)
- Inspections : finding code faults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

Proving code correct

- Formal proof techniques
- Symbolic execution
- Automated theorem proving

Black box / Closed box testing

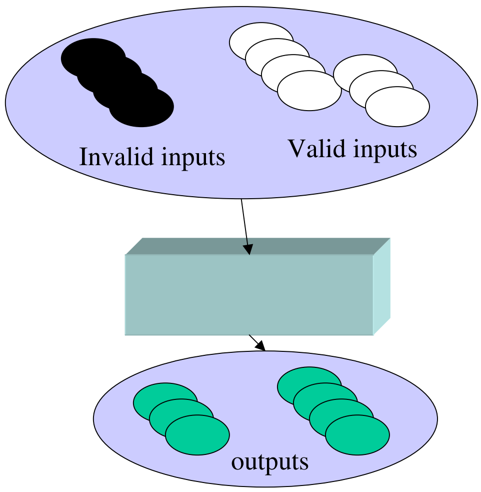
- incorrect or missing functions
- interface errors
- performance error



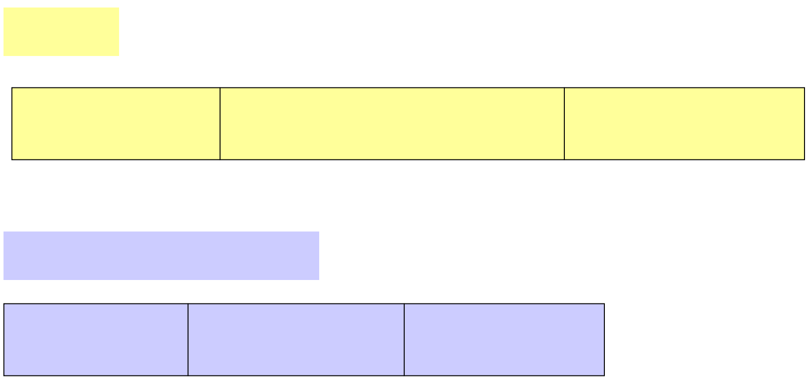
Black box testing

- Equivalence partitioning
- Boundary value analysis
- Exhaustive testing

Equivalence partitioning



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

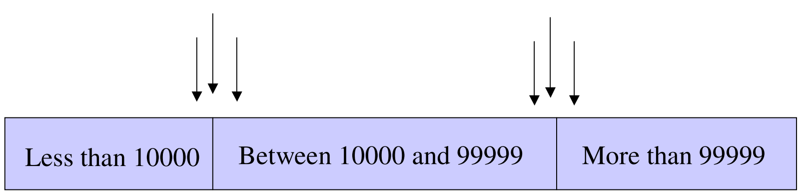


Guidelines

If an input condition specifies

- A range: one valid and two invalid equivalence classes.
- A specific value: one valid and two invalid equivalence classes.
- A member of a set: one valid and one invalid equivalence classes.
- A boolean: one valid and one invalid class.

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.

Glass box testing!
White box testing!
Open box testing!
Clear box testing!

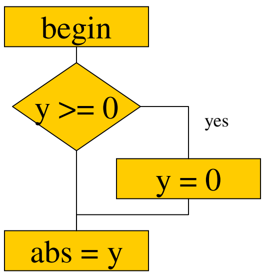
Glass box testing

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

Coverage!!

Statement Coverage

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



test case-1 (yes):
input: y = ?
expected result: ?
actual result: ?

Branch Coverage

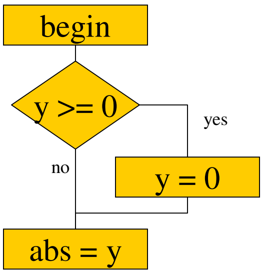
Begin

if (y >= 0)

then y = 0;

abs = y;

end;



test case-1(yes):
input: y = 0
expected result: 0
actual result: 0

test case-2 (no):
input: y = ?
expected result: ?
actual result: ?

