Path Testing (cont.)

Dr. John H Robb, PMP, IEEE SEMC
UTA Computer Science and Engineering

Sample Test Problems

Sample Problem 1

```
1    public double applyDiscount (double balance, boolean prime,
                                          int years_prime, double discount) {
2
3    double invoice;
4    invoice=balance;
5    if (balance>3_000.00)
6        if (prime)
7            if (years_prime>5)
8                invoice=(1-discount)*balance;
9    return invoice;
10   }
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$3,000.01	TRUE	5	0.5	$1,500.00

what is wrong, if anything, with the above test case designed to test statement 8?

a) no error - it is a legitimate test case.

b) one or more of the expected outputs is incorrect.

c) the logic will never get executed because of the conditions given.

d) it is not one of the test cases we would use to test the code.

Sample Problem 1

```
1     public double applyDiscount (double balance, boolean prime,
                                  int years_prime, double discount) {
2
3     double invoice;
4     invoice=balance;
5     if (balance>3_000.00)
6         if (prime)
7             if (years_prime>5)
8                 invoice=(1-discount)*balance;
9     return invoice;
10    }
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$3,000.01	TRUE	5	0.5	$1,500.00

what is wrong, if anything, with the above test case designed to test statement 8?

a) no error - it is a legitimate test case.

b) one or more of the expected outputs is incorrect.

c) the logic will never get executed because of the conditions given.

d) it is not one of the test cases we would use to test the code.

Sample Problem 2

```
public double applyDiscount (double balance, boolean prime,
                              int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **\$0.00 <= balance <= \$10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to \$0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	\$3,000.01	TRUE	6	0.5	\$1,500.01

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.

Sample Problem 2

```
public double applyDiscount (double balance, boolean prime,
                                        int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test Case	Inputs				Exp Out
	balance	prime	years_prime	discount	Result
1	$3,000.01	TRUE	6	0.5	$1,500.01

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.

Sample Problem 3

```
public double applyDiscount (double balance, boolean prime,
                             int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$3,000.00	TRUE	6	0.5	$1,500.00

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.

Sample Problem 3

```
public double applyDiscount (double balance, boolean prime,
                                                  int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$3,000.00	TRUE	6	0.5	$1,500.00

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.
Answer c is only applicable when I ask about a specific line of code

Sample Problem 4

```
public double applyDiscount (double balance, boolean prime,
                              int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$2,999.99	TRUE	6	0.5	$1,499.99

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.

Sample Problem 4

```
public double applyDiscount (double balance, boolean prime,
                                        int years_prime, double discount) {

double invoice;
invoice=balance;
if (balance>3_000.00)
    if (prime)
        if (years_prime>5)
            invoice=(1-discount)*balance;
return invoice;
}
```

Assume:

1. **$0.00 <= balance <= $10,000.00**
2. **0 <= years_prime <= 100**
3. **Significance and truncation to $0.01**

Test	Inputs				Exp Out
Case	balance	prime	years_prime	discount	Result
1	$2,999.99	TRUE	6	0.5	$1,499.99

what is wrong, if anything, with the above test case?

a) no error - it is a legitimate test case.
b) one or more of the expected outputs is incorrect.
c) the logic will never get executed because of the conditions given.
d) it is not one of the test cases we would use to test the code.

Coverage Exercises

Student Exercises

- For the following code snippet what coverage levels are achieved for the following cases?

- a) t1 & t2
- b) t1 & t3
- c) t2 & t4

- t1=> x=0, y=2
- t2=> x=0, y=3
- t3=> x=-1, y=4
- t4=> x=-1, y=2

```
if (x<0 | y<3)
      z=1;
else
      z=2;
```

Coverage goal	What's measured?
Statement	Each node is reached
Branch	Each edge is reached
Decision	Each edge is reached
Condition	Each condition is tested both T&F **(for this ignore decision coverage)**
Basis Path	All unique paths
Boundary Value	Each boundary value is tested at partition edges
Path	All possible paths

Student Exercises

- a) t1 & t2
- b) t1 & t3
- c) t2 & t4

a) all but condition coverage and boundary value coverage

b) Condition only

c) all

Coverage goal	What's measured?
Statement	Each node is reached
Branch	Each edge is reached
Decision	Each edge is reached
Condition	Each condition is tested both T&F **(for this ignore decision coverage)**
Basis Path	All unique paths
Boundary Value	Each boundary value is tested at partition edges
Path	All possible paths

Student Exercises

- How many branches and decisions are in the following code snippet?

```
for (i = 0; i < arrayOfInts.length; i++) {
   for (j = 0; j < arrayOfInts[i].length; j++) {
         if (arrayOfInts[i][j] == searchfor) {
            goto search; } } }

search:
```

- How many branches and decisions are in the following code snippet?

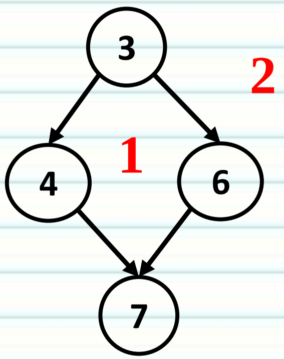
```
return (year == 1582 && month ==10 && day>=5 && day <= 14);
```

- Which is harder to achieve branch or decision coverage?

Achieving Condition Coverage (Example 1)

- Returning to the Gregorian Calendar example

