Software Testing

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

C.S. Lewis, 1934

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

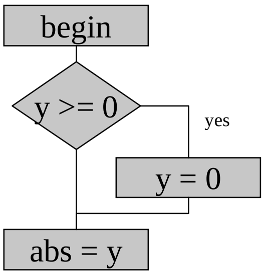
- Australia: Man jailed because of computer glitch. He was jailed for traffic fine although he had actually paid it for 5 years ago.
- Dallas Prisoner released due to program design flaw: He was temporary transferred from one prison to another (witness). Computer gave him “temporary assignment”.

Why Testing and Analysis?

- Software is never correct no matter which developing technique is used
- Any software must be verified.
- Software testing and analysis are
  - important to control the quality of the product (and of the process)
  - very (often too) expensive
  - difficult and stimulating

Statement Coverage

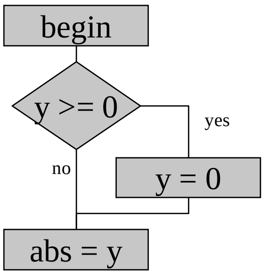
```
Begin
if ( y >=  0)
        then y = 0;
abs = y;
end;
```



test case-1:
input: y = 0
expected result: 0
actual result: ?

Branch Coverage

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



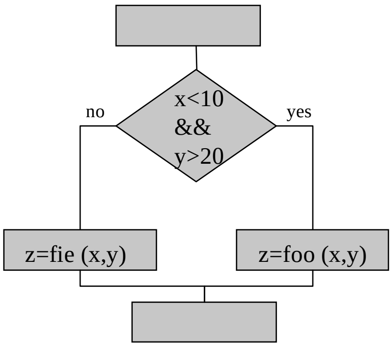
test case-1:
input: y = 0
expected result: 0
actual result: ?

test case-2:
input: y = -5
expected result: 5
actual result: ?

```
Begin

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

z = foo (x,y); else z =fie (x,y);
}
end;
```

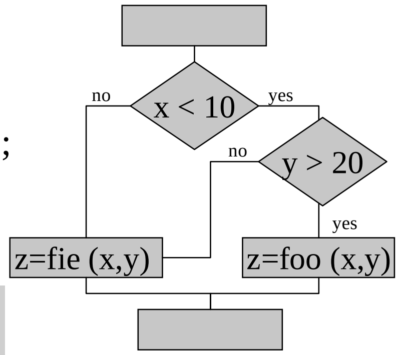


test case-1:
input: x = -4, y = 30
expected result: ...
actual result: ?

test case-2:
input: x = 12, y = 12
expected result: ...
actual result: ?