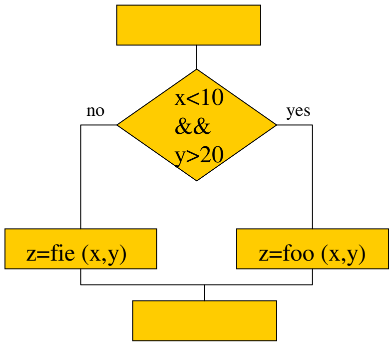
Begin

if (x < 10 && y > 20) {

z = foo (x,y); else z =fie (x,y);

}

end;

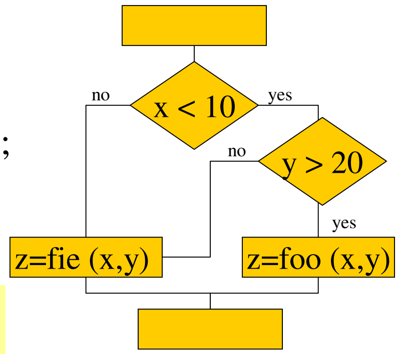


test case-1 (yes):
input: x = ?, y = ?
expected result: ?
actual result: ?

test case-2 (no):
input: x = ?, y = ?
expected result: ?
actual result: ?

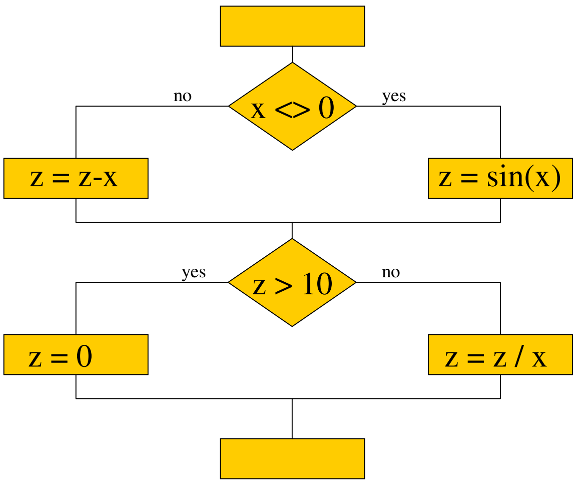
Condition - Branch Coverage

```
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 Coverage

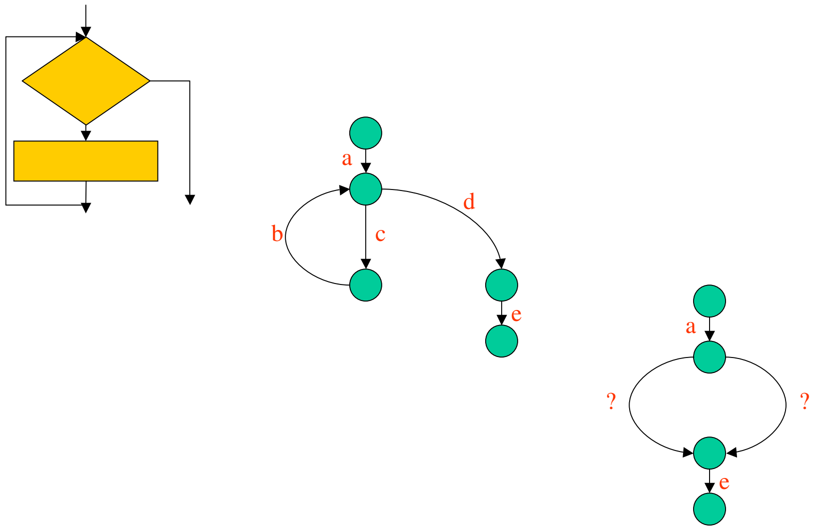


(n, y) x = ?, z = ?
(y, n) x = ?, z = ?

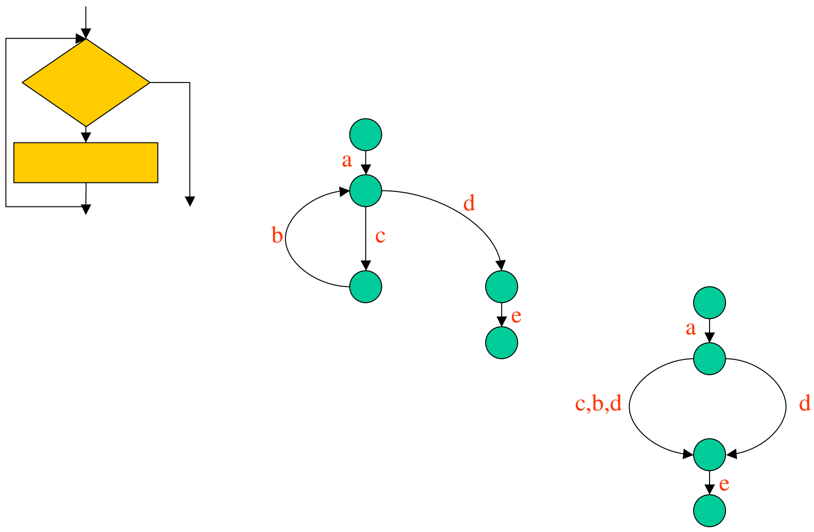
↓