```
1   public boolean gregCalend (int year, int month, int day) {
2   boolean a;
3       if ((year == 1582) & (month==10) & (day>=5) & (day <= 14))
4           a=true;
5       else
6           a=false;
7       return a;
8   }
```



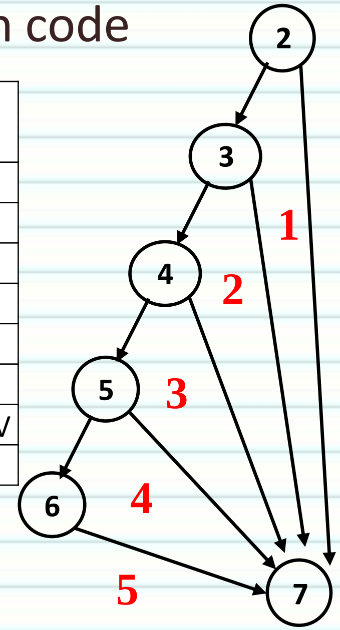
Test Case Number	Inputs			Exp Out	Basis Path	MCDC
	Year	Month	Day	Return		
1	1582	10	5	TRUE	3-4-7	MCDC TTTT
2	1581	10	5	FALSE	3-6-7	MCDC FTTT
3	1582	11	5	FALSE	-	MCDC TFTT
4	1582	10	4	FALSE	-	MCDC TTFT
5	1582	10	15	FALSE	-	MCDC TTTF
6	1583	10	5	FALSE	-	Untested year BV
7	1582	9	5	FALSE	-	Untested month BV
8	1582	10	14	TRUE	-	Untested day BV

Arrange MCDC tests:
TTTT, FTTT
Add MCDC tests:
TFTT, TTFT, TTTF

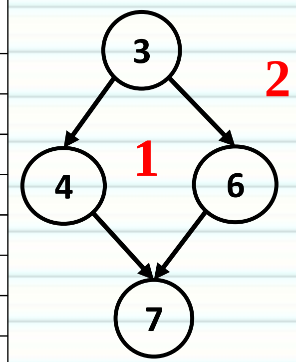
Add tests to test untested boundary conditions (as before)

Achieving Condition Coverage (Example 1 cont.)

- Let's compare the two tables for the Gregorian calendar function for multiple decision code and multiple condition code



Test Case Number	Inputs Year	Month	Day	Exp Out Return	Basis Path	MCDC
1	1582	10	5	TRUE	2-3-4-5-6-7	MCDC TTTT
2	1581	10	5	FALSE	2-7	MCDC FTTT
3	1582	11	5	FALSE	2-3-7	MCDC TFTT
4	1582	10	4	FALSE	2-3-4-7	MCDC TTFT
5	1582	10	15	FALSE	2-3-4-5-7	MCDC TTTF
6	1583	10	5	FALSE	-	Untested year BV
7	1582	9	5	FALSE	-	Untested month BV
8	1582	10	14	TRUE	-	Untested day BV



Test Case Number	Inputs Year	Month	Day	Exp Out Return	Basis Path	MCDC
1	1582	10	5	TRUE	3-4-7	MCDC TTTT
2	1581	10	5	FALSE	3-6-7	MCDC FTTT
3	1582	11	5	FALSE	-	MCDC TFTT
4	1582	10	4	FALSE	-	MCDC TTFT
5	1582	10	15	FALSE	-	MCDC TTTF
6	1583	10	5	FALSE	-	Untested year BV
7	1582	9	5	FALSE	-	Untested month BV
8	1582	10	14	TRUE	-	Untested day BV

The only changes are the test case numbers and the basis path

Achieving Condition Coverage (Example 1 cont.)

- What if it is coded as the following? This is the most likely coding of this function.

```
public boolean gregCalend (int year, int month, int day) {
        return ((year == 1582) & (month==10) & (day>=5) & (day <= 14))
}
```

This is the CFG - we don't have much to work with

Cyclomatic complexity is 2 (it has a true and false)

Test Case	Inputs			Exp Out		
Number	Year	Month	Day	Return	Basis Path	MCDC
1	1582	10	5	TRUE	3 true	MCDC TTTT
2	1581	10	5	FALSE	3 false	MCDC FTTT
3	1582	11	5	FALSE	-	MCDC TFTT
4	1582	10	4	FALSE	-	MCDC TTFT
5	1582	10	15	FALSE	-	MCDC TTTF
6	1583	10	5	FALSE	-	Untested year BV
7	1582	9	5	FALSE	-	Untested month BV
8	1582	10	14	TRUE	-	Untested day BV

Achieving Condition Coverage (Example 1 cont.)

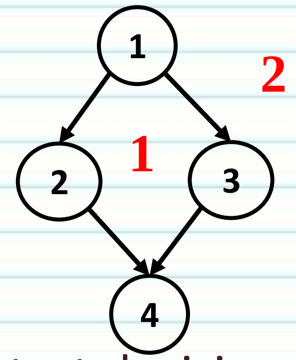
- In all three cases we have used the Basis Path approach as the starting point.
    1. Use Basis path to develop test case
    2. Use MCDC to solve multiple condition expressions
    3. Add in any untested BVs, extreme range values, and MCDC test cases

- Why do we use Basis path then?
- Because there are certain constructs where it is very helpful and that MCDC cannot provide a solution
    - Threshold logic (R/Y/G lights problems)
    - Sequential if statements

Achieving Condition Coverage (Example 2)

- For the expression