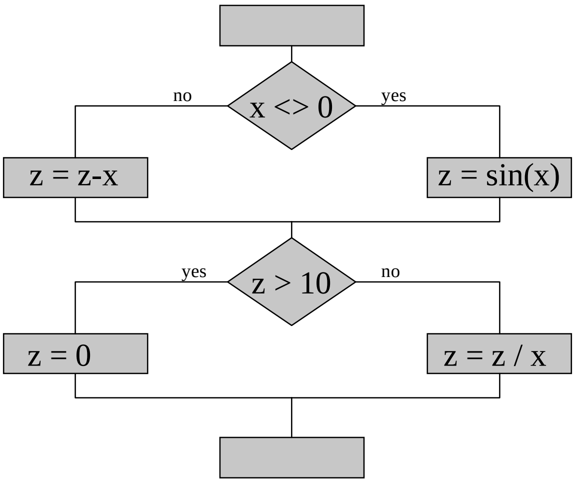
Condition / Branch Coverage

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



	x<10	y>20
test-case-1:	t	t
test-case:2	t	f
test-case-3:	f	t
test-case-4	f	f

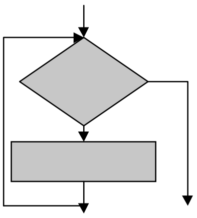
Path Coverage

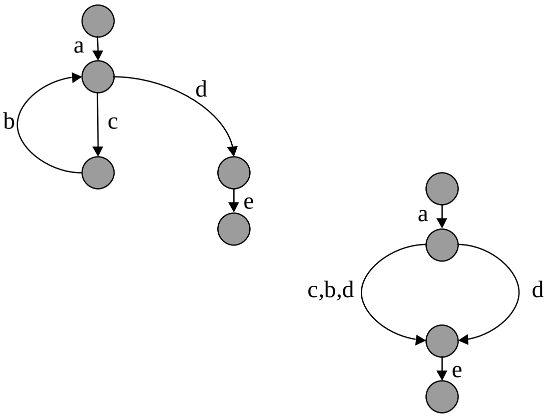


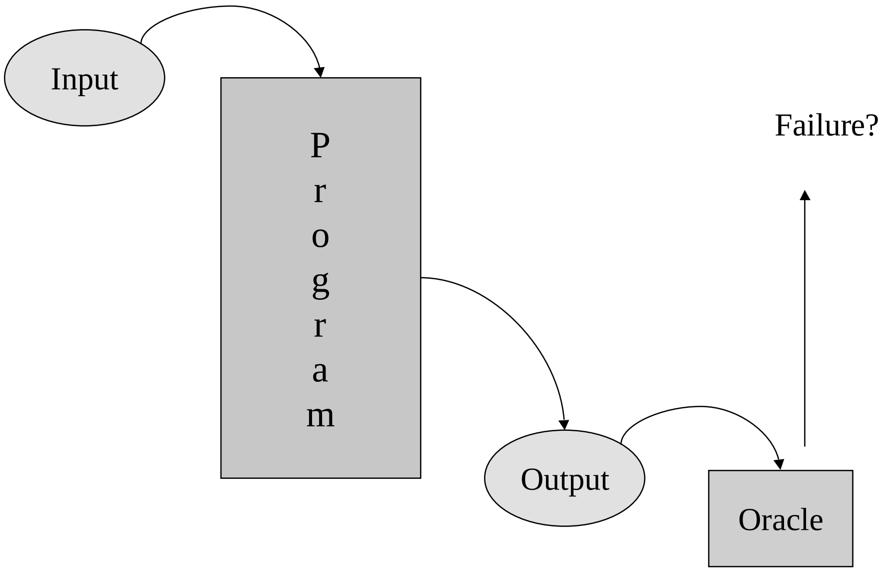
x = 0, z = 12
x = 2, z = 6

x = 0, z = 7
x = 0, z = 13
x = 1, z = 5
x = 2, z = 15

Path with loops







Input

Program

Failure?

Output

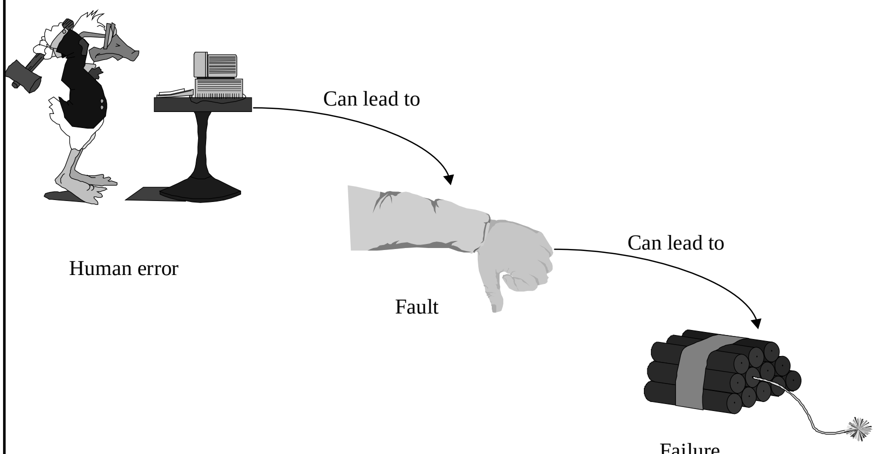
Oracle

Glass box testing!
White box testing!
Structural testing!

