Software Testing (1)

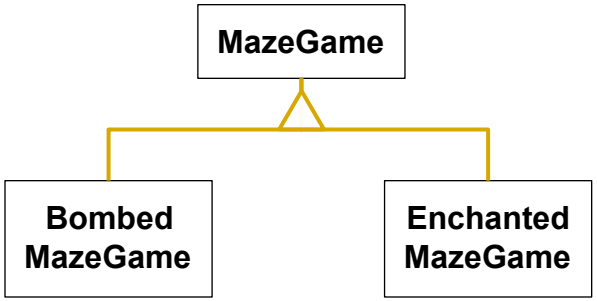
CS580 Advanced Software Engineering
http://cs580.yusun.io
November 3, 2014

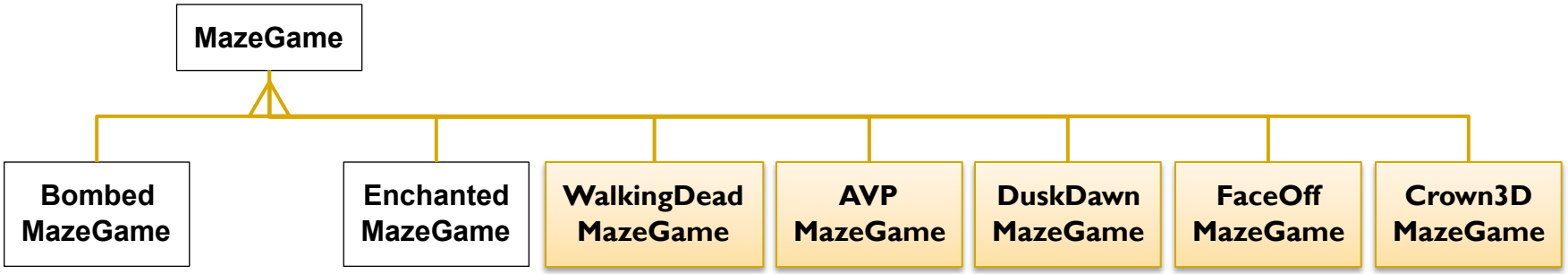
Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Horror Nights

Design Pattern?





Suggestion

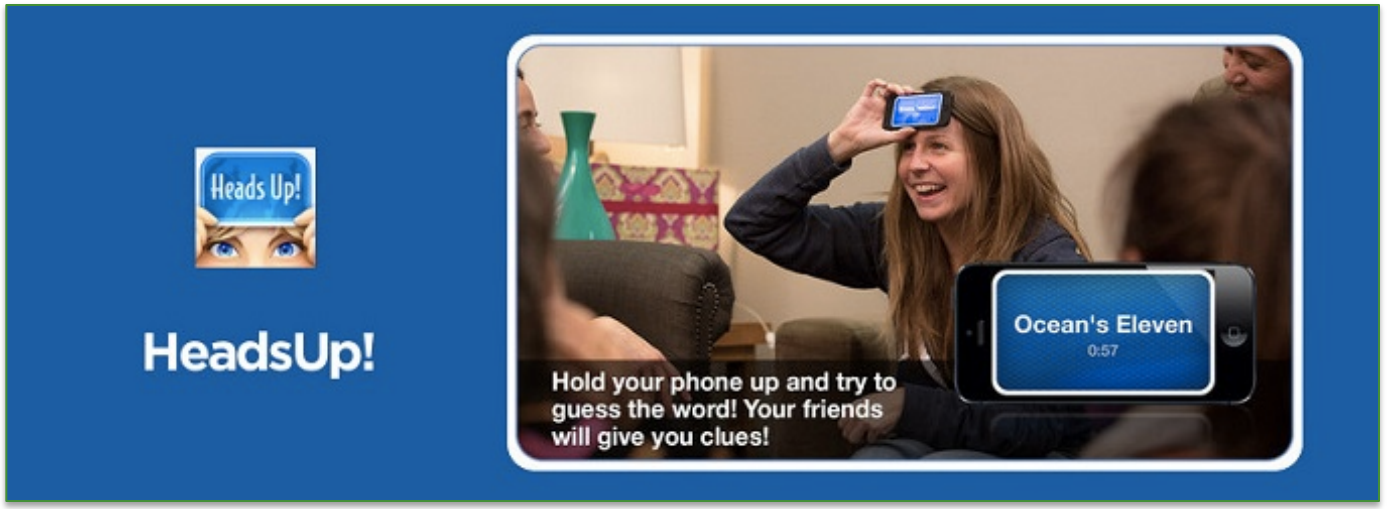

SEPTEMBER / OCTOBER / NOVEMBER
= KILLER DEAL NIGHTS
SU M T W TH F SA
14 15 16 17 18 19 20 Sold Out
21 SEPTEMBER 22 23 24 25 26 Sold Out 27 Sold Out
28 29 30 1 OCTOBER 2 3 4 Sold Out
5 6 7 8 9 10 Sold Out 11 Sold Out
12 13 14 15 16 Sold Out 17 18 Sold Out
19 20 21 22 23 Sold Out 24 25 Sold Out
26 27 28 29 30 Sold Out 31 Sold Out 1 NOVEMBER Sold Out
Thanks for a great 2014!
See you in 2015!
FRONT OF LINE PASS
Cut to the front one time at each maze, ride and Terror Tram
VIP EXPERIENCE
Unlimited Front of Line to all mazes and rides, exclusive VIP Horror Lounge serving dinner & drinks, VIP guide escort to the backlot mazes, and valet parking.

Problem


GRINGOTTS
WAIT TIME
450
MINUTES
WARNING!
© Twitter.com/InsideTheMagic


STUDIO DIRECTORY
LAST TERROR TRAM 11:45PM
NEXT PERFORMANCE
LIVE SHOWS
RIDES & ATTRACTIONS
WAIT TIME
120 min
150 min
70 min
75 min
30 min
30 min
45 min
TERROR TRAM
TRANSFORMERS
JURASSIC PARK IN THE DARK
REVENGE OF THE MUMMY
THE SIMPSONS RIDE
DESPICABLE ME
CHECK WAIT TIMES
HHNWaitTimes.com

Solution 1: Kill Time in the Line



- Senior Project:
  - Apps to help kill the time
  - Context-aware
  - Group involvement
  - Social
  - Move and exercise

Solution 2: Be Smart on Wait Times

- Not very useful inside the park
- Historical data might be more useful
- Senior Project:
  - Crawling and saving the data
  - Wait Time Prediction
  - Trip planner

Software Testing (1)

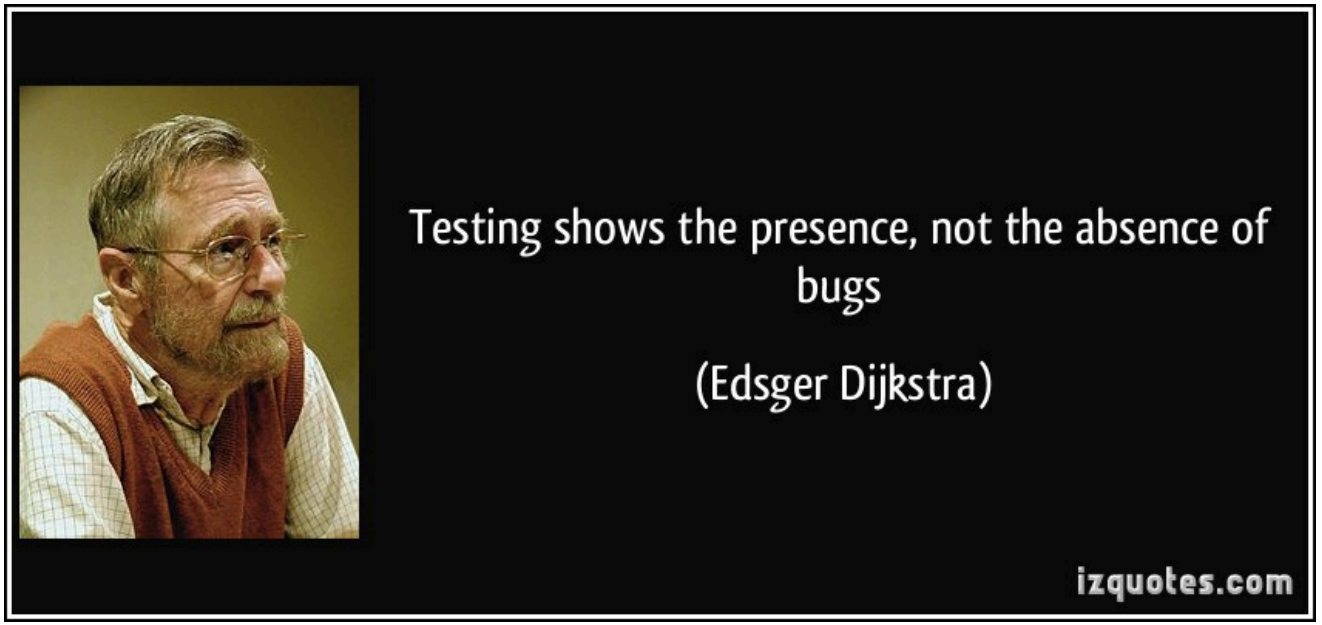
CS580 Advanced Software Engineering
http://cs580.yusun.io
November 3, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