```
if (a&&b)
    x=1;
else
    x=2;
```

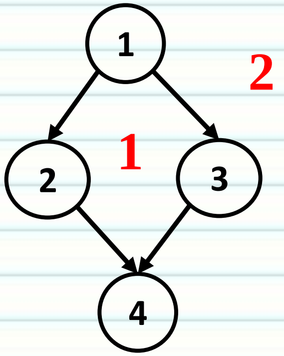


- we get the CFG above, which allows us to test decisions and statements
- But for multiple condition code this is inadequate testing
  - it leaves conditions (and possibly) boundary values untested
- We don't want to develop a condition level CFG for multiple reasons:
  1. It gets very complicated very quickly - especially for medium and large sized methods
  2. The extra nodes in the condition level CFG don't correspond to physical statements in the code
- So, we will use the decision level CFG and mentally add the extra tests required by MC/DC

Achieving Condition Coverage (Example 2 - cont.)

- Previous expression

```
1    if (a&&b)
2        x=1;
3    else
4        x=2;
```



- Here are the tests for the decision level CFG

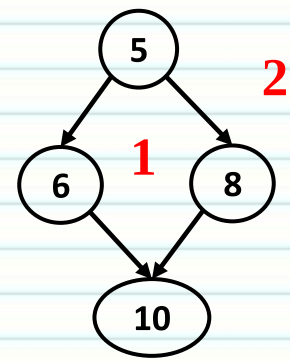
Test Case Number	Inputs		Exp Out	Basis Path Tested
	a	b	x	
1	T	T	T	1,2,4
2	F	T	F	1,3,4
3	T	F	F	-

Notice that decision level testing only requires either 2 or 3

- For the above fragment, there are 2 conditions so MC/DC requires 3 tests - one additional to the CFG above (n-1 extra tests, here n=2)
  – Our MC/DC tests are 1, 2, AND 3

Achieving Condition Coverage (Example 3)

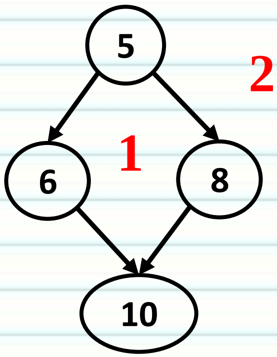
```
1     public boolean getres(boolean reservation,
                        boolean tableavailable, boolean tip) {
2
3         boolean result;
4
5         if (reservation || (tableavailable && tip))
6             result = true;
7         else
8             result = false;
9
10        return result;
11    }
```



MCDC solution for a + bc is: TFT, FFT, FTT, FTF: with result of T, F, T, F

Achieving Condition Coverage (Example 3 - cont)

```
1    public boolean getres(boolean reservation,
                               boolean tableavailable, boolean tip) {
2
3         boolean result;
4
5         if (reservation || (tableavailable && tip))
6              result = true;
7         else
8              result = false;
9
10        return result;
11   }
```



Test Case	Inputs			Exp Out		
Number	reservation	tableavailable	tip	return	Basis Path	MCDC
1	TRUE	FALSE	TRUE	TRUE	5-6-10	TFT
2	FALSE	FALSE	TRUE	FALSE	5-8-10	FFT
3	FALSE	TRUE	TRUE	TRUE	-	FTT
4	FALSE	TRUE	FALSE	FALSE	-	FTF

Test case 1 uses TFT=T
Test Case 2 uses FFT=F
These are the first two MCDC solutions in sequence - we add 3 and 4

Achieving Condition Coverage (Example 3 - cont)

- What if it is coded as the following? This is the most likely coding of this function.

```
4 public boolean getres(boolean reservation,boolean tableavailable, boolean tip) {
5
6   return (reservation || (tableavailable && tip));
7 }
```

6

This is the CFG - we don't have much to work with

Cyclomatic complexity is 2 (it has a true and false)

Test Case Number	Inputs			Exp Out	Basis Path	MCDC
	reservation	tableavailable	tip	return		
1	TRUE	FALSE	TRUE	TRUE	6 true	TFT
2	FALSE	FALSE	TRUE	FALSE	6 false	FFT
3	FALSE	TRUE	TRUE	TRUE	-	FTT
4	FALSE	TRUE	FALSE	FALSE	-	FTF

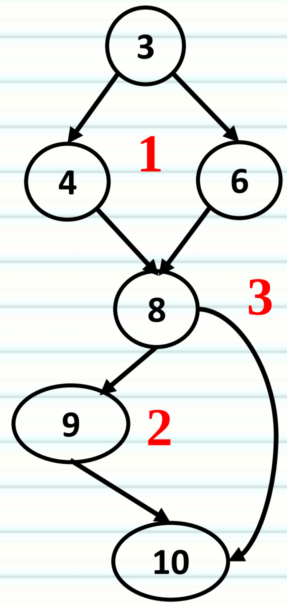
Test case 1 uses TFT=T
Test Case 2 uses FFT=F
These are the first two MCDC solutions in sequence - we add 3 and 4

Achieving Condition Coverage (Example 4)