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

Path with loops



Path with loops



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

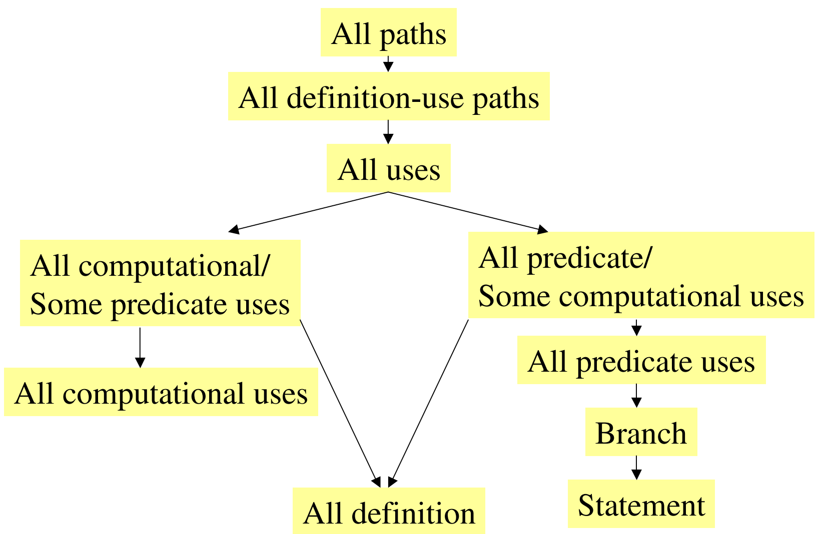
```
s = 0;
i = 1;
while (i <= n)
    {
        s + = i;
        i ++
    }
print (s);
print (i);
print (n);
```

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

        i ++
    }

print (i);
```


Relative strengths of test strategies (B. Beizer 1990)



Outline

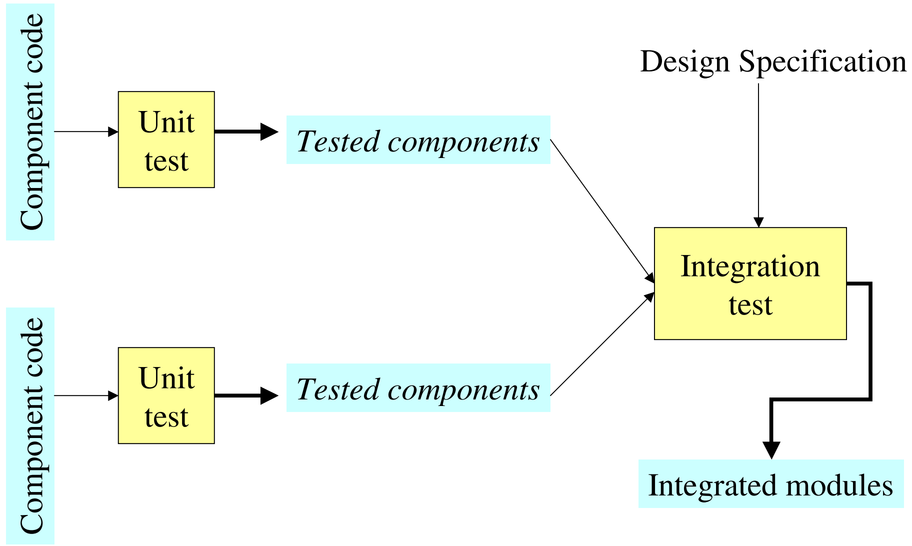
- Function testing
- Performance testing
- Acceptance testing
- Installation testing