CAL POLY POMONA

Program Testing

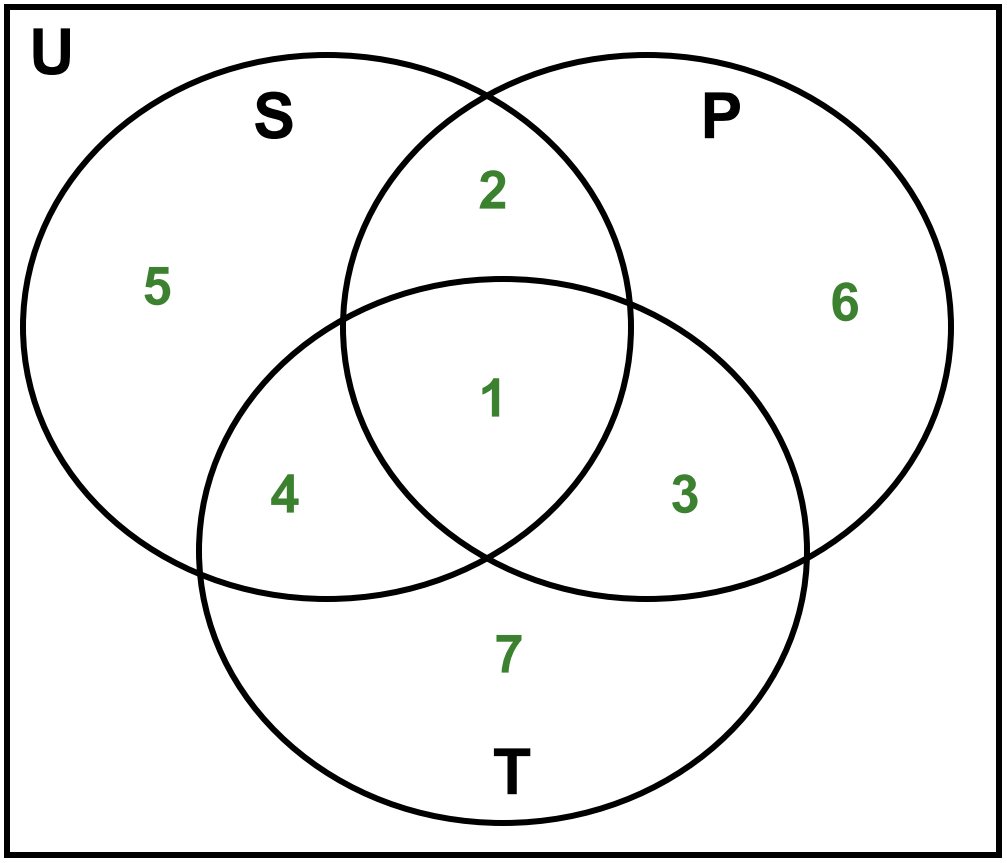
- Can reveal the presence of errors NOT their absence
  - Only exhaustive testing can show a program is free from defects
  - Exhaustive testing for anything but trivial programs is impossible



Program Testing

- Can reveal the presence of errors NOT their absence
  - Only exhaustive testing can show a program is free from defects
  - Exhaustive testing for anything but trivial programs is impossible
- A successful test discovers one or more errors
- Run all tests after modifying a system
  - Regression testing

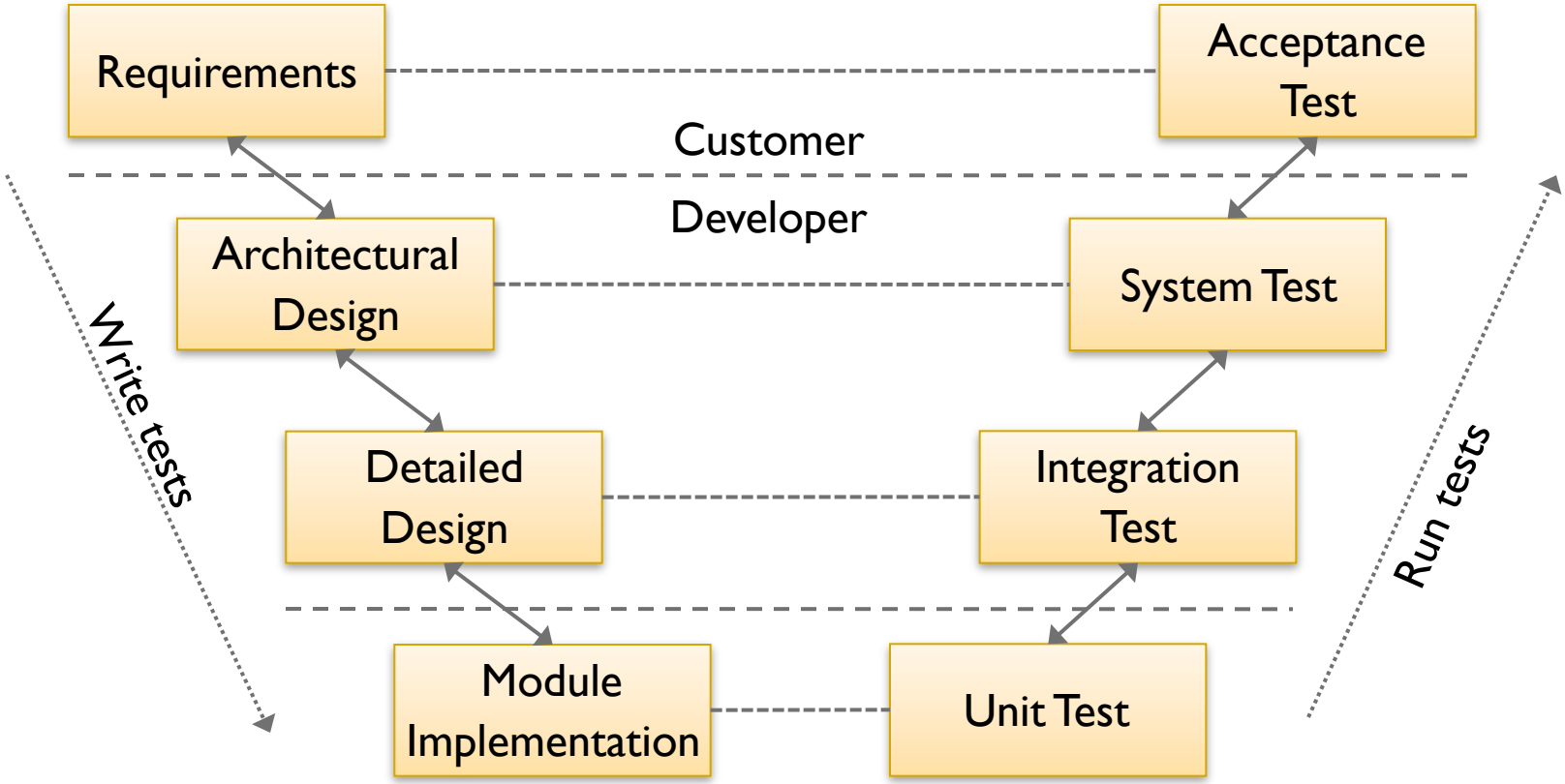
Specified, Programmed, and Tested Behaviors