Glass box testing

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

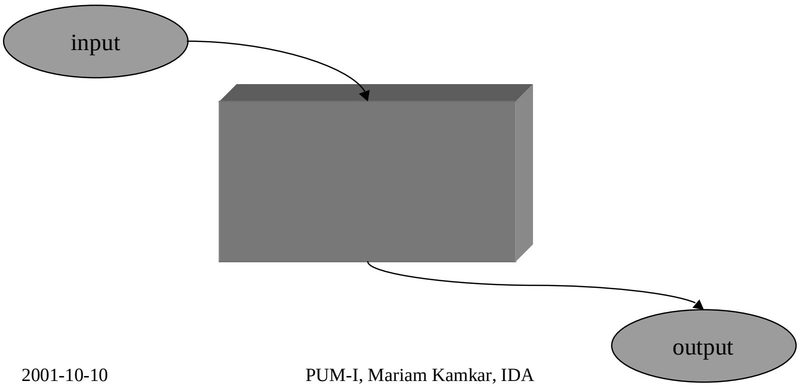
Error, Fault, Failure



Black box testing!
Functional testing!

Black box testing

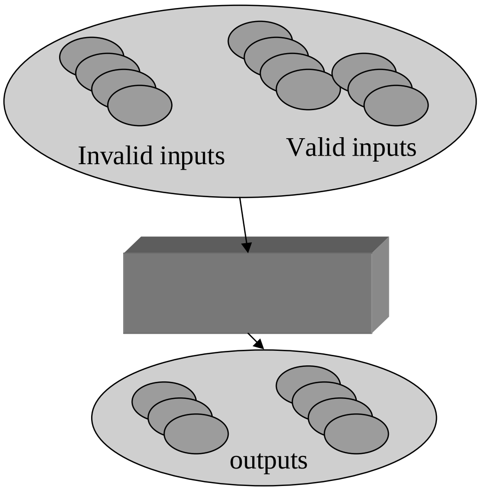
- incorrect or missing functions
- interface errors
- performance error



Black box testing

- Equivalence partitioning
- Boundary value analysis
- Decision Table
- Cause-Effect
- Exhaustive testing

Equivalence partitioning



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

input values

Less than 10000	Between 10000 and 99999	More than 99999

Number of input values

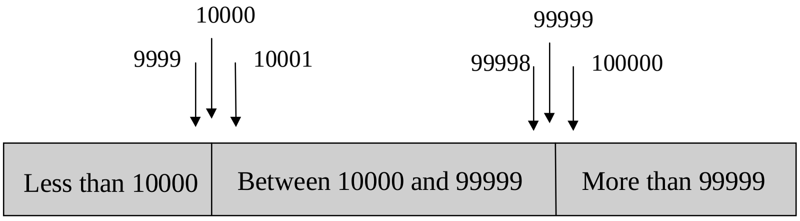
Less than 4	Between 4 and 8	More than 8

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



Cause-Effect

Causes

C$_1$: command is credeit

C$_2$: command is debit

C$_3$: account number is valid

C$_4$: transaction amount is valid

Effects

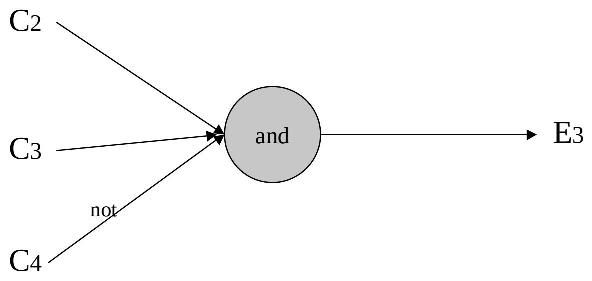
E$_1$: print “invalid command”

E$_2$: print “invalid account number”

E$_3$: print “debit amount not valid ”