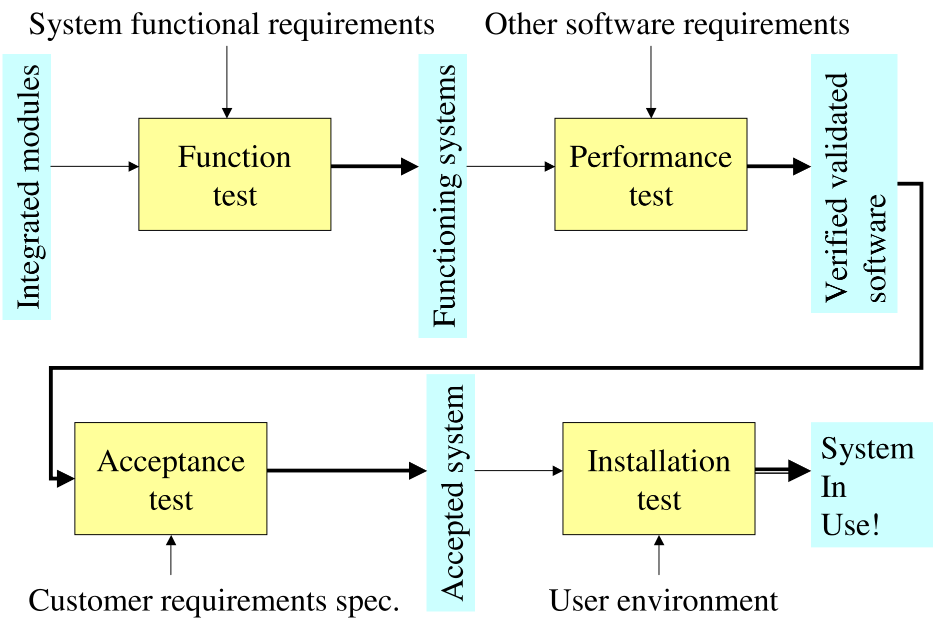
Objective: to ensure that the system does what the customer wants it to do.

Customer Developer

Requirements definition
Requirements specification

Functional requirements
Nonfunctional requirements


Component code
Unit test
Tested components
Design Specification
Integration test
Component code
Unit test
Tested components
Integrated modules


System functional requirements
Other software requirements
Integrated modules
Function test
Functioning systems
Performance test
Verified validated software
Acceptance test
Accepted system
Installation test
System In Use!
Customer requirements spec.
User environment

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

C_1: command is credit

C_2: command is debit

C_3: account number is valid

C_4: transaction amount is valid

Effects

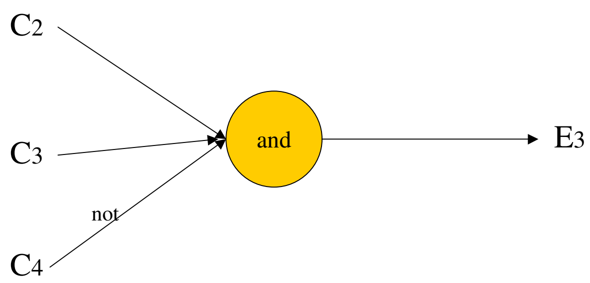
E_1: print "invalid command"

E_2: print "invalid account number"

E_3: print "debit amount not valid "

E_4: debit account print

E_5: credit account print



Performance testing
nonfunctional requirements

- Security
- Accuracy
- Speed
- Recovery
- Stress test
- Volume test
- …

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
- 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

Oracle

Scaffolding

What can be automated?