```
public void operateMicrowave (boolean startButton, boolean stopButton, boolean doorOpen) {

if (startButton && !stopButton && !doorOpen )
      heatOn=true;
else
      heatOn=false;

if (timerValue>0)
      timerValue--;
}
```



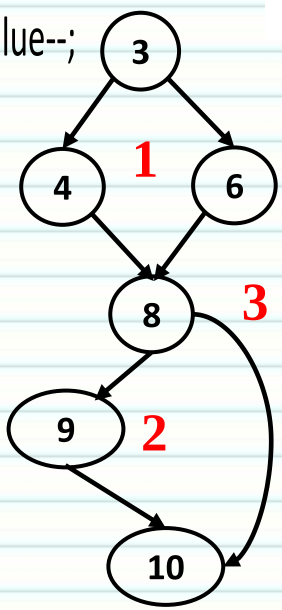
heatOn and timerValue are private class variables

Achieving Condition Coverage (Example 4 cont.)

```
1   public void operateMicrowave (boolean startButton, boolean stopButton, boolean doorOpen) {
2
3   if (startButton && !stopButton && !doorOpen )
4         heatOn=true;
5   else
6         heatOn=false;
7
8   if (timerValue>0)
9         timerValue--;
10  }
```

Test Case	Inputs				Exp Out		Basis Path Tested
	startButton	stopButton	doorOpen	timerValue	heatOn	timerValue	
1	T	F	F	1	T	0	3-4-8-9-10
2	F	F	F	1	F	0	3-6-8-9-10
3	F	F	F	0	F	0	3-6-8-10
4	T	T	F	1	F	0	-
5	T	F	T	1	F	0	-

- use timerValue from other MCDC tests



1) MCDC solution for ab'c' = TFF,FFF,TTF,TFT
2) We have combined MCDC with basis path as shown above
3) test cases 4 and 5 are the two added MCDC test cases

Achieving Condition Coverage (Example 5)

```
public void carCollAlarms (boolean selfDrive, double speed, double distance) {
  carCollAlert=carCollWarn=carCollCaut=emerBrake=false;
  if (selfDrive)
    if (speed > 50.0)
      if (distance<=150.0)
        if (distance>100.0)
          carCollCaut=true;
        else
          if (distance>50.0)
            carCollWarn=true;
          else {
            carCollAlert=true;
            if (distance<=25.0)
              emerBrake=true;}}
```

From M09 - what would this look like as a multiple condition decision statement?

Achieving Condition Coverage (Example 5 cont.)

```
public void carCollAlarms (boolean selfDrive, double speed, double distance) {
      boolean temp = (selfDrive) && (speed > 50.0);
      carCollCaut  = temp && (distance<=150.0) && (distance>100.0);
      carCollWarn  = temp && (distance<=100.0) && (distance>50.0);
      carCollAlert = temp && (distance<=50.0);
      emerBrake =    temp && (distance<=25.0);
}
```

9

This is the CFG - we don’t have much to work with

Cyclomatic complexity is 5 (each statement is a decision)

We end up with the same tests as M09 after eliminating repeated tests - they will be in different order

Interestingly enough the code is more time and space efficient written as in M09 than here because of the ranges for distance that are tested for each statement here but not in M09

Achieving Condition Coverage (Example 6)

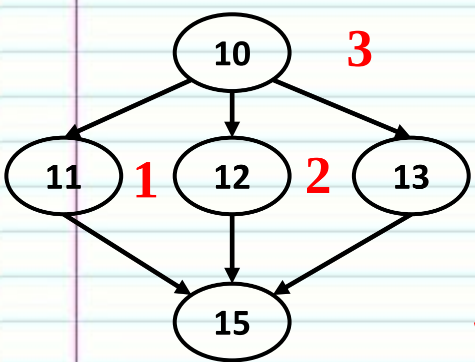
```
public boolean checkOut (double cart, int creditRating, statusClass.Status status) {
      boolean approved=false;

      if (status==statusClass.Status.gold) {
          if (cart < 3_500.00)
              approved = true;
          else
              if (creditRating > 650)
                  approved = true; }
      else {
          if (status==statusClass.Status.silver) {
              if (cart < 2_500.00)
                  approved = true;
              else
                  if (creditRating > 750)
                      approved = true; }
          else {
                  if (cart < 1_500.00)
                      approved = true;
                  else
                      if (creditRating > 800)
                          approved = true; }}
      return approved; }
```

From M09 - what would this look like as a multiple condition decision statement?

Achieving Condition Coverage (Example 6 - cont)

```
8  public boolean checkOut (double cart, int creditRating) {
9      boolean approved=false;
10         switch (status) {
11             case gold: approved = (cart < 3_500.00) || (creditRating > 650); break;
12             case silver: approved = (cart < 2_500.00) || (creditRating > 750); break;
13             case regular: approved = (cart < 1_500.00) || (creditRating > 800);
14         }
15         return approved;
16 }
```



Basis path set:

1) 10-11-15
2) 10-12-15
3) 10-13-15

We solve each statement 11, 12, and 13 for MCDC as we did in M09 - we end up with the same 9 test cases

Achieving Condition Coverage (Example 6 - cont)

An alternative implementation to the previous slide