S = Specified behaviors
P = Programmed behaviors
T = Tested behavior
U = All possible behaviors

We want to make region 1 as large as possible

The V-model of Testing



Testing Stages

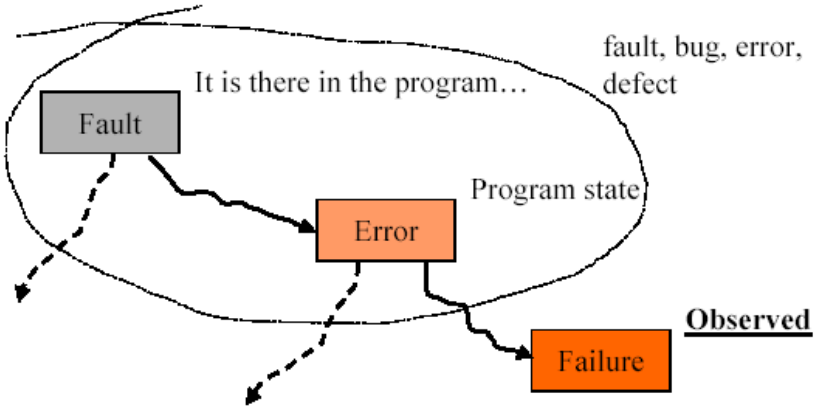
- Unit testing
  - Testing of individual components
- Integration testing
  - Testing to expose problems arising from the combination of components
- System testing
  - Testing the complete system prior to delivery
- Acceptance testing
  - Testing by users to check that the system satisfies requirements

Testing Stages

- Alpha testing
  - When a product is used by many users, an alpha test is conducted in a controlled environment at the development site with end-user participation
- Beta testing
  - An extension to alpha testing where the users test the software in a "live" environment; developers are typically not present

Terminology

- Failure
  - A failure is said to occur whenever the external behavior does not conform to system spec
- Error
  - An error is a state of the system which, in the absence of any corrective action, could lead to a failure
- Fault
  - The actual cause of an error



Distinction Between Debugging and Testing

- Defect testing and debugging are distinct processes
- Defect testing is concerned with confirming the presence of errors
- Debugging is concerned with locating and repairing these errors
- Debugging involves formulating a hypothesis about program behavior then exploring these hypotheses to find the system error

Historical Views: Thinking About Testing

- Phase 0
  - Testing = Debugging
- Phase 1
  - Testing is an act whose purpose is to show that the software works
- Phase 2
  - Testing is an act whose purpose is to show that the software does not work

Historical Views: Thinking About Testing

- Phase 3
  - Testing is an act whose purpose is not to prove anything, but to reduce the perceived risk of failure to an acceptable level

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

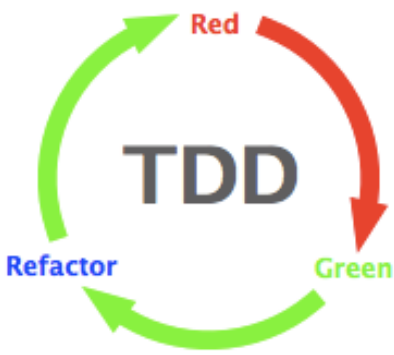
All Prime Numbers (1-100)

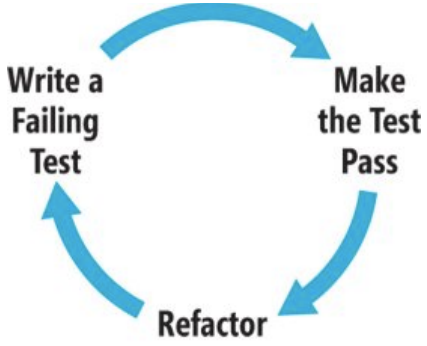
Availability	Downtime Per Year (24X7X365)
99.000%	3 Days 15 Hours 36 Minutes
99.500%	1 Day 8 Hours 48 Minutes
99.900%	8 Hours 46 Minutes
99.950%	4 Hours 23 Minutes
99.990%	53 Minutes
99.999%	5 Minutes
99.9999%	30 Seconds

Service Level Agreement (SLA)

Historical Views: Thinking About Testing

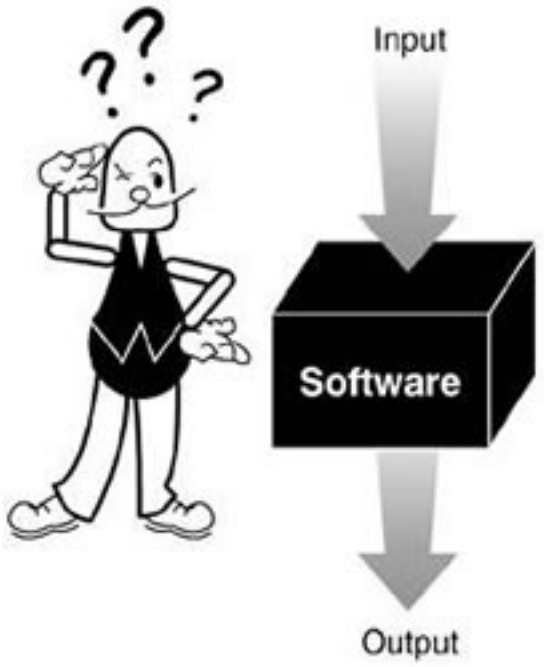
- Phase 3
  - Testing is an act whose purpose is not to prove anything, but to reduce the perceived risk of failure to an acceptable level
- Phase 4
  - Testing is not an act; rather, it is a mindset that involves development and coding practices along with a systematic approach to exercising the software





Testing Methods

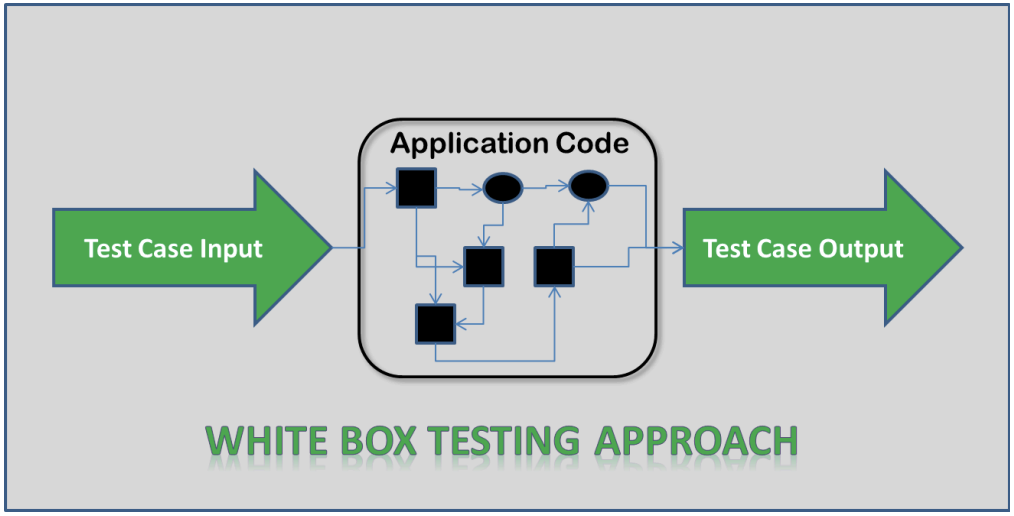
- Functional (Black Box) Testing
  - Knowing the specified functions that a product has been designed to perform, tests can be conducted to demonstrate that each function is fully operational
  - Test cases are based on external behavior
  - Aka: specification-based, data-driven, or input/output driven testing



