

Agenda

Unit/White Box Testing

Testing Models - Review

- **White box testing . . . You know the code**
 - Given knowledge of the internal workings, you thoroughly test what is happening on the inside
 - Close examination of procedural level of detail
 - Logical paths through code are tested
 - **Conditionals**
 - **Loops**
 - **Branches**
 - Status is examined in terms of expected values
 - Impossible to thoroughly exercise all paths
 - **Exhaustive testing grows without bound**
 - Can be practical if a limited number of “important” paths are evaluated
 - Can be practical to examine and test important data structures

Types of Testing

- **Unit Testing**
 - Done by programmer(s)
 - Generally all white box
 - Automation desirable for repeatability
- **Integration Testing**
 - Done by programmer as they integrate their code into code base
 - Generally white box, maybe some black box
 - Automation desirable for repeatability
- **Functional/System Testing**
 - It is recommended that this be done by external test group
 - Mostly **black box** so that testing is not 'corrupted' by too much knowledge
 - Test automation desirable
- **Acceptance Testing**
 - Generally done by customer/customer representative in their environment . . . Definitely **black box**

Build scaffolding for incomplete programs

- Stubs and drivers are code that are (temporarily) written in order to unit test a program
- **Driver** is a *software module used to invoke a module under test and often, provide test inputs, controls, and monitor execution and report test results*

```
▪ main () {  
    movePlayer(Player, diceRoll);  
}
```

- **Stub** is a module that simulates components that aren't written yet, formally defined as a *computer program statement substituting for the body of a software module that is or will be defined elsewhere*

```
▪ public void movePlayer(Player player, int diceValue) {  
    player.setPosition(1);  
}
```

Devising a prudent set of test cases

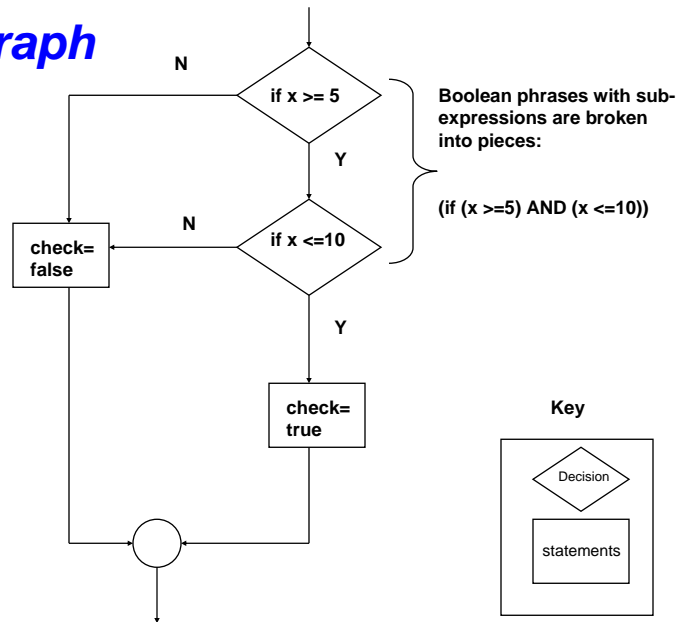
- **Equivalence Class/Boundary Value Analysis**
 - Still applies!
- **Basis Set**
- **A metric for assessing how good your test suite is.**
 - Method Coverage
 - Statement Coverage
 - Decision/Branch Coverage
 - Condition Coverage
- **Think diabolically**

▪ **The act of planning tests is often thought of as a better debugger than testing itself.**

Basis Set

- **Compute Cyclomatic number - $V(G)$**
 - This gives us an estimate of how many tests must be designed and executed to guarantee coverage
 - The number of independent paths that must be tested to ensure that all statements have been executed at least once and every condition will have been executed on its true and false side!!
- **Computed 2 ways . . .with same result ☺:**
 1. $V(G)$ = The number of regions of the flow graph
 2. $V(G) = P + 1$ (P = number of predicate nodes)