```
public boolean checkOut (double cart, int creditRating) {
  return    (status==Status.gold)    && ((cart < 3_500.00) || (creditRating > 650)) ||
            (status==Status.silver)  && ((cart < 2_500.00) || (creditRating > 750)) ||
            (status==Status.regular) && ((cart < 1_500.00) || (creditRating > 800));
}
```

9

This is the CFG - we don't have much to work with

Cyclomatic complexity is 2 (the return statement is a decision)

What is wrong with this approach?

1) The MCDC tests for this would be horrible - we have 3 strongly coupled conditions and a total of 15 test cases (ignoring extreme range)
2) This is not as efficient as the previous slide - the status checks have to be performed 3 times versus once as in the previous slides

This demonstrates the importance of testable code - it helps knowing how to test the various structures and which is more efficient and testable

Short Circuiting and Code Coverage

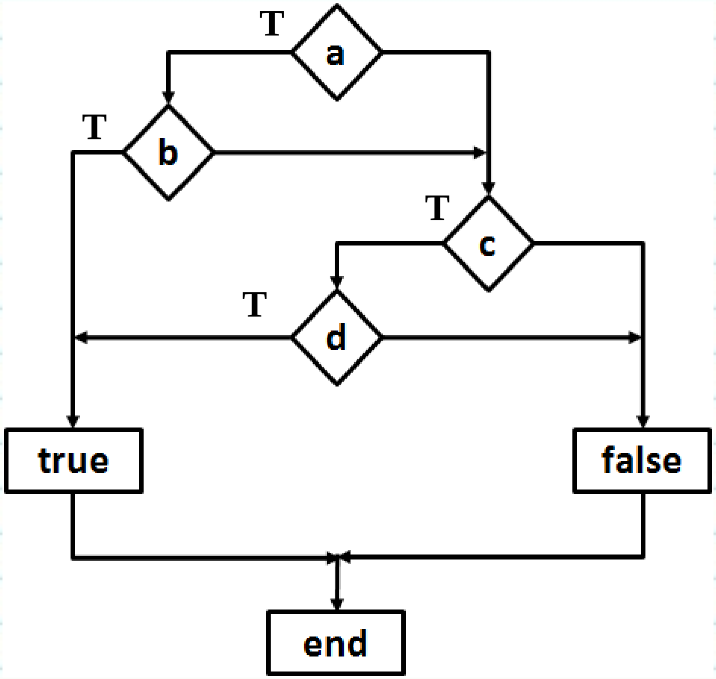
- Now that we are dealing with code we have to address an issue that is built into most languages (e.g., Java,C++, Ada) - short circuiting
- For the expression, if (a || b || c || d) - if a is true, then no further testing of conditions is required since or's are sensitive to true, similarly for b and c
- For the expression, if (a && b && c && d) - if a is false, then no further testing of conditions is required since and's are sensitive to false , similarly for b and c
- So when designing test cases this can become a problem. When we run code coverage tools on well-designed tests we find that the desired coverage was not achieved because of short-circuiting
- The easiest solution is to change the operator from a short-circuiting operator to a "bit-wise" operator. In Java and C this is from && to & and || to |
- The use of these can be defined in programming standards and automatically checked via code checkers (e.g., Static Code Analyzers)

Multiple Condition Coverage

- Multiple condition coverage has two flavors
  - Without with short-circuiting is simply the entire truth table
  - With short circuiting - we derive tests by taking the entire truth table and then applying short circuiting to the logical expression
- For the expression a & b & c & d, we get the following cases for MCC/SC -> TTTT, F---,TF--,TTF-,TTTF
  - This is the authors solution for the Gregorian calendar method from the "How We Test Software at Microsoft" book
  - Notice how, for this example, we **can** get the same answer as MC/DC
  - The MCC/SC solution set for a&b|c&d is: F-F-,F-TF,--TT,TFF-,TFTF,TT--
  - This **can** get an MC/DC solution, but also can get **much lower coverage**.
- MC/DC is far superior because it provides a **criteria** for test case design - stimulating one COI at a time to show its affect on the outcome

Byte Code Short-Circuiting Coverage Only

- If we map the Java byte code of the expression ab + cd we get:



- The four terms, TFTT, FFFF,TTTT,FFTF give complete short-circuit coverage
- These terms produce low fault coverage (40%)
- Short-circuiting coverage only can provide quite poor fault detection capabilities

Byte Code Short-Circuiting Coverage Only (cont.)

- Contrast this with the published paper - “Reasonability of MC/DC for Safety-Relevant Software Implemented in Programming Languages with Short-Circuit Evaluation”
- Authors conclude “We conclude with the strong recommendation to use MCC as a coverage metric for testing safety-relevant software implemented in programming languages with short-circuit evaluation.”
- We just saw that MCC with short-circuiting can only detect 40% of SFFs (Single Fault Failures) - why did the authors conclude this?
- They studied a single set of code (21,100 lines) with “24 decisions with a different number of conditions”
- They didn’t look at the SFFs that we have seen - these represent typical failures - this goes way beyond sampling a small code base

More Code Coverage - Example 1

