Strategies of white-box testing to drive test case design

Barbara Russo

SwSE - Software and Systems Engineering Research Group

Strategies

- Code coverage
- Test Driven Development
- Control Flow Diagrams
  - Path coverage

Code coverage

Barbara Russo

SwSE - Software and Systems Engineering research group

Coverage

How thoroughly a test suite exercises a program

Example

```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
   return e;
}
```

bug(): if a=1 it returns true

and false if a!=1.

Coverage

- Coverage is a measure of the completeness of the set of test cases
  - Method coverage
  - Statement coverage
  - Branch coverage
  - Condition coverage

Method coverage

- Measure: *percentage of methods that have been executed at least once by test cases*

- Test cases should exercise 100% of the methods

- It is irresponsible to deliver non-tested methods
  - Testers need to ensure 100% method coverage

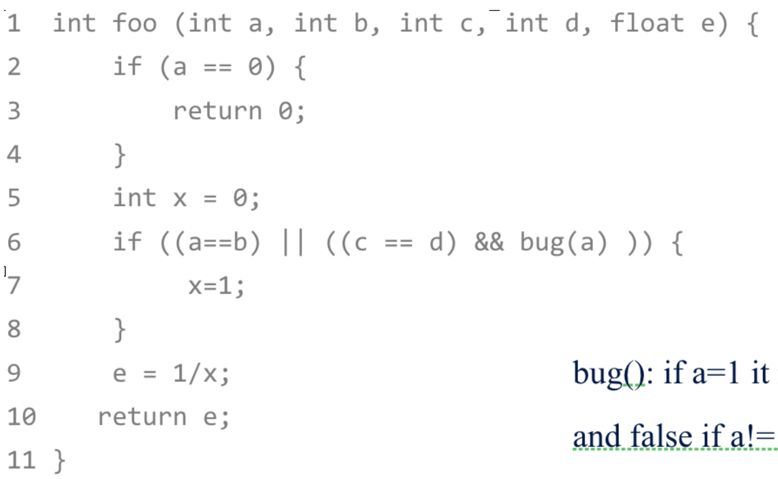
TC1

There is only one method

`int foo (int a, int b, int c, int d, float e)`

for a=0 `foo` returns 0 no matter the values of the other parameters

calling `foo` with input (0,0,0,0,0) we attain 100% method coverage in our example



```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
   return e;
}
```

bug(): if a=1 it returns true and false if a!=1.

Statement coverage

- Measure: *percentage of statements that have been executed by test cases*
  - Achieve 100% statement coverage: cover statements with test cases

Statement coverage

Check coverage of TC1 first: executed statements on lines 1-4 only out of 11 lines of code

Statement coverage: ~36% (4/11) with TC1

```
1  int foo (int a, int b, int c, int d, float e) {
2      if (a == 0) {
3          return 0;
4      }
5      int x = 0;
6      if ((a==b) || ((c == d) && bug(a) )) {
7           x=1;
8      }
9      e = 1/x;
10    return e;
11 }
```

bug(): if a=1 it returns true and false if a!=1.

We need another test case!

Reach the execution of line 5 -> a!=0

TC2

- TC2(1, 1, 1, 1,1), expected return value = 1.
- executes statements on lines 5-11

- **100% statement coverage obtained!**

Branch Coverage

- Measure: *percentage of the decision points evaluated as both true and false in test cases*
  - Achieve 100% branch coverage: cover all branches as both true and false with test cases

