Lecture 18: Structure-based Testing

Test Case First Strategy

White box testing:

- Statement Coverage
- Branch Coverage
- Condition Coverage
- Data Path Coverage

Testing with good and bad data

Testing Object Oriented Code

Developer Testing

Write the test cases first

- minimize the time to defect discovery
- forces you to think carefully about the requirements first
- exposes requirements problems early
- supports a "daily smoke test"

Limitations of Developer Testing

- Emphasis on clean tests (vs. dirty tests)
  - immature organisations have 1 dirty : 5 clean
  - mature organisations have 5 dirty : 1 clean
- Developers overestimate test coverage
- Developers tend to focus on statement coverage rather than …

Summary:

- Test-case first strategy is extremely valuable
- Test-case first strategy is not enough

Random testing isn't enough

Source: Adapted from Horton, 1999

Structurally…

```
boolean equal (int x, y) {
/* effects: returns true if
   x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

Test strategy: pick random values for x and y and test 'equals' on them

But:

...we might never test the first branch of the 'if' statement

Functionally…

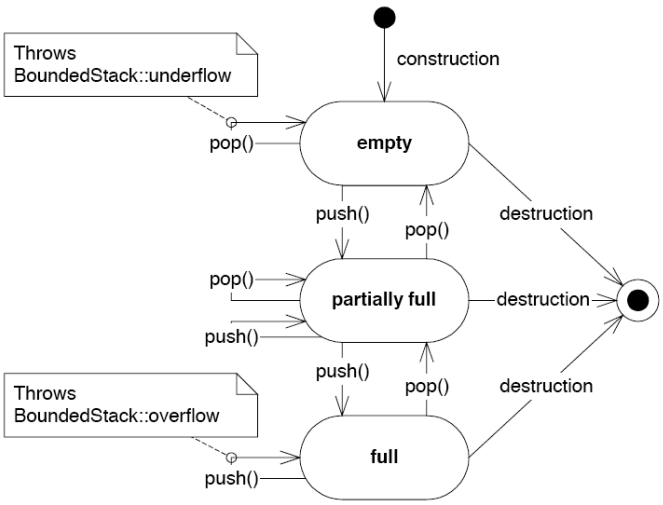
```
int maximum (list a)
/* requires: a is a list of
   integers
   effects: returns the maximum
   element in the list
*/
```

Try these test cases:

Input	Output	Correct?
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes
9 6 7 11 5	1	Yes
5 11 7 6 9	1	Yes
561 13 1024 79 86 222 97	1024	Yes
97 222 86 79 1024 13 561	1024	Yes

Why is this not enough?

Testing in every state?



Structured Basis Testing

Source: Adapted from McConnell 2004, p506-508

The minimal set of tests to cover every branch

How many tests?

start with 1 for the straight path
add 1 for each of these keywords: if, while, repeat, for, and, or
add 1 for each branch of a case statement

Example

```
int midval (int x, y, z) {
/* effects: returns median
   value of the three inputs
*/
if (x > y) {
  if (x > z) return x
  else return z }
else {
  if (y > z) return y
  else return z } }
```

Count 1 + 3 'if's = 4 test cases

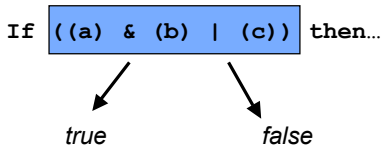
Now choose the cases to exercise the 4 paths:

e.g. x=3, y=2, z=1
x=3, y=2, z=4
x=2, y=3, z=2
x=2, y=3, z=4

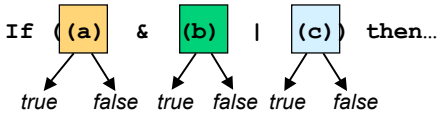
Complex Conditions

Source: Adapted from Christopher Ackermann's slides

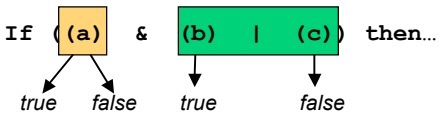
Branch Coverage:



Condition Coverage:



But can you show that each part has an independent effect on the outcome?



MC/DC Coverage

Source: Adapted from Christopher Ackermann's slides

Show that each basic condition can affect the result