```
public boolean Logic1Class (boolean a, boolean b, boolean c, boolean d) {
    return (a&&b || c&&d);
}
```

1. Test Input Set 1 = (a=true , b=true , c=false , d=true)
2. Test Input Set 2 = (a=true , b=false , c=false , d=true)
3. Test Input Set 3 = (a=false , b=true , c=false , d=true)
4. Test Input Set 4 = (a=false , b=false , c=false , d=false)

- Select the highest level of coverage (None, Statement, Decision, MCDC) achieved given the following combinations
- PROBLEMS

1. Set 1 + Set 2 = None, Statement, Decision, MCDC
2. Set 2 + Set 3 = None, Statement, Decision, MCDC
3. Set 1 + Set 3 = None, Statement, Decision, MCDC

Indicate your answers

More Code Coverage - Example 1

```
public boolean Logic1Class (boolean a, boolean b, boolean c, boolean d) {
    return (a&&b || c&&d);
}
```

1. Test Input Set 1 = (a=true , b=true , c=false , d=true)
2. Test Input Set 2 = (a=true , b=false , c=false , d=true)
3. Test Input Set 3 = (a=false , b=true , c=false , d=true)
4. Test Input Set 4 = (a=false , b=false , c=false , d=false)

- Select the highest level of coverage (None, Statement, Decision, MCDC) achieved given the following combinations
- PROBLEMS

1. Set 1 + Set 2 = None, Statement, **Decision**, MCDC
2. Set 2 + Set 3 = **None**, Statement, Decision, MCDC
3. Set 1 + Set 3 = None, Statement, **Decision**, MCDC

More Code Coverage - Example 2

```
public int returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
 int x=0;
 if (conditiona)
   x++;
 if (conditionb)
   x++;
 if (conditionc)
   x++;
 return x;
}
```

1. Test Input Set 1 = (a=false , b=false , c=false)
2. Test Input Set 2 = (a=true , b=false , c=false)
3. Test Input Set 3 = (a=false , b=true , c=false)
4. Test Input Set 4 = (a=true , b=true , c=true)

- Select the highest level of coverage (None, Statement, Decision, MCDC) achieved given the following combinations
- PROBLEMS

1. Set 1 + Set 2 = None, Statement, Decision, MCDC
2. Set 2 + Set 3 = None, Statement, Decision, MCDC
3. Set 1 + Set 4 = None, Statement, Decision, MCDC
4. Set 4 = None, Statement, Decision, MCDC

Indicate your answers

More Code Coverage - Example 2

```
public int returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
 int x=0;
 if (conditiona)
   x++;
 if (conditionb)
   x++;
 if (conditionc)
   x++;
 return x;
}
```

1. Test Input Set 1 = (a=false , b=false , c=false)
2. Test Input Set 2 = (a=true , b=false , c=false)
3. Test Input Set 3 = (a=false , b=true , c=false)
4. Test Input Set 4 = (a=true , b=true , c=true)

- Select the highest level of coverage (None, Statement, Decision, MCDC) achieved given the following combinations
- PROBLEMS

1. Set 1 + Set 2 = **None**, Statement, Decision, MCDC
2. Set 2 + Set 3 = **None**, Statement, Decision, MCDC
3. Set 1 + Set 4 = None, Statement, **Decision**, MCDC
4. Set 4 = None, **Statement**, Decision, MCDC

Indicate your answers

Loops

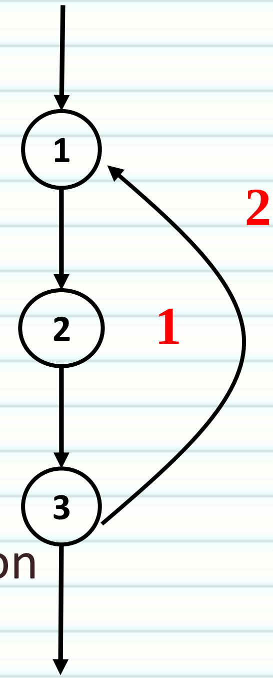
- For Java there are basically three kinds of looping structures
  1. Do while (post-condition loop)
  2. For (pre-condition loop)
  3. While (pre-condition loop)

- There are many different rules about how to test loops, this course will select the most common approach but be aware that other approaches exist (so check with your governing standards)

- There are three basic categories of loops
  – Simple (non-nested)
  – Nested
  – Concatenated (series of simple loops)

Looping Structures

```
1    do {
2         i++;
3    } while (i != 5);
```



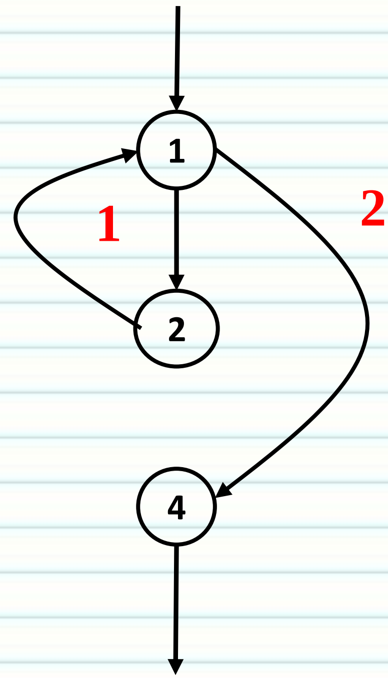
- The CFG is shown on the right

- It is a post-condition loop meaning that is always takes at least one iteration through the loop

- Cyclomatic complexity is 2, but this doesn't mean anything for loops because we have special techniques that need to be applied

Looping Structures (cont.)

