Unit Testing 101

Black Box v. White Box

Definition of V&V

- Verification - is the product correct
- Validation - is it the correct product

Background Info…

- Textbook's Definition of Testing:
  - Software Testing is a <u>formal</u> process carried out by a <u>specialized team</u> in which a software unit, several integrated units or an entire software package are examined by <u>running the programs</u> on a computer.
- Testing is the single biggest SQA task.
  - on average, 24% of the development budget is testing
- Code Testing ≠ Code Walkthrough
- Objectives of Testing:
  - reveal errors
  - after retesting, assurance of acceptable quality
  - compile record of software errors

textbook pages 178-182

Laws of Testing

- The best person to test your code is someone else.
- A good test is one that finds an error.
- Testing can not prove the absence of errors.
- Complete test coverage is impossible, so concentrate on problem areas.
- It cost a lot less to remove bugs early.

Testing Stages

- **Unit Testing**
  - modules of code
- **Integration Testing**
  - design
- **Validation Testing**
  - requirements
- **System Testing**
  - system engineering

today

next class

Reality Check…

- Why not just run the whole thing and see if it gives us the right answer or if it crashes?

Unit Testing

- **White Box Testing**
  - testing a module of code based on the source code
- **Black Box Testing**
  - testing a module based on its description and/or the requirements specification
  - Also called "functional" and "behavioral" testing
- Proof of Correctness
  - mathematically-based analysis of the requirements, similar to theorem proving

White Box Testing Fundamentals

- White Box testing is ***much*** more expensive than Black Box testing.
- White Box is most appropriate when we must assure that the calculations are correct.
- Covering every possible path through a module is usually not practical.
  - 10 if-then statements might require 1024 test cases
  - instead, base the number of tests on the complexity of the module

Types of Code Coverage

- **Function coverage**
  - Has each function in the program been executed?
- **Statement coverage**
  - Has each line of the source code been executed?
- **Condition coverage**
  - Has each evaluation point (such as a true/false decision) been executed?
- **Path coverage**
  - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage**
  - Has every possible call and return of the function been executed?

Example One

```
int example1 (int value, boolean cond1, boolean cond2)
{
   if ( cond1 )
      value ++;
   if ( cond2 )
      value --;
   return value;
}
```

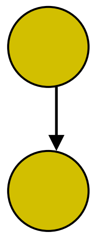
- Total Statement Coverage with one case - True True.
- Total Path Coverage with four paths - TT TF FT FF.
- But, total path coverage is usually impractical, so Basis Path Testing is usually better.

Basis Path Testing

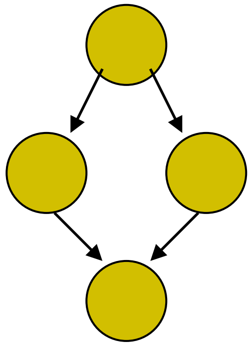
Objective is to test each conditional statement as both true and false

1. Draw a Flow Graph
2. Determine the Cyclomatic Complexity
   - CC = number of regions
   - CC = E - N + 2
3. Max Number of tests = CC
4. Derive a basis set of independent paths
5. Generate data to drive each path

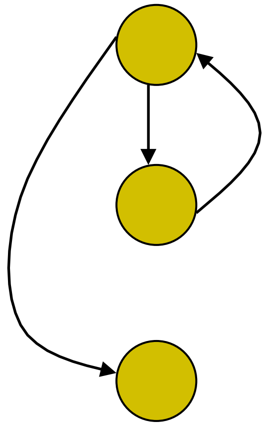
Flow Graphs



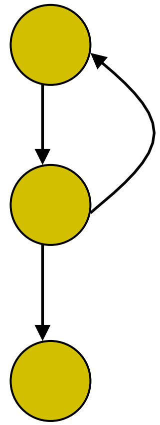
simple sequence



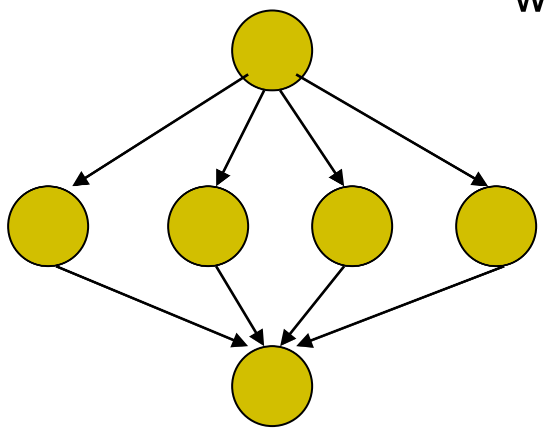
if then else



while



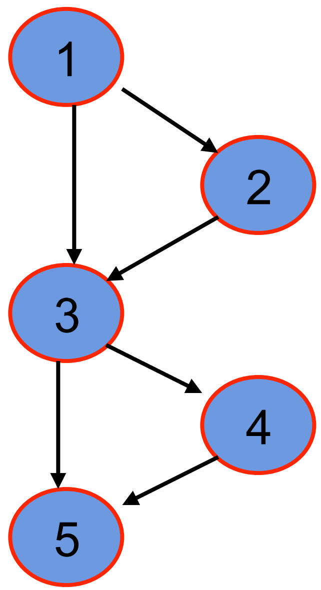
do until



case statement

Example One - using basis path

```
  int example1 (int value, boolean cond1, boolean cond2)
  {
1     if ( cond1 )
2        value ++;
3     if ( cond2 )
4        value --;
5     return value;
  }
```