Black-Box Testing

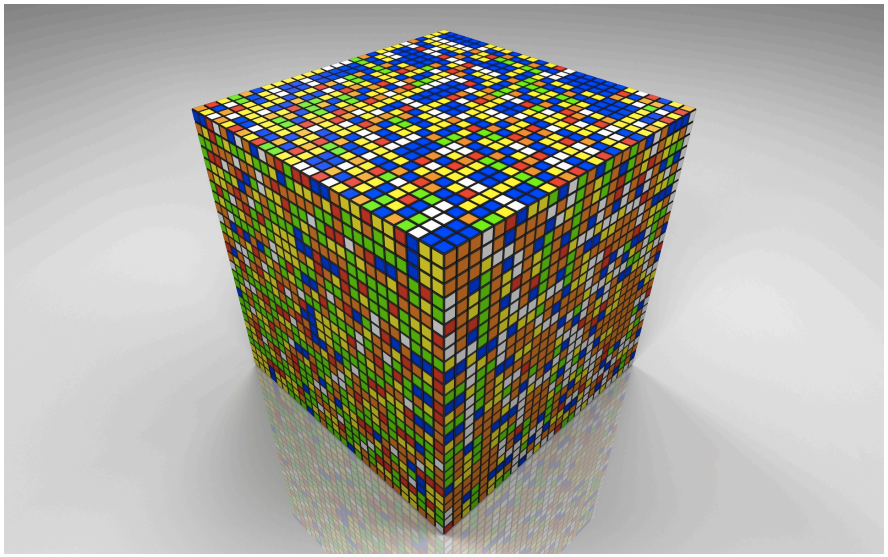
Testing Methods

- Structural (White Box) Testing
  - Knowing the internal workings of a program, tests can be conducted to assure that the internal operation performs according to specification, and all internal components have been exercised
  - Test cases are based on internal structure of the program and a specific level of coverage.



Feasibility of Black-Box Testing

- Suppose specs include 20 factors, each taking on 4 values
  - 4^{20} or 1.1×10^{12} test cases
  - If each takes 30 seconds to run, running all test cases takes > 1 million years
- Combinatorial explosion makes exhaustive testing to specifications impossible



Feasibility of White-Box Testing

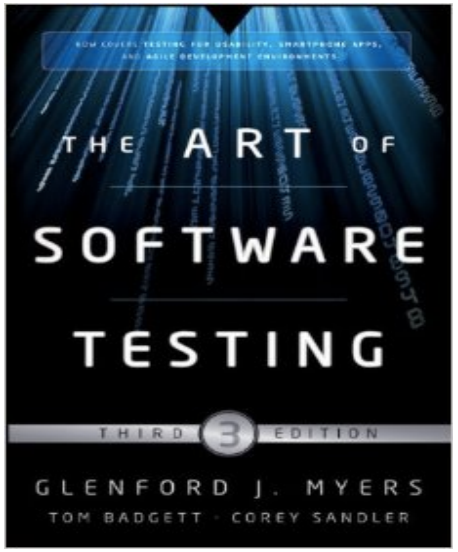
- Can exercise every path without detecting every fault (what if x=2, y=1, z=3?)

```
if ((x + y + z)/3 == x)
   print "x, y, z are equal in value";
else
   print "x, y, z are unequal";
```

Test case 1: $x = 1, y = 2, z = 3$
Test case 2: $x = y = z = 2$

Coping with the Combinatorial Explosion

- Neither testing to specifications nor testing to code is feasible toward ensuring complete correctness
- The art of testing
  - Select a small, manageable set of test cases to
    - Maximize chances of detecting fault, while
    - Minimizing chances of wasting test case
  - Every test case must detect a previously undetected fault



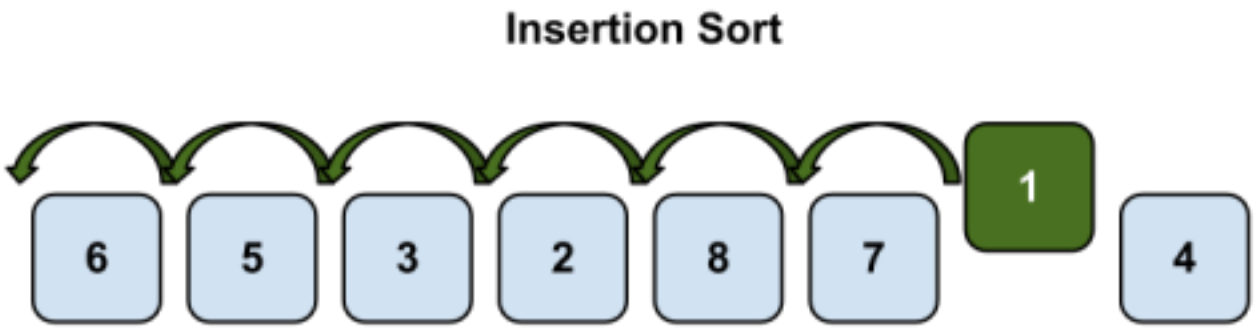
Coping with the Combinatorial Explosion

- We need a method that will highlight as many faults as possible
  - First black-box test cases (testing to specifications)
  - Then white-box methods (testing to code)

Functional Testing

Functional (Black Box) Testing

- Knowing the specified function (requirements), design test cases to ensure that those requirements are met
  - Example : Sort (list);
    - Functional Testing - How well does Sort perform its intended function?
    - Structural Testing - How well is the code exercised?
- In general, complete functional testing is not feasible
  - Attempting to test every possible input to the function



Functional (Black Box) Testing

- A randomly selected set of test cases is statistically insignificant
  - "Not all test cases are created equally"
- Test case selection
  - Based on characteristics of input and output sets relative to specified functionality

Goals and Methods of Functional Testing

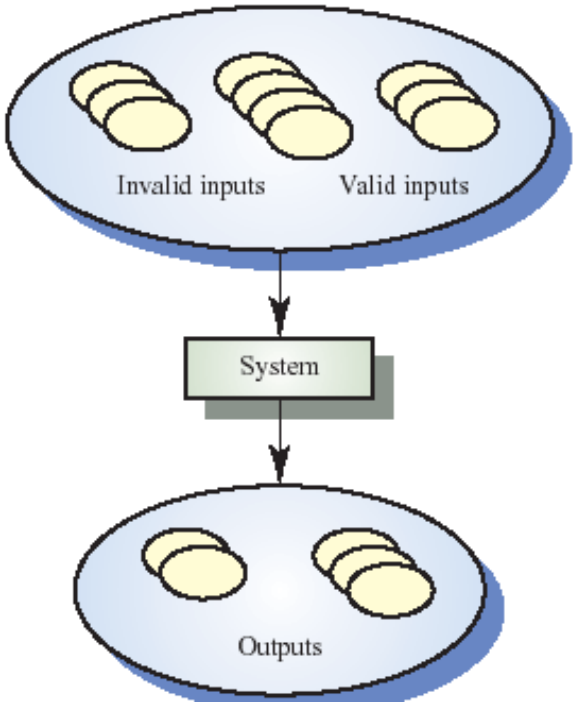
- Goals
  - Produce test cases that reduce the overall number of test cases
  - Generate test cases that will tell us something about the presence or absence of errors for an entire class of input
- Methods/Approaches
  - Equivalence partitioning
  - Boundary value analysis
  - Cause-effect graphing

Equivalence Partitioning

- It is impossible to test all cases
- Equivalence partitioning provides a systematic means for selecting cases that matter and ignoring those that don't
- An equivalence class or equivalence partition is a set of test cases that tests the same aspect or reveals the same bugs

e.g., **If** X ≥ 15 **then** do-this **else** do-that

(-∞ 15) 15 (15 ∞)



Equivalence Partitioning

- Equivalence partitions – groups for similar inputs, outputs, and/or operation of the software

e.g., file-name, 1 .. 255 characters
- valid characters
- invalid characters
- valid length
- invalid length

Equivalence Class Example

```
function in_list (input1 : name_type;
                  input_list : list_names)
        return boolean is ...
```

Equivalence Classes:

(1) Inputs where input1 is in the list

(2) Inputs where input1 is not in the list

Specific Input Partitions:

List	input1
Empty	?
One element	In the list
One element	Not in the list
>One element	First element
>One element	Last element
>One element	Middle element
>One element	Not in list

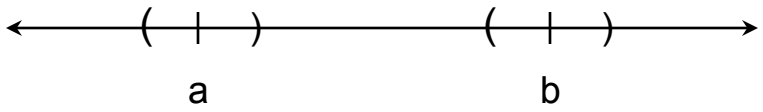
Test Cases

List	input1	Output
<nil>	?	false
bird	bird	true
bird	fish	false
bird, cat, owl	bird	true
dog, pig, chicken	chicken	true

...