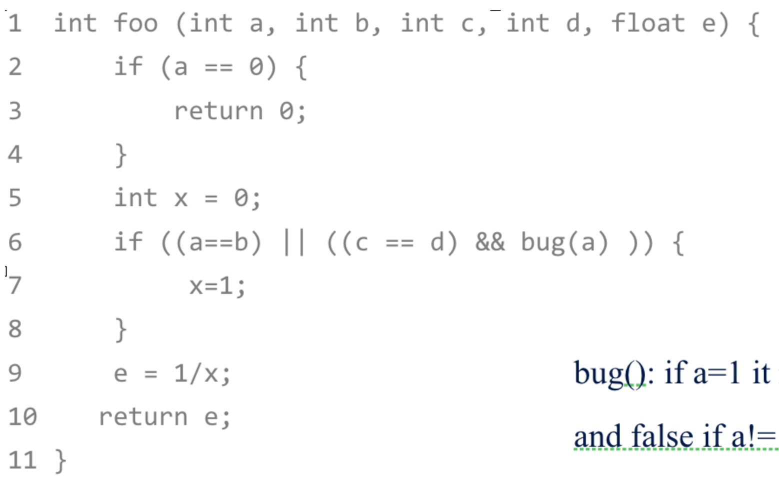
Branch Coverage

Two decision points:

one at line 2 and the other at line 6

```
if (a == 0) {}
```

```
if ((a==b) OR ((c == d) AND bug(a))) {}
```



```
1  int foo (int a, int b, int c, int d, float e) {
2      if (a == 0) {
3          return 0;
4      }
5      int x = 0;
6      if ((a==b) || ((c == d) && bug(a) )) {
7           x=1;
8      }
9      e = 1/x;
10    return e;
11 }
```

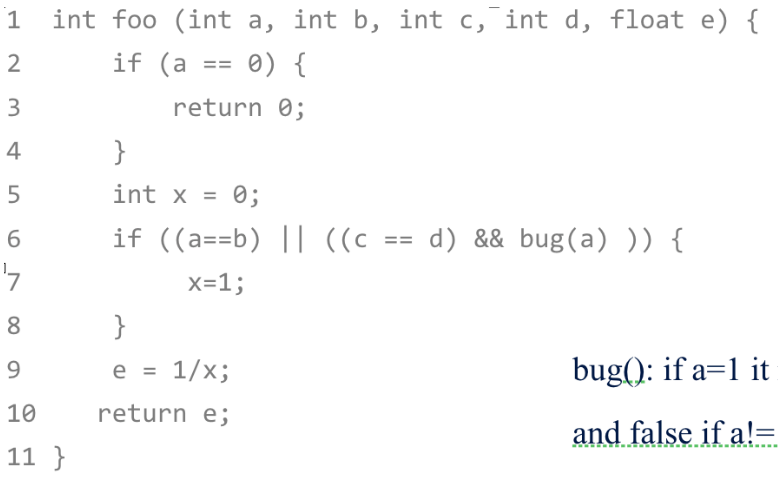
bug(): if a=1 it returns true and false if a!=1.

Branch Coverage

- For decision/branch coverage, we evaluate an entire Boolean expression of the condition as one true-or-false predicate

Branch Coverage

Line #	Predicate	True	False
3	(a == 0)	TC1(0, 0, 0, 0, 0) return 0	TC2(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	TC2(1, 1, 1, 1, 1) return 1	



```
1  int foo (int a, int b, int c, int d, float e) {
2      if (a == 0) {
3          return 0;
4      }
5      int x = 0;
6      if ((a==b) || ((c == d) && bug(a) )) {
7           x=1;
8      }
9      e = 1/x;
10    return e;
11 }
```

bug(): if a=1 it returns true

and false if a!=1.

TC3

- With TC1 and TC2 we have executed three of the four necessary conditions
  - 75% branch coverage so far

We need another test case!