```
while (i < args.length) {
        System.out.println(args[i]);
        i++;
}
```



- The CFG is shown on the right

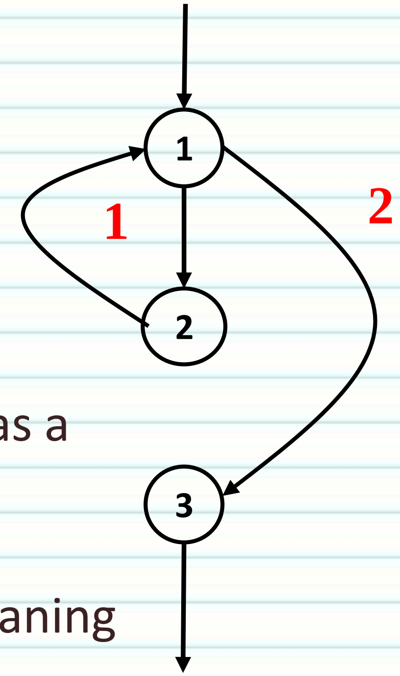
- It is a pre-condition loop meaning that is can take zero or more iterations through the loop

- Cyclomatic complexity is also 2

Looping Structures (cont.)

```
for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);

```



- The CFG is shown on the right

- It is a the same as a while except that it has a built in iterator.

- The for loop is also pre-condition loop meaning that is can take zero or more iterations through the loop

- Java and C++ have a special variant of this loop used to iterate through elements in a collection. This has the same CFG as above

- Cyclomatic complexity is also 2

Loop Failure Taxonomy

- The typical kind of loop errors that we are going to see fall in the off-by-one software error, these result in:
  1. Too few writes
  2. Too many writes
  3. Too few reads
  4. Too many reads
- Only one of the above will result in an exception being raised!
- How do these occur - they are distinct for each type of loop
- We'll examine each next

For Loop Failure Taxonomy

- The following source code shows us the most common problems with for loops:

```
for (int i = 0; i < args.length; i++)
        System.out.println(args[i]);

```

Initialized to the wrong value (ZBN)

1) Loop body - intervals vs. items and vice-versa
2) Forgetting that the loop may execute 0 times and later data may not have a valid value to use

Loop termination (logic) incorrect - off-by-one error in logic.
e.g. < vs. <=

While Loop Failure Taxonomy

- The following source code shows us the most common problems with while loops:

```
while (i < args.length) {
        System.out.println(args[i]);
        i++;
}
```

Loop iterator incremented before use

Loop termination (logic) incorrect - off-by-one error in logic. e.g. < vs. <=

1) Loop body - intervals vs. items and vice-versa
2) Forgetting that the loop may execute 0 times and later data may not have a valid value to use

Do While Loop Failure Taxonomy

- The following source code shows us the most common problems with while loops:

```
do {
    i++;
} while (i != 5);
```

Loop iterator incremented before use

Loop termination (logic) incorrect - off-by-one error in logic. e.g. < vs. <=

1) Loop body - intervals vs. items and vice-versa
2) Remembering that the loop will always execute once

Loop Testing Strategies

- For simple loops, the following is the typical approach (you need to know both approaches)
  - Try to bypass the loop (pre-condition loops only)
  - Execute the loop body once, twice, typical number
  - I use once only (my claim is that twice and typical are in the same equivalence class)
  - If the loop has an upper bound then execute the loop body n-1, n, n+1 times (if possible)

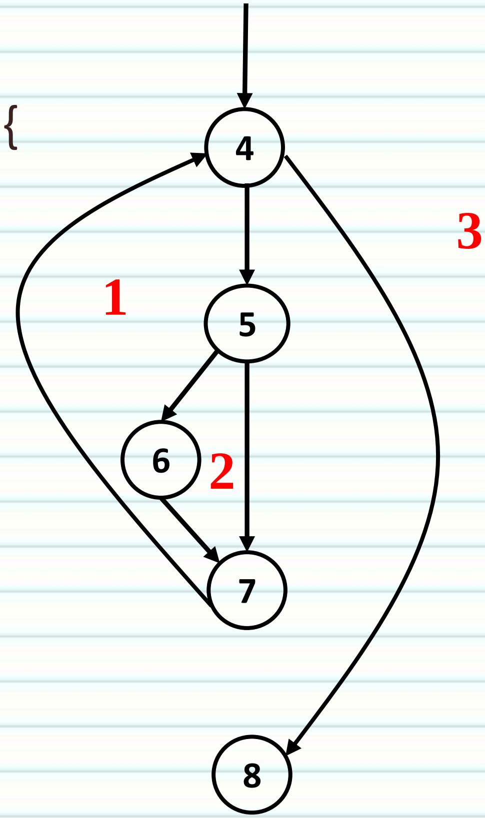
- For nested loops conduct the previous tests for the simple loop on the inner loop, work outward holding
  - Outer loops at the minimal non-zero times through the loop
  - Inner loops are kept at a typical values
  - Repeat these steps until all loops are tested
  - This approach keeps the number of tests linear to the nesting level

- Concatenated loops (independent loops) are tested as independent simple loops

Is There a Different Way?

- As an alternative can I apply some testing principles to the looping construct to test it?