Boundary Value Analysis

- Range : a..b



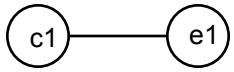
  - Example : 100..200
    - Test cases : 99, 100, 101, 199, 200, 201

- Number of values
  - Test cases that exercise minimums and maximums
- Apply the above to the output conditions
  - Try to drive output to invalid range
- Internal data structures with boundaries
  - Example : A(1..100) with test cases A(0), A(1), A(2), A(99), A(100), A(101)
  - A(0) and A(101) should generate exceptions

Cause Effect Graphing

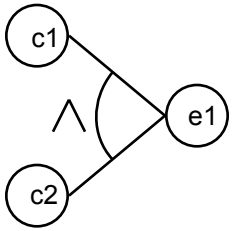
- Causes (input conditions) and effects (actions) are listed for a module, and an identifier is assigned to each
- A cause-effect graph is developed
  - Looking for causes without effects
  - Looking for effects without causes
- The graph is converted to a decision table (if a decision table has been used as a design tool, developing the graph and table is not necessary)
- Decision table rules are converted to test cases

Cause-Effect Graph Symbology

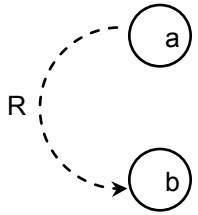


Identity

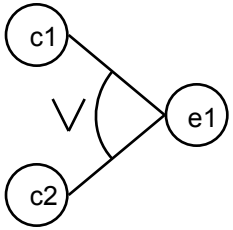
“Not”



“And”



Requires



“Or”

Cause-Effect Graphing Example

- The CHANGE subcommand - used to modify a character string in the "current line" of the file being edited
  - Inputs
    - Syntax : C /string1/string2
    - String1 represents the character string you wish to replace
      - 1-30 characters
      - Any character except '/'
    - String2 represents the character string that is to replace string1
      - 0-30 characters
      - Any character except '/'
    - At least one blank must follow the command name "C"

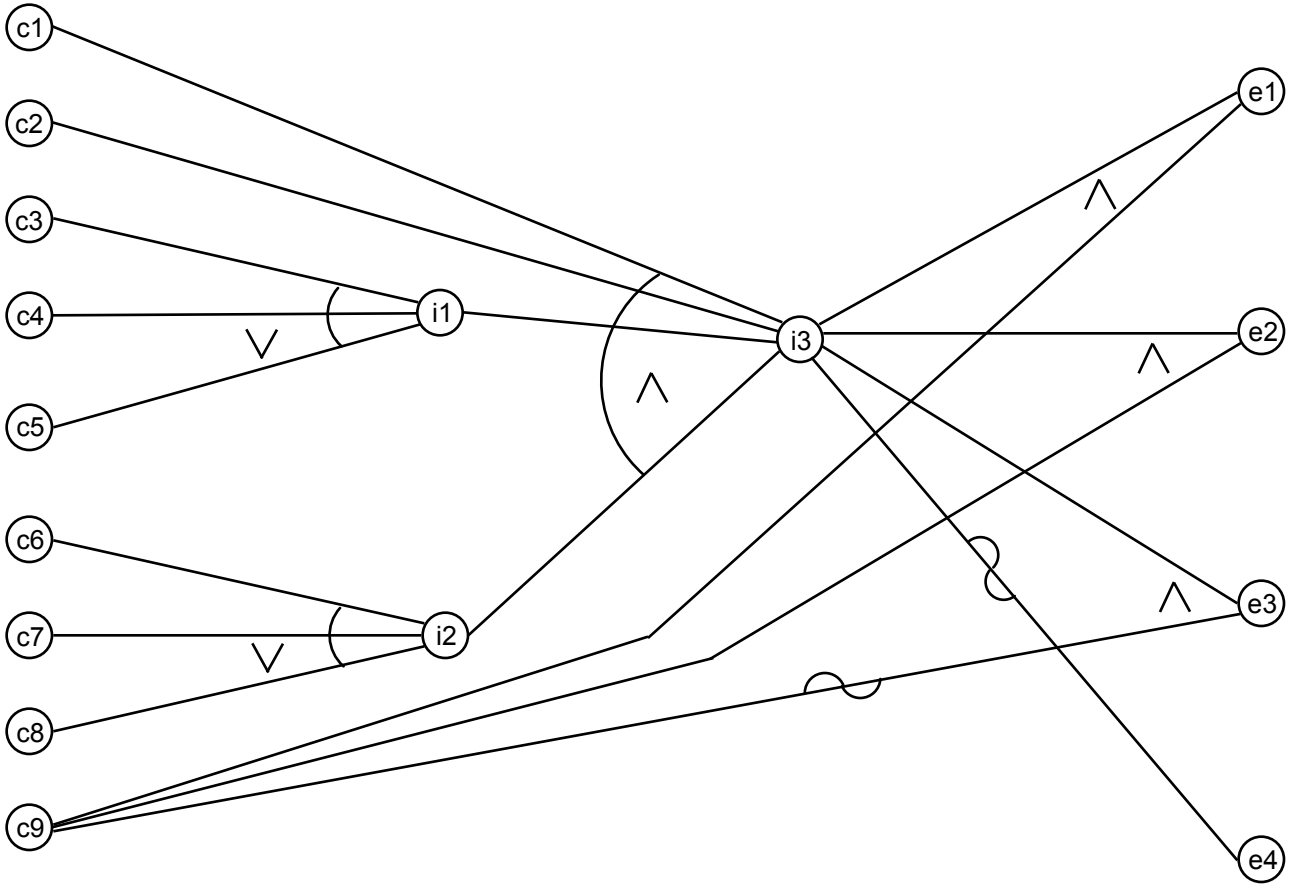
Cause-Effect Graphing Example

- Outputs
  - Changed line is printed to the terminal if the command is successful
  - "NOT FOUND" is printed if string1 cannot be found
  - "INVALID SYNTAX" is printed if the command syntax is incorrect
- System Transformations
  - If the syntax is valid and string1 can be found in the current line, then string1 is removed and string2 replaces it
  - If the syntax is invalid or string1 cannot be found, the line is not changed

Cause-Effect Graphing Example

- Cause 1: The first nonblank character following the "C" and one or more blanks is a '/'
- Cause 2: The command contains exactly two '/' characters
- Cause 3: String1 has length 1
- Cause 4: String1 has length 30
- Cause 5: String1 has length 2-29
- Cause 6: String2 has length 0
- Cause 7: String2 has length 30
- Cause 8: String2 has length 1-29
- Cause 9: The current line contains an occurrence of string1
- Effect 1: The changed line is typed
- Effect 2: The first occurrence of string1 in the current line is replaced by string2
- Effect 3: NOT FOUND is printed
- Effect 4: INVALID SYNTAX is printed

Example Cause-Effect Graph



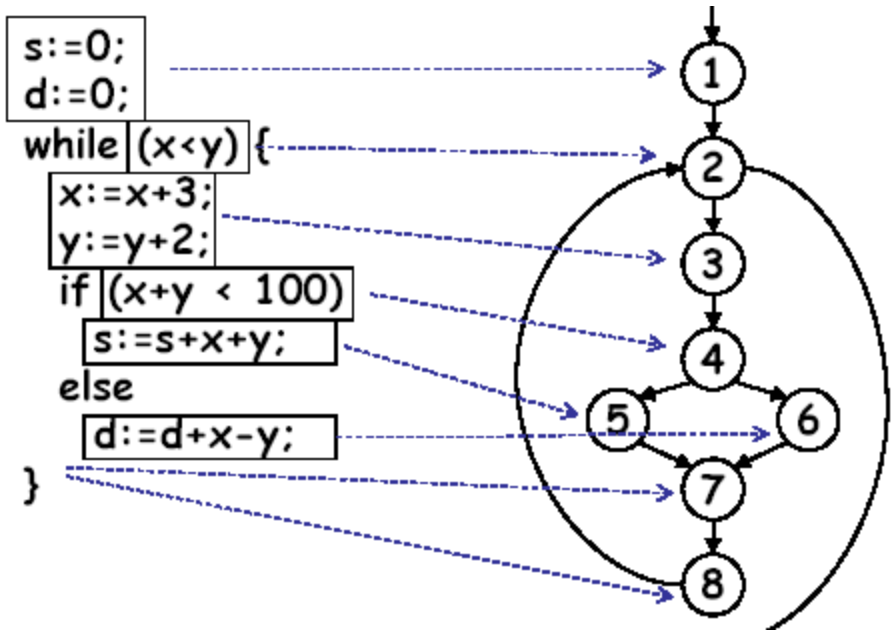
White Box Testing

Glass-Box Module Testing Methods

- Structural testing
  - Statement coverage
  - Branch coverage
  - Condition coverage
  - Path coverage
  - Linear code sequences (not covered)
  - All-definition-use path coverage