Complexity = 3

Basis Paths	Test Data
1 3 5	false false
1 2 3 5	true false
1 2 3 4 5	true true

Example One - sample driver

<u>Test Data</u>

```
false  false
true   false
true   true
```

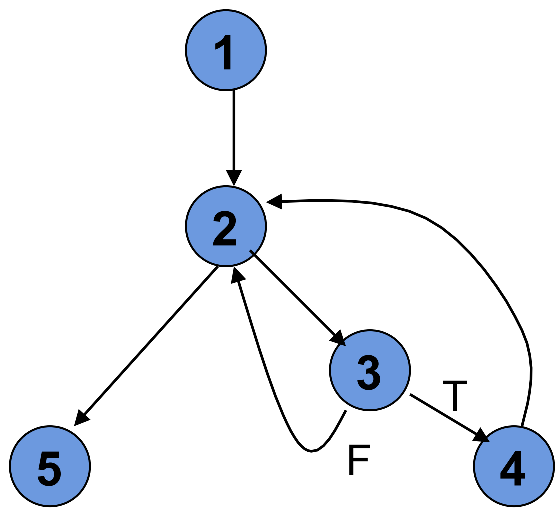
```
int example1 (int value, boolean cond1, boolean cond2)
{
   ...
}

print ("test one   ", example1 (5, false, false));
print ("test two   ", example1 (5, true, false));
print ("test three ", example1 (5, true, true));
```

Example Two

```
float avg_negative_balance (int arraysize,
                            float balances[])
{
   float total = 0.0;
   int count = 0;
   for I = 1 to arraysize
      if ( balances[I] < 0 )
         total += balances[I];
         count ++;
      end if;
   end for;
   return total / count;
}
```

Example Two - using basis path

```
float avg_negative_balance (int arraysize,
                            float balances[])
{
1   float total = 0.0;
    int count = 0;
2   for I = 1 to arraysize
3      if ( balances[I] < 0 )
4         total += balances[I];
          count ++;
       end if;
    end for;
5   return total / count;
}
```



Basis Paths	Test Data
1 2 5	arraysize = 0
1 2 3 2 5	arraysize = 1, balance[1] = 25
1 2 3 4 2 5	arraysize = 1, balance[1] = -25

Example Two - Test Report

Test #	Test Data	Result
1	array size = 0	failed
2	size = 1 array = [25]	passed
3	size = 1 array = [-25]	passed

Errors not detected:

- precious errors when "total" gets very small

Loop Testing

Errors often occur near the beginnings and ends of loops.

- For each loop that iterates max N times, test
  - N = 0
  - N = 1
  - N = max -1
  - N = max
  - N = max + 1
- For nested loops
  - repeat above for the innermost loop, outer loop iterates once
  - then repeat all 5 possibilities for outer loop, while inner loop iterates only once

Example Two - using loop testing

```
float avg_negative_balance (int arraysize, float balances[])
{
   float total = 0.0;
   int count = 0;
   for I = 1 to arraysize
      if ( balances[I] < 0 )
         total += balances[I];
         count ++;
      end if;
   end for;
   return total / count;
}
```

Test Case	Test Data
N = 0	size = 0
N = 1	size = 1
N = max-1	size = 999 (if SRS says max=1000)
N = max	size = 1000
N = max+1	size = 1001, but array has only 1000 elements

Simple Black Box Testing

- Create Test Cases for
  - Easy-to-compute data
  - Typical data
  - Boundary / extreme data
  - Bogus data

Boundary Value Analysis

- Form of Black Box testing
  - similar to "Equivalence Partitioning"
- Rationale:
  - off-by-one is the most common coding error
  - errors usually occur near the ends - **boundaries**

BVA Test Cases

- if an input specifies a range bounded by A and B,
  1. test value = A
  2. test value = B
  3. test value < A
  4. test value > B
- do the same for outputs

Example Two - using BVA

Inputs and Outputs to be Tested:

- Inputs
  - size
    - range from 0 to 1000
  - array of balances
    - elements are positive and negative floats
- Outputs
  - average of negative balances

Example Two - using BVA

Test Cases:

- Based on Input Boundaries
  1. size = 0
  2. size = 1, balance[1] is negative
  3. size = 1, balance[1] is positive
  4. size = 1000, all negatives
  5. size = 1000, all positives
  6. size = 1001
- Based on Output Boundaries
  7. a test where the average negative is huge
  8. a test where the average negative is small