```
If ((a)  &  (b)  |  (c)) then…
```

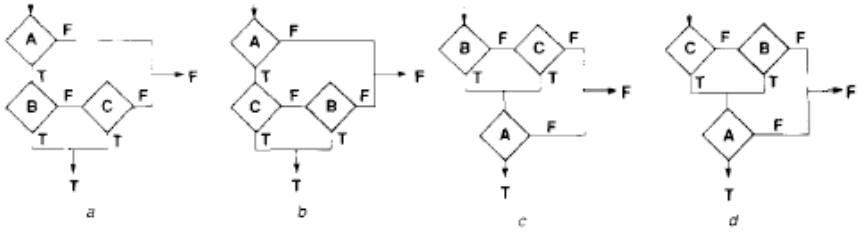
number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

Choose a minimal set:
Eg. {2, 3, 4, 6}
or {2, 3, 4, 7}

MC/DC versus Branch Coverage

Source: Adapted from Christopher Ackermann's slides

Compiler can translate conditions in the object code:



Test sets for condition/decision coverage:

{1, 8} or {2, 7} or {3, 6}

Covers all paths in the source code, but not all paths in the object code

Test sets for Modified Condition/Decision Coverage

{2, 3, 4, 6} or {2, 3, 4, 7}

Covers all paths in the object code

About MC/DC

Advantages:

- Linear growth in the number of conditions
- Ensures coverage of the object code
- Discovers dead code (operands that have no effect)

Mandated by the US Federal Aviation Administration

- In avionics, complex boolean expressions are common
- Has been shown to uncover important errors not detected by other test approaches

It's expensive

- E.g. Boeing 777
- approx 4 million lines of code, 2.5 million newly developed
- approx 70% of this is Ada (rest is C or assembler)
- Total cost of aircraft development: $5.5 billion
- Cost of testing to MC/DC criteria: approx $1.5 billion

Dataflow testing

Source: *Adapted from McConnell 2004, p506-508*

Things that happen to data:

- Defined - data is initialized but not yet used
- Used - data is used in a computation
- Killed - space is released
- Entered - working copy created on entry to a method
- Exited - working copy removed on exit from a method

Normal life:

- Defined once, Used a number of times, then Killed

Potential Defects:

- D-D: variable is defined twice
- D-Ex, D-K: variable defined but not used
- En-K: destroying a local variable that wasn't defined?
- En-U: for local variable, used before it's initialized
- K-K: unncessary killing - can hang the machine!
- K-U: using data after it has been destroyed
- U-D: redefining a variable after is has been used

Testing all D-U paths

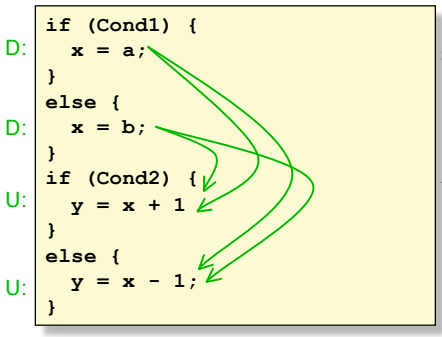
Source: Adapted from McConnell 2004, p506-508

The minimal set of tests to cover every D-U path

How many tests?

1 test for each path from each definition to each use of the variable

Example



Structured Basis Testing:

2 test cases is sufficient

Case 1: Cond1=true, Cond2=true

Case 2: Cond1=false, Cond2=false

All DU testing:

Need 4 test cases

Boundary Checking

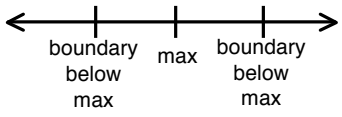
Source: Adapted from McConnell 2004, p506-508

Boundary Analysis

Every boundary needs 3 tests:

Example:

```
if (x < MAX) {
  … }
```



test for MAX+1, MAX-1 and MAX

Compound Boundaries

When several variables have combined boundaries

```
for (i=0; i<Num; i++) {
  if (a[i] < LIMIT) {
    y = y+a[i];
  }
}
```

Test when lots of array entries are close to the max?

Test when lots of entries are close to zero?

Data Classes

Source: Adapted from McConnell 2004, p506-508

Classes of Bad Data

- Too little data (or no data)
- Too much data
- The wrong kind of data (invalid data)
- The wrong size of data
- Uninitialized data

Classes of Good Data

- Nominal cases - middle of the road, expected values
- Minimum normal configuration
- Maximum normal configuration
- Compatibility with old data

Testing Object Oriented Code

Encapsulation

- If the object hides it's internal state, how do we test it?
- E.g. add methods only to be used in testing, which expose internal state
- But: how do we know these extra methods are correct?

Inheritance

- When a subclass extends a well-tested class, what extra testing is needed?
- e.g. Test just the overridden methods?
- But with dynamic binding, this is not sufficient
- e.g. other methods can change behaviour because they call over-ridden methods

Polymorphism

- When class A calls class B, it might actually be interacting with any of B's subclasses…