Test case generation

Termination

The Distribution of Faults in a Large Industrial Software System

Thomas J. Ostrand, Elaine J. Weyuker
AT&T Labs – Research

ACM SIGSOFT

2002 International Symposium on Software Testing and Analysis (ISSTA 02)

Empirical studies:

- difficult to locate and gain access to large systems.
- very time consuming, and therefore expensive, to collect and analyze the necessary data.
- difficult to find personnel with the appropriate skills to perform the empirical studies.

Questions

- How faults are distributed over the different files
  - Between release
  - Lifecycle stage
  - Severity
- How the size of modules affected their fault density
- Whether files that contained large numbers of faults during early stages of development, also had larger numbers of faults during later stages, and whether faultiness persisted from release to release.
- Whether newly written files were more fault-prone than ones that were written for earlier releases of the product.

Goal: identify characteristics of files that can be used as predictors of fault-proneness, thereby helping organizations determine how best to use their testing resources.

System description

- 13 successive releases
- Fault data was collected during:
  - Requirements
  - Design
  - Development
  - Unit testing
  - Integration testing
  - System testing
  - Beta release
  - Limited release:
    - Controlled release
    - General release
- Current version: 1,974 files, 500,000 lines of code, most of the system written in Java (1,412 files)

Distribution of Faults

			Early-Pre- Release		Late-Pre-Release		Post-Release		
Release	Files	KLOC	Dev	Unit	Int	Sys	Limited	Gen'l	Total
1	584	146	7	763	2	218	0	0	990
2	567	154	2	171	3	24	0	1	201
3	706	191	15	387	0	85	0	0	487
4	743	203	0	293	0	31	0	4	328
5	804	232	2	282	13	30	2	11	340
6	867	254	1	287	5	33	0	13	339
7	993	292	14	156	7	12	1	17	207
8	1197	339	14	363	28	77	2	6	490
9	1321	377	50	298	28	50	2	8	436
10	1372	396	84	119	7	24	1	11	246
11	1607	427	17	158	17	71	4	14	281
12	1740	476	62	130	8	53	1	19	273
13	1772	471	35	52	1	26	2	9	125
Total			303	3459	119	734	15	113	4743

Classification of 4,743 faults

- *Severity-1*: 78 faults (1,6%)
- *Severity-2*: 687 faults (14.5%)
- *Severity-3*: 3,847 faults (81%)
- *Severity-4*: 131 faults (2,8%)

Fault concentration by release

- For each release, the faults were heavily concentrated in a relatively small number of files.
- For all of the releases, the percentage of the code mass contained in the files containing faults exceeded the percentage of the files.

Overall Pareto Distribution by Release

	10% Files Contain		**100% Faults Contained In**	
Release	% Faults	% LOC	% Files	% LOC
1	68	35	40	72
2	85	33	16	36
3	83	33	20	43
4	88	37	15	42
5	85	41	16	46
6	92	33	13	34
7	97	32	11	32
8	94	32	12	38
9	96	30	11	31
10	100	-	8	26
11	100	-	7	26
12	100	-	7	24
13	100	-	4	13

Distribution of Faults by Lifecycle Stage

	Early-Pre- Release		Late-Pre-Release		Post-Release	
Release	% Files	% LOC	% Files	% LOC	% Files	% LOC
1	36	67	18	43	0	0
2	15	35	3	11	0	3
3	18	44	7	22	0	0
4	14	39	3	12	1	4
5	12	39	4	17	1	4
6	12	32	3	16	2	7
7	9	28	2	6	1	6
8	11	34	4	19	1	1
9	9	27	4	12	1	5
10	7	22	2	9	1	3
11	5	18	3	16	1	4
12	6	20	2	11	1	5
13	3	10	1	7	1	4