E$_4$: debit account print

E$_5$: credit account print



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}

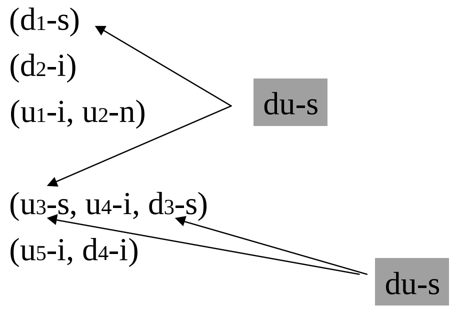
S_1: i = 1; DEFS (S_1) = {i} (d_1-i)

S_2: while (i <= n) USE(S_2) = {i, n} (u_1-i, u_2-n)

definition-use chain (du chain) = [x, S, S’]

du-1: [i, S_1, S_2] (d_1-i), (u_1-i)

Data Flow testing



```
s = 0;                (d1-s)
i = 1;                (d2-i)
while (i <= n)        (u1-i, u2-n)
   {
        s + = i;      (u3-s, u4-i, d3-s)
        i ++          (u5-i, d4-i)
   }
print (s);            (u6-s)
print (i);            (u7-i)
print (n);            (u8-n)
```

Data Flow Testing

```
s = 0;                (d1-s)
i = 1;                (d2-i)
while (i <= n)        (u1-i, u2-n)
   {
        s + = i;      (u3-s, u4-i, d3-s)
        i ++          (u5-i, d4-i)
   }
print (s);            (u6-s)
print (i);            (u7-i)
print (n);            (u8-n)
```