Control Flow Graph

- Step 1: From the source code, create a graph describing the flow of control (called the control flow graph). The graph is created (extracted from the source code) manually or automatically
- Step 2: Design test cases to cover certain elements of this graph nodes, edges, paths



Statement Coverage

- Statement coverage:
  - Series of test cases to check every statement at least once
- Weakness
  - Branch statements
- Consider ABS function

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

test case-1:

input: y = 0

expected result: 0

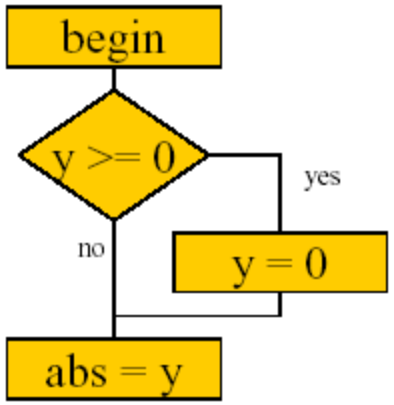
actual result: 0

This case gives the false assumption that the code is correct

Branch Coverage

- Series of tests to check all branches (solves problem on previous slide); edge coverage

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

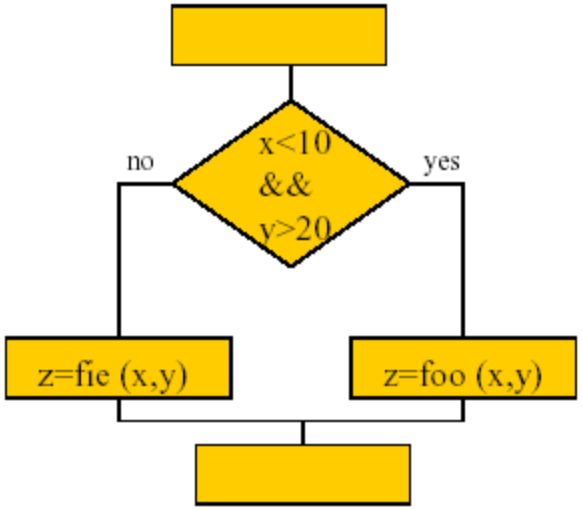


Branch=>statement not true other way…

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

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

Branch Coverage



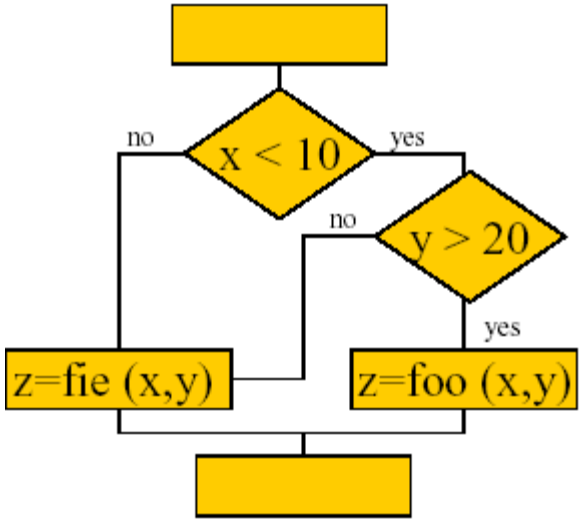

```
if ( x < 10 && y > 20) {
  z = foo (x,y);
else
  z = fie (x,y);
}
```

What is the *potential* problem here?

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

test case-2 (no-branch):
input: x = 12, y = 12
expected result: ...
actual result: ?

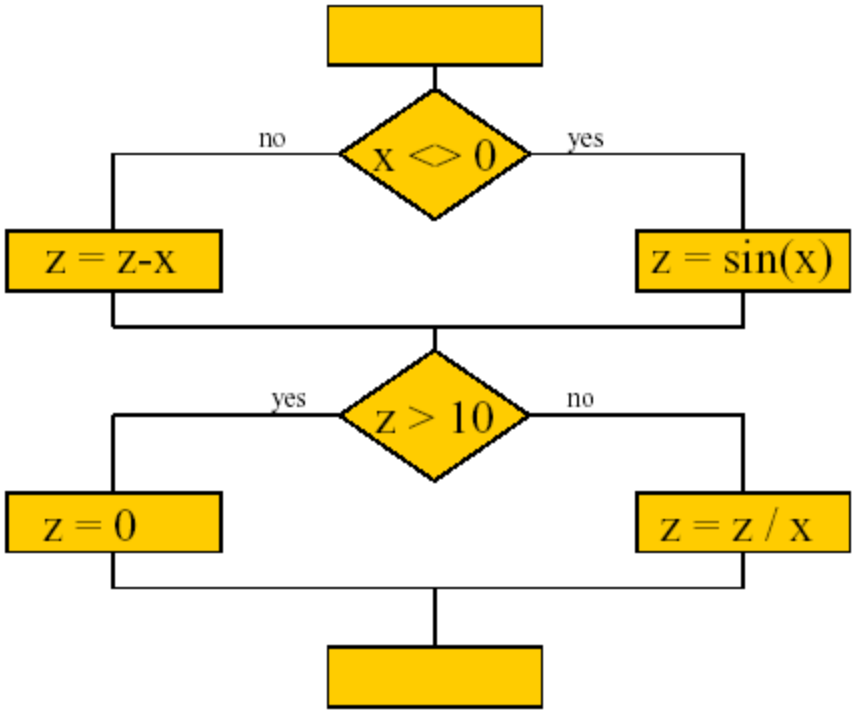
Condition Coverage




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

	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

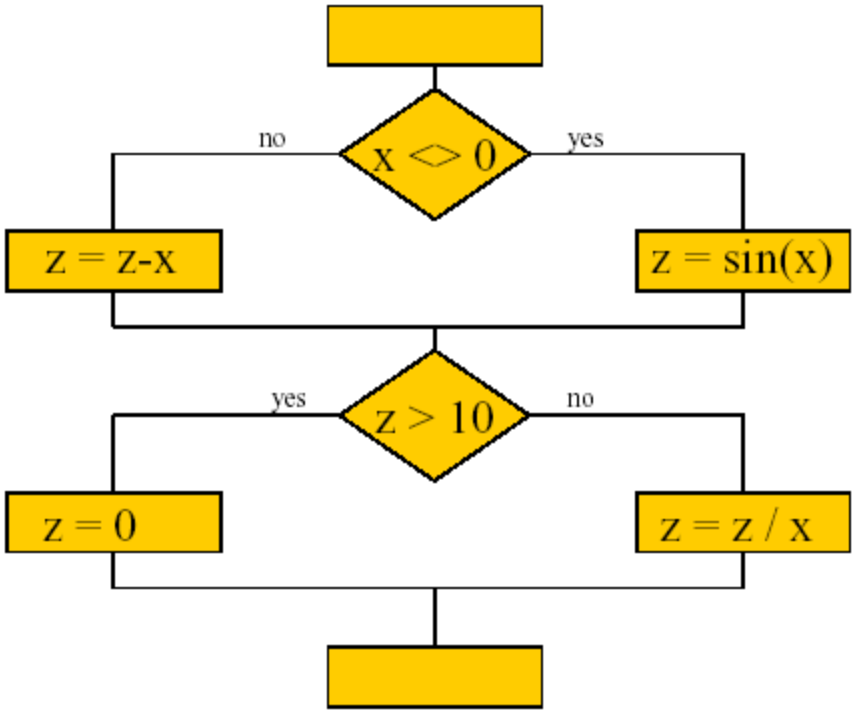


Consider:

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

What is the problem here?