Fault Concentration by Severity

- *Severity-1*: 78 faults (1,6%) -- in 3% files in release-1 to 0% in release 12
- *Severity-4*: 131 faults (2,8%) -- in 4% files in release-1 to 0,3% in release 12
- *Severity-2*: 687 faults (14.5%) -- in small percentage of files
- *Severity-3*: 3,847 faults (81%)

	Severity-3 (80% faults)		**100% Faults Contained In**	
Release	% Files	% LOC	% Files	% LOC
1	36	68	40	72
8	11	35	12	38
12	6	22	7	24

Effects of Module size on Fault-proneness

- Standard wisdom: large modules – much more fault-prone
- Basili (1984), Moller (1993): contrary, opsite was true.
- In the study here
  - Fault densities between 10 and 75 faults/KLOC for smallest files (under 100 lines)
  - 2-3 faults/KLOC for larger than 1000 lines
- Hatton (1997): fault density was high for smallest components, decreased to minimum for medium-size components, and then started increasing again as components size grew.
- Fenton (2000): don't agree with any of the above, found no trend at all
- Various other factors: new file, amount of changed code, amount of testing performed, experience of the programmer…

=> needs more investigation

Persistence of High-Fault Files

Release	1	2	3	4	5	6	7	8	9	10	11	12
Rel (n-1)	-	27	54	21	45	42	52	34	21	17	39	24
Rel (n+1)	63	46	27	30	56	34	34	27	22	36	22	-

• High-Fault Files: top 20% of files ordered by decreasing number of faults, plus all other files that have as many faults as the least number among the top 20%

• Rel (n-1): shows the percent of high-fault files in Release (n-1) that remained high-fault files in Release n.

•Rel (n+1): shows the percent of high-fault files in Release (n+1) that had been high-fault files in Release n.

=> file containing high numbers of faults in one release, remain high-fault files in later release

Comparison of Faults For Old and New Files

	%Faulty Files		**Fault /KLOC**	
Release	**OLD**	**NEW**	**OLD**	**NEW**
2	15.4	16.3	1.29	1.46
3	18.5	24.8	2.32	4.01
4	13.8	39.1	1.42	5.44
5	13.8	31.0	1.33	2.09
6	11.0	36.8	1.13	4.90
7	10.2	13.3	.69	.90
8	12.2	12.8	1.36	1.97
9	8.9	36.3	.81	4.85
10	7.6	20.7	.60	1.15
11	6.9	7.9	.60	1.54
12	5.8	14.4	.49	1.08

- Percentage of faulty new files is larger than the percentage of faulty pre-existing files
- The fault density is higher for new files than for pre-existing ones

=> More resources for testing new files

Conclusions

- Fault concentrate in small numbers of files and small percentages of the code mass.
- For each release, the early-pre-release faults accounted for a clearly majority of the faults.
- Percentage of lines of code contained in files that contained faults exceeded the percentage of files that contained faults.
- Across successive releases, high-fault files of one release tend to remain high-fault in later release.

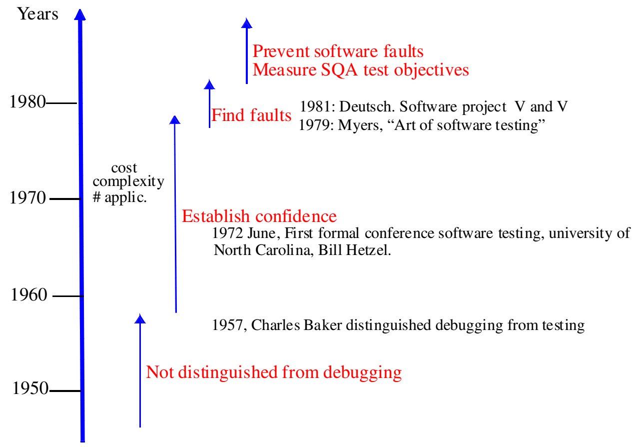
Real life examples

- 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).

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".

Goals of software testing: Historical Evolution



And …

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

E. Dijkstra, 1969