```
1   public static int numNegs (int [ ] arr) {
2
3     int count = 0;
4     for (int i = 0; i <= arr.length-1; i++)
5       if (arr [ i ] < 0)
6         count++;
7
8     return count;
9   }
```



- What is this doing?

Is There a Different Way? (cont.)

- So if we look at the code it is testing the number of negative values in an array.

  {0,-1,-4,-5} - result is 3
- Are there any equivalence classes or partitions that we can come up with?
  - Position (first, middle, last)
  - Number (zero, one, multiple)
- Test cases could be
  - {0, 1, 2, 3} (zero)
  - {-1, 0, 1, 2} (first),(one)
  - {0, 1, 2, -1} (last),(one)
  - {0,-1,-2, 0} (multiple), (middle)
- These test cases conditions could be adopted for the array being any size larger than 4
- Student exercise: perform these tests and provide the actual outputs from them - does it work?

Is There a Different Way? (cont.)

- Outputs are
  - {0, 1, 2, 3}	0
  - {-1, 0, 1, 2}	1
  - {0, 1, 2, -1}	1
  - {0,-1,-2, 0}	2
- All tests pass and the function works (despite the odd logic)

- This approach provides a better assurance that the algorithm actually works - it maps back to the function or even possible the specification

- The mechanical approach provides a means to perform this when it is difficult to develop equivalence classes for the algorithm

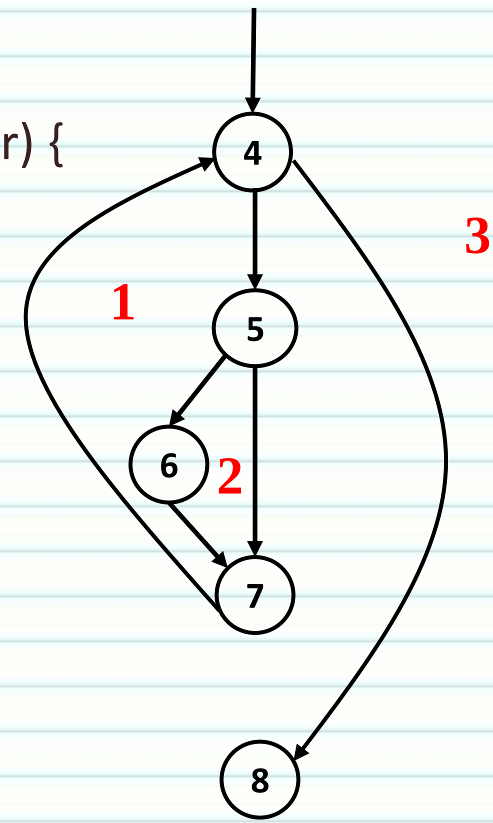
Student Exercise

- Student exercise - check the following code with the test cases below does it work?

```
public static int numZeroes (int [ ] arr) {

  int count = 0;
  for (int i = 0; i <= arr.length; i++)
    if (arr [ i ] == 0)
      count++;

  return count;
}
```



  - {-1, 1, 2, 3}
  - {0, 1, 2, 3}
  - {1, 2, 3, 0}
  - {1, 0, 0, 3}

Student Exercise (cont.)

Answers

– {-1, 1, 2, 3}	0
– {0, 1, 2, 3}	0
– {1, 2, 3, 0}	1
– {1, 0, 0, 3}	2

- The test failed - there is an error in the code

Common Sense Still Applies

```
private enum Day
{
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
  public static void workingWeek(Day today)
  {
    switch (today)
    {
      case MONDAY:
      case TUESDAY:
      case WEDNESDAY:
      case THURSDAY:  System.out.println("Workday");
                      break;
      case FRIDAY:    System.out.println("Last workday!");
                      break;
      case SATURDAY:
      case SUNDAY:    System.out.println("Weekend!");
                      break;
       default: } }

public static void main(String[] args) {
            for (Day d:Day.values())
                        workingWeek(d);
}
```

I can only execute the loop through the entire collection, so I augment my loop testing strategy accordingly. So I test this as a switch as before and not as a loop.

Loops With Multiple Conditions

- We apply conditions to loops consistent with our previous treatment of conditions

```
while (i < args.length) & (!eof) {
        System.out.println(args[i]);
        i++;
}
```

- We will test the loop condition at a condition level using MCDC test data and perform all the required loop tests

Summary of Path Testing

- We have developed methods for testing various source code constructs
  - Logical expressions
  - Mathematical expressions
  - Switch statements
  - Loops

- We have explored the various levels of coverage
  - Statement
  - Decision
  - Branch
  - Condition
  - Boundary value

Summary of Path Testing (cont.)

- To develop comprehensive tests we need to be aware of the weaknesses of the testing approaches that we have explored
  - Basis path testing is a very powerful technique used industry wide to develop unit level test cases - it is very effective, however it does have some weaknesses that we have compensated for, we added
    1. complete BV coverage and extreme range
    2. multiple condition logical expressions
    3. loop coverage
    4. switch statements
       a) use of the default condition in the switch
       b) fully test common ECPs (common ECPs/case statements)
    5. guidance on what inputs to use/not use for mathematical operations
- The section of the course has dealt with test case design from code - but the most important point is to develop test cases from the **requirements**
  - **That's why the homework has required you to develop the description of the code (aka "requirements")**