Path Coverage



Consider:

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

All paths leading from initial to the final node of P's control flow graph are traversed…

All-definition-use-path Coverage

- Statements interact through *data flow*
- Each occurrence of a variable, zz say, is labeled either as
  - The definition of a variable
    - zz = 1 or read (zz)
  - or the use of variable
    - y = zz + 3 or if (zz < 9) errorB ()
- Identify all paths from the definition of a variable to the use of that definition
  - This can be done by an automatic tool
- A test case is set up for each such path

Data-flow Testing using Def-Use

DEF(S) = {x | statement S contains a definition of variable x}

USE(S) = {x | statement S contains a use of variable x}

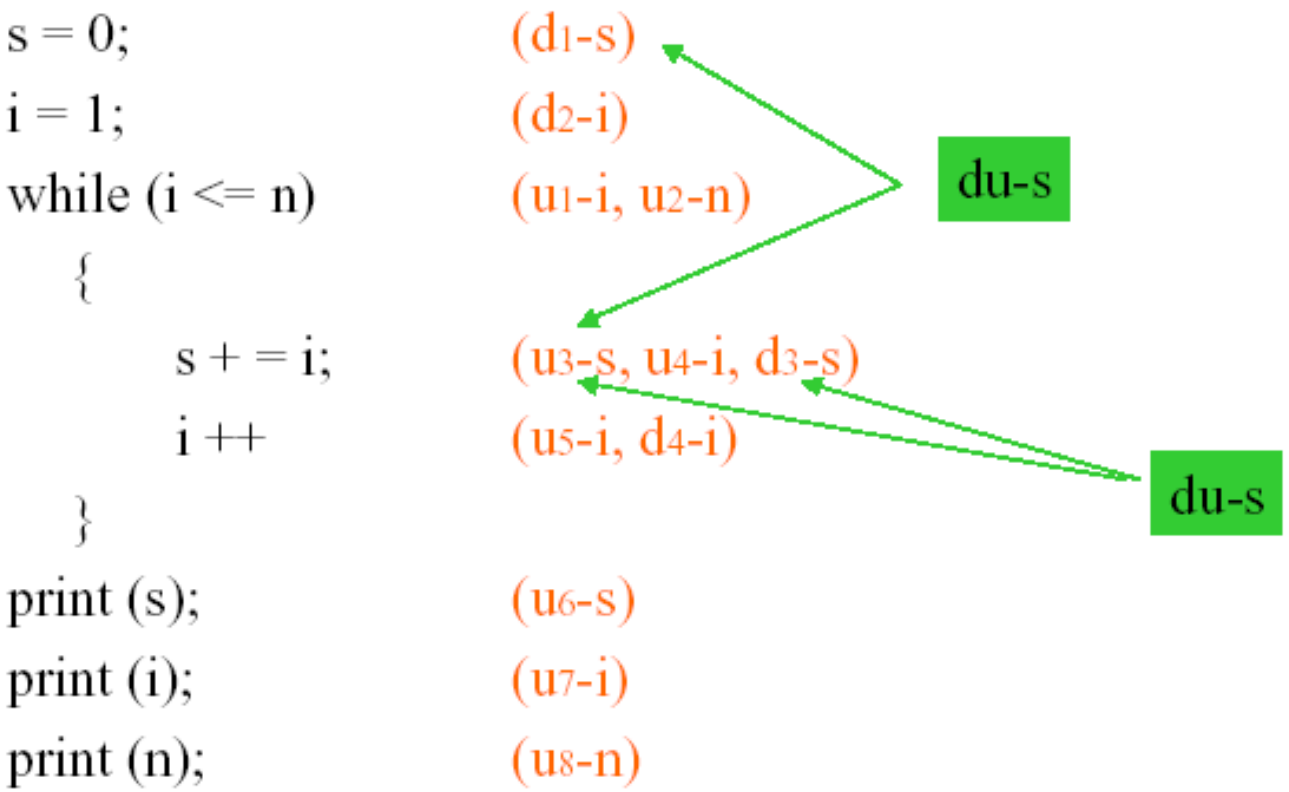
S_1: i = 1; DEFS (S_1) = {i} (d_1-i)

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 using Def-Use



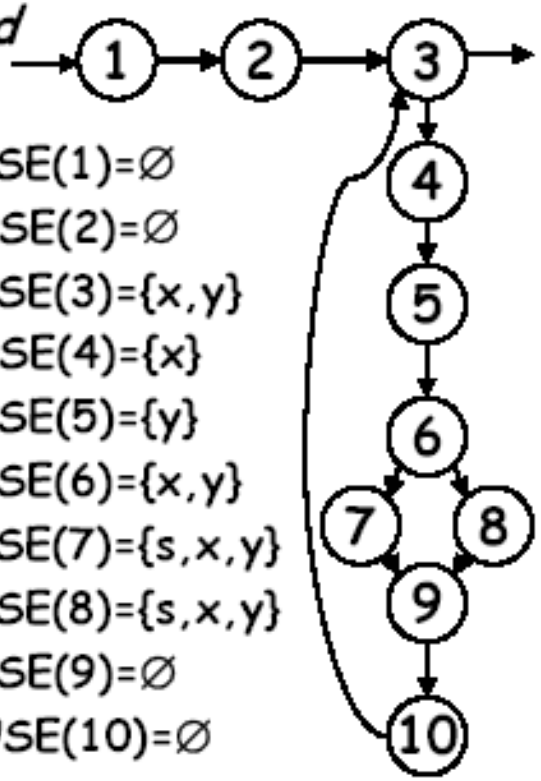
Led to research ideas of program slicing, and other static analysis techniques

Another Example of Def-Use

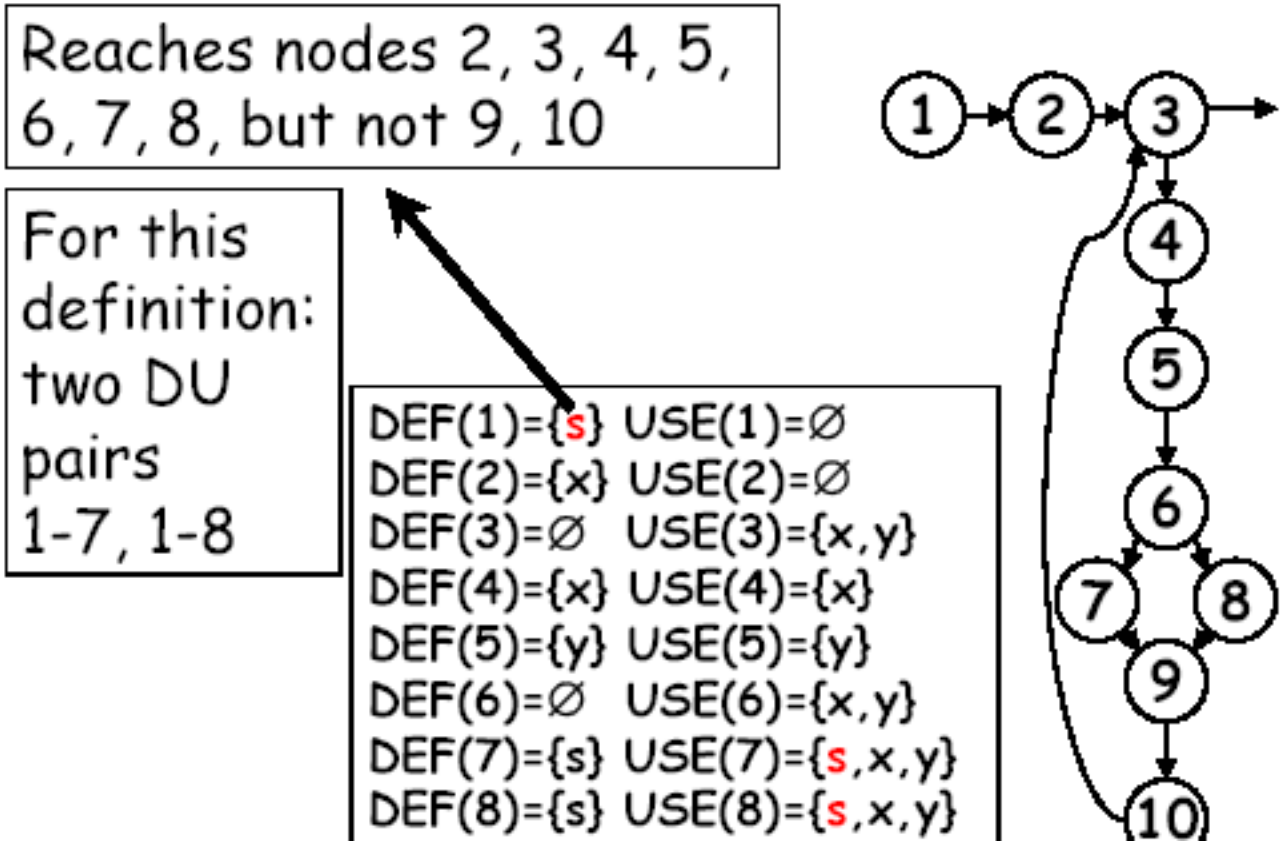
assume y is already initialized

```
1 s:=0;
2 x:=0;
3 while (x<y) {
4   x:=x+3;
5   y:=y+2;
6   if (x+y<10)
7     s:=s+x+y;
    else
8     s:=s+x-y;
  }
```