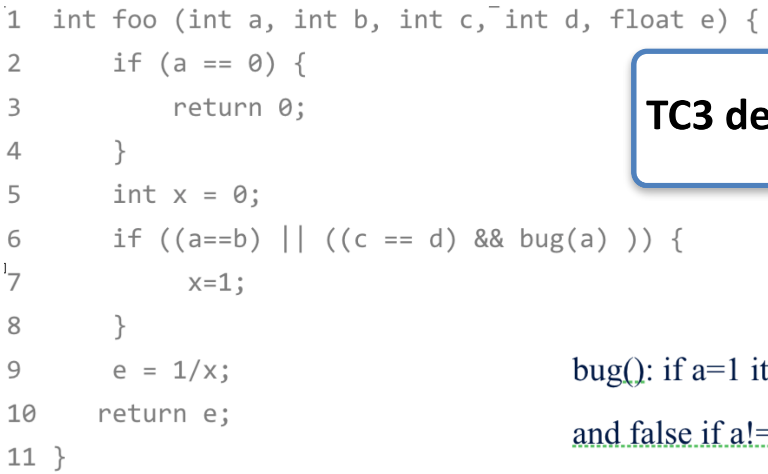
TC3

- TC3(1, 2, 1, 2, 1) return ??
- **Division by 0** that can cause future failures!
  - That was due to a local variable that we could not control by using strategies based on the analysis of the input space of `foo()`!
  - It depends on how we implemented the method

Branch coverage

!((a==b) OR ((c == d) AND bug(a))) = (a!=b) AND ((c != d) OR !bug(a)))

Line #	Predicate	True	False
3	(a == 0)	TC1(0, 0, 0, 0, 0) return 0	TC2(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	TC2(1, 1, 1, 1, 1) return 1	TC3(1,2,1,2,1) Division by zero



```
1  int foo (int a, int b, int c, int d, float e) {
2      if (a == 0) {
3          return 0;
4      }
5      int x = 0;
6      if ((a==b) || ((c == d) && bug(a) )) {
7           x=1;
8      }
9      e = 1/x;
10    return e;
11 }
```

TC3 defined for both (a!=b) AND (c != d)

bug(): if a=1 it returns true

and false if a!=1.

Condition Coverage

- Measure: *percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome in test cases*

- Condition coverage measures the outcome of each of these sub-expressions independently of each other

Condition Coverage

We need another TC!

Predicate	True	False
(a == 0)	TC1(0, 0, 0, 0, 0) return 0	TC2(1, 1, 1, 1, 1) return 1
(a == b)	TC2(1, 1, 1, 1, 1) return 1	TC3(1,2,1,2,1) Division by zero
(c == d)		TC3(1,2,1,2,1) Division by zero
bug(a)		

To reach the execution of (c==d) must be a!=b and a!=0

```
int foo (int a, int b, i
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
  return e;
}
```

bug(): if a=1 it returns true

and false if a!=1.

TC4

for a==1 bug(a) = TRUE and return 1, otherwise division by zero. It does not matter for cond. cov whether to enter the if-block!

Predicate	True	False
(a == 0)	TC1(0, 0, 0, 0, 0) return 0	TC2(1, 1, 1, 1, 1) return 1
(a == b)	TC2(1, 1, 1, 1, 1) return 1	TC3(1,2,1,2,1) Division by zero
(c == d)	TC4(1,2,1,1,1) return 1	TC3(1,2,1,2,1) Division by zero
bug(a)		

```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
  return e;
}
```

bug(): if a=1 it returns true and false if a!=1.

TC4

We need another TC!

Predicate	True	False
(a == 0)	TC1(0, 0, 0, 0, 0) return 0	TC2(1, 1, , 1, 1) return 1
(a == b)	TC2(1, 1, 1, 1, 1) return 1	TC3(1,2,1 2,1) Division by zero
(c == d)	TC4(1,2,1,1,1) return 1	TC3(1,2,1 2,1) Division by zero
bug(a)	TC4(1,2,1,1,1) return 1	

```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
   return e;
}
```

bug(): if a=1 it returns true

and false if a!=1.

TC5

a!=b but a!=1 ->
change only a

Predicate	True	False
(a == 0)	TC1(0,0,0,0,0) return 0	TC2(1,1,1,1,1) return 1
(a == b)	TC2(1,1,1,1,1) return 1	TC3(1,2,1,2,1) Division by zero
(c == d)	TC4(1,2,1,1,1) return 1	TC3(1,2,1,2,1) Division by zero
bug(a)	TC4(1,2,1,1,1) return 1	TC5(3,2,1,1,1) Division by zero

Again, c == d or c!=d
changes only the return value

```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ((a==b) || ((c == d) && bug(a) )) {
         x=1;
    }
    e = 1/x;
   return e;
}
```

bug(): if a=1 it returns true
and false if a!=1.