Flow Graph



100% Method Coverage

- All methods in all classes have been called
- Test case 1: Foo(0, 0, 0, 0, 0) = 0.0
- `float foo (int a, b, c, d, e) {`
 - `if (a == 0) {`
 - `return 0.0;`
 - `}`
 - `int x = 0;`
 - `if ((a==b) OR ((c==d) AND bug(a))) {`
 - `x=1;`
 - `}`
 - `e = 1/x;`
 - `return e;`
 - `}`

100% Statement Coverage

- All lines in a method have been executed
- Test case 2: Foo(1, 1, 1, 1, 1) = 1.0

```
void foo (int a, b, c, d, e) {  
    if (a == 0) {  
        return;  
    }  
    int x = 0;  
    if ((a==b) OR ((c==d) AND bug(a) )) {  
        x=1;  
    }  
    e = 1/x;  
}
```

100% Branch/Decision Coverage

- All predicates have been true and false
- Test case 3: Foo(1, 2, 1, 2, 1) ← division by zero!

```
void foo (int a, b, c, d, e) {  
    if (a == 0) {  
        return;  
    }  
    int x = 0;  
    if ((a==b) OR ((c==d) AND bug(a) )) {  
        x=1;  
    }  
    e = 1/x;  
}
```

Line #	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	

100% Condition Coverage

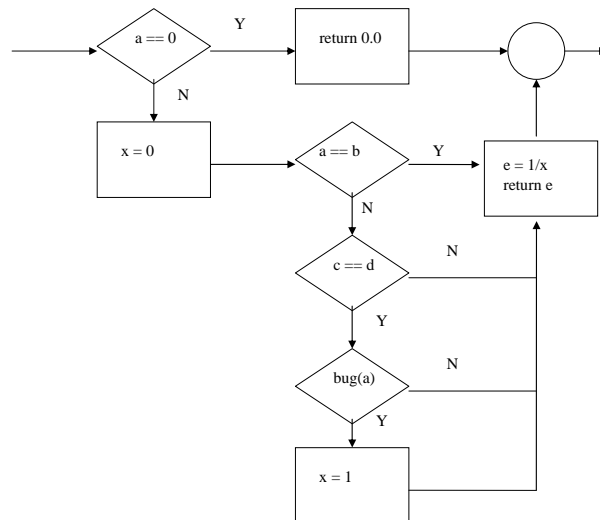
- All sub-expression predicates have been true and false
- Test case 4: Foo(1, 2, 1, 1, 1)

```

void foo (int a, b, c, d, e) {
    if (a == 0) {
        return;
    }
    int x = 0;
    if ((a==b) OR ((c==d) AND bug(a) )) {
        x = 1;
    }
    e = 1/x;
}
    
```

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)		Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)		

Compound Predicate



Triangle Program

- **Given: three sides that ARE a triangle**
- **Determine if the triangle is equilateral, isosceles, or scalene**

- **Derive a flowgraph for the program.**
- **Compute the cyclomatic number of your program.**
- **What test cases do you need? [testid, description, expected results, actual results]**
- **What kind of method, statement, branch, and condition coverage do these test cases do you have?**

Loops

- **Write a test case such that you:**
 - **Don't go through the loop at all**
 - **Go through the loop once**
 - **Go through the loop twice**

```

Float calcRentalFee(Tape[] tapes, Customer customer){
    float total = 0;
    for(int i = 0; i < tapes.length; i++){
        total += tapes[i].price;
    }
    if (tapes.length > 10){
        total *= .8;
    } else if(tapes.length > 5){
        total *= .9;
    }
    if(customer.isPremium()){
        total *= .9;
    }
    return total;
}

```

```

tapes[0].price = 0
tapes[1].price = 1
tapes[2].price = 2
... And so on

```

- What test cases should you run? Consider:
 - Basis path
 - Equivalence class partitioning
 - Boundary value analysis
 - Diabolical testing
 - Statement coverage
 - Branch coverage
 - Condition coverage

Devising a prudent set of test cases

- **Equivalence Class/Boundary Value Analysis**
 - Still applies!
- **Basis Set**
- **A metric for assessing how good your test suite is.**
 - Method Coverage
 - Statement Coverage
 - Decision/Branch Coverage
 - Condition Coverage
- **Think diabolically!**