DEF(1)={s} USE(1)=∅
DEF(2)={x} USE(2)=∅
DEF(3)=∅ USE(3)={x,y}
DEF(4)={x} USE(4)={x}
DEF(5)={y} USE(5)={y}
DEF(6)=∅ USE(6)={x,y}
DEF(7)={s} USE(7)={s,x,y}
DEF(8)={s} USE(8)={s,x,y}
DEF(9)=∅ USE(9)=∅
DEF(10)=∅ USE(10)=∅



Continued…


Reaches nodes 2, 3, 4, 5, 6, 7, 8, but not 9, 10
For this definition: two DU pairs 1-7, 1-8
DEF(1)={s} USE(1)=∅
DEF(2)={x} USE(2)=∅
DEF(3)=∅ USE(3)={x,y}
DEF(4)={x} USE(4)={x}
DEF(5)={y} USE(5)={y}
DEF(6)=∅ USE(6)={x,y}
DEF(7)={s} USE(7)={s,x,y}
DEF(8)={s} USE(8)={s,x,y}
1
2
3
4
5
6
7
8
9
10

Infeasible Code

- It may not be possible to test a specific statement
  - May have an infeasible path ("dead code") in the module
- Frequently this is evidence of a fault

```
if (k < 2)
{
   if (k > 3)                    [should be: k > −3]
          ↑
      x = x * k;
}
```

(a)

```
for (j = 0; j < 0; j++)  [should be: j < 10]
                ↑
   total = total + value[j];
```

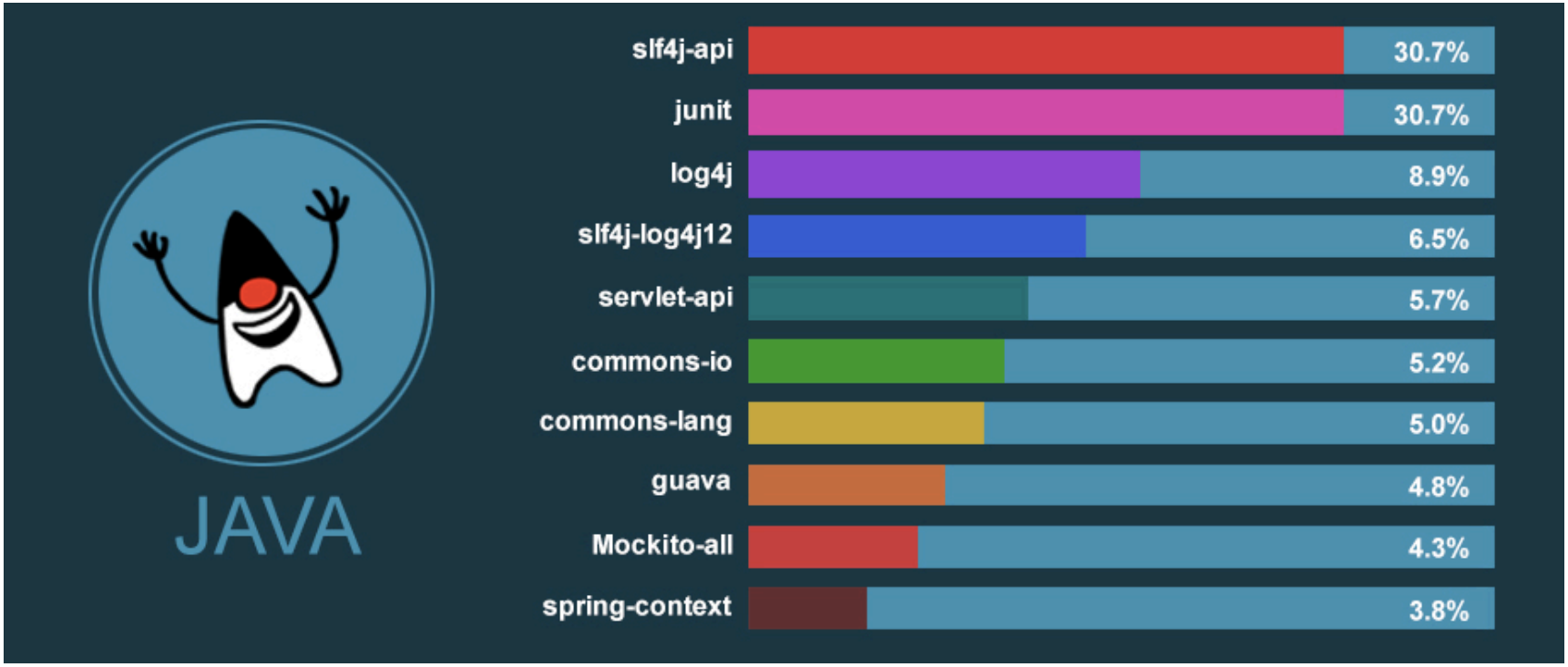
(b)

jUnit Exercise

jUnit

- Test framework
  - Initially developed to support Extreme Programming
  - Focused on unit testing
- Features and Goals
  - Trivial to add unit tests
  - Trivial to run tests
    - Tests must be completely automated (no user input)
  - Trivial presentation of test results
    - Green = good
    - Red = one or more tests failed
  - Rerun tests without exiting the testing interface
  - Large assertion API to aid condition checking
- Idea: Unit tests are good, but a pain to write -> jUnit alleviates much of the pain

jUnit


JAVA
slf4j-api
30.7%
junit
30.7%
log4j
8.9%
slf4j-log4j12
6.5%
servlet-api
5.7%
commons-io
5.2%
commons-lang
5.0%
guava
4.8%
Mockito-all
4.3%
spring-context
3.8%

Assertion API

- If any exceptions are thrown, jUnit will catch and log them, too

```
assertEquals(T expected, T actual);

assertEquals(double expected, double actual, double delta);

assertNull(Object object);

assertNotNull(Object object);

assertSame(Object expected,Object actual);

assertTrue(boolean condition);

assertFalse(boolean condition);

fail(String message);
```

When Data to Test are Inaccessible

1. Add getters/setters
2. Make a subclass just for unit test

Running the jUnit Tests

- Different ways:
  - Text UI – java junit.textui.TestRunner <Suite>
  - AWT UI - java junit.awtui.TestRunner <Suite>
  - Swing UI - java junit.swingui.TestRunner <Suite>
  - Eclipse – preferred
  - Integrated with build tools (e.g., maven, gradle)
- GUI features
  - Reload classes each run
  - Browse test cases
  - Run individual tests