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

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);
```

Static Software Testing

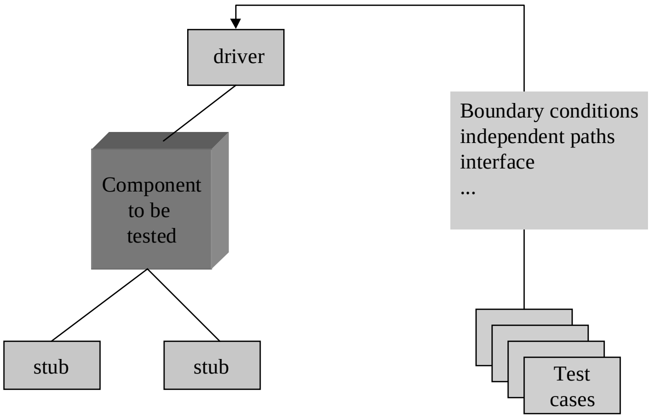
- Inspection (design, code)
  - – overview
  - – preparation
  - – inspection
  - – rework
  - – follow-up

- Walkthroughs (design, code, chapter of user’s guide,…)
  - presenter
  - coordinator
  - secretary
  - maintenance oracle
  - standards bearer
  - user representative

Levels of Software Testing

- Component
- Integration
- System
- Acceptance

Components



driver

Component to be tested

stub

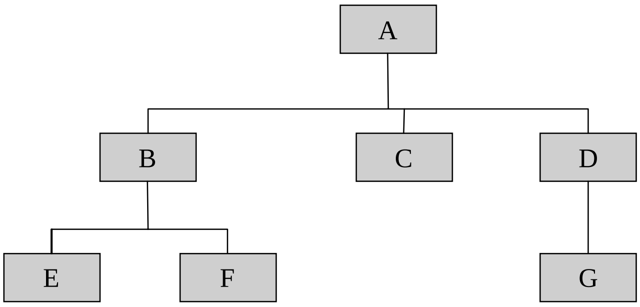
stub

Boundary conditions
independent paths
interface
...

Test cases

Classes of Integration Testing

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



A

B

C

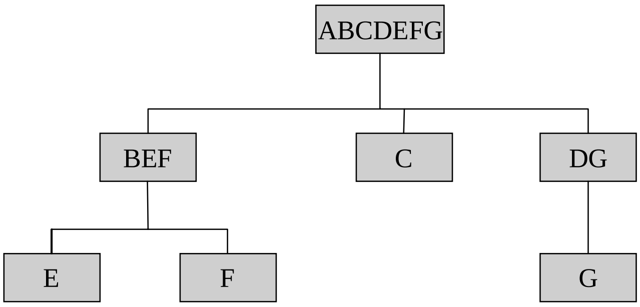
D

E

F

G

Bottom-up



ABCDEFG

BEF

C

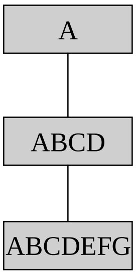
DG

E

F

G

Top-down



System Testing

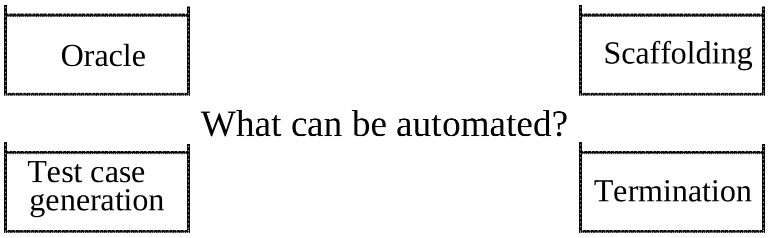
- Recovery testing (fault tolerant)
- Security testing
- Stress testing (volume, resources,…)
- Performance testing (real-time, embedded system)

Acceptance Testing

- Alpha test: at the developer’s site, controlled environment
- Beta test: at one or more customer site.

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

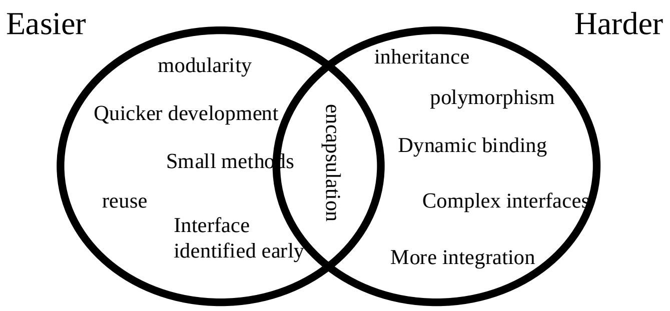
Scaffolding

What can be automated?

Test case generation

Termination

Testing object-oriented systems



Easier

Harder

modularity

Quicker development

Small methods

reuse

Interface identified early

encapsulation

inheritance

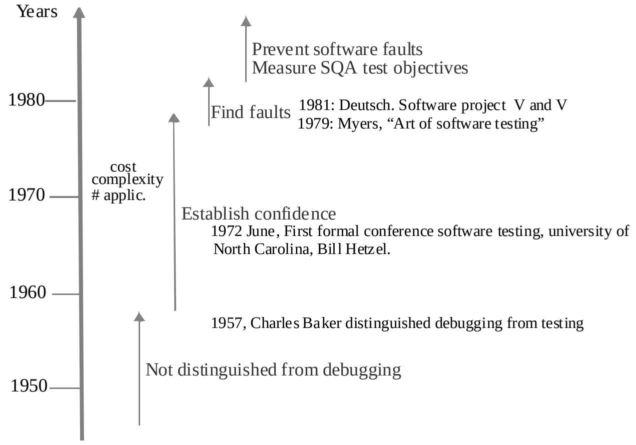
polymorphism

Dynamic binding

Complex interfaces

More integration

Goals of software testing: Historical Evolution



Years

1980

1970

1960

1950

cost
complexity
# applic.

Prevent software faults
Measure SQA test objectives

Find faults

1981: Deutsch. Software project V and V
1979: Myers, “Art of software testing”

Establish confidence

1972 June, First formal conference software testing, university of North Carolina, Bill Hetzel.

1957, Charles Baker distinguished debugging from testing

Not distinguished from debugging

And …

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

E. Dijkstra, 1969