Note

- Condition coverage does not imply branch coverage!
- Predicate: `A && B - e.g.: a=b && c=d`

Condition	Branch	Example
TF, FT	**F,F**	**T(1,1,0,1) and T(1,0,1,1)**
TT, FF	**T,F**	**T(1,1,1,1) and T(1,0,1,0)**

- **Condition coverage does not subsumes branch coverage!**

but I can build a test suite for condition coverage that contains a test suite for branch coverage

Traceability matrix

Predicate	True	False
(a == 0)	TC1(0,0,0,0,0) return 0	TC2(1,1,1,1,1) return 1
(a == b)	TC2(1,1,1,1,1) return 1	TC3(1,2,1,2,1) Division by zero
(c == d)	TC4(1,2,1,1,1) return 1	TC3(1,2,1,2,1) Division by zero
bug(a)	TC4(1,2,1,1,1) return 1	TC5(3,2,1,1,1) Division by zero

JaCoCo

- It is an Eclipse plug-in
- With Maven: In the node build and sub-node plugins of the POM file include

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.2</version>
  <executions>
     <execution>
       <goals>
          <goal>prepare-agent</goal>
       </goals>
      </execution>
<!-- attached to Maven test phase -->
     <execution>
          <id>report</id>
          <phase>test</phase>
          <goals>
               <goal>report</goal>
          </goals>
     </execution>
  </executions>
</plugin>
```

Exercise

```
public class Hailstone {
  public static void main(String[] args) {
    int n = 3;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

- Run this class with JaCoCo code coverage highlighting turned on, by choosing Run → Coverage As → Java Application.
- By changing the initial value of n, you can observe how JaCoCo highlights different lines of code differently.

Exercise

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

🟢 **Executed**

🔴 **Not Executed**

🟡 **Partially Executed Branch**

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

🟢 **Executed**

🔴 **Not Executed**

🟡 **Partially Executed Branch**

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

n
3
10
5
16
8
4
2
1

🟢 **Executed**

🔴 **Not Executed**

🟡 **Partially Executed Branch**

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

🟢 **Executed**

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

🟢 **Executed**

🔴 **Not Executed**

🟡 **Partially Executed Branch**

n
3
10
5
16
8
4
2
1

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

🟢 Executed

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

n
3
10
5
16
8
4
2
1

🟢 Executed

🔴 Not Executed

🟡 Partially Executed Branch

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

🟢 Executed

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

🔴 Not Executed

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

🟢 Executed

🔴 Not Executed

🟡 Partially Executed Branch

n
3
10
5
16
8
4
2
1

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

○ **Executed**

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

○ **Not Executed**

What initial value of n would make the line while (n != 1) yellow after execution?

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

○ **Executed**

○ **Not Executed**

○ **Partially Executed Branch**

n
3
10
5
16
8
4
2
1

Exercise

When n=3 initially, what color is the line n = n/2 after execution?

○ Executed

When n=16 initially, what color is the line n = 3 * n + 1 after execution?

○ Not Executed

What initial value of n would make the line while (n != 1) yellow after execution?

○ Partially Executed Branch

n=1

```
public class Hailstone {
  public static void main(String[] args) {
    int n = ?;
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
  }
}
```

○ Executed

○ Not Executed

○ Partially Executed Branch

n
3
10
5
16
8
4
2
1

Testing

- Testing is a **dynamic activity**
- It can be done only when the artefacts to be tested are “executable”

Testing as a development technique

- Move forward testing to the earliest possible is one of the practices of agile methods:
  - **Test First in XP**
- Testing has been also used to develop new code:
  - **Test Driven Development**

Test Driven Development (TDD)

- Practice for writing unit tests and production code *concurrently and at a very fine level of granularity*
- Programmers
  - **first write a small portion of a unit test**, and
  - then they **write just enough production code** to make that unit test compile and execute

Test Driven Development (TDD)

- This cycle lasts somewhere between **30 seconds and five minutes**. Rarely does it grow to ten minutes.
- Once a unit test is done, the developer goes on to the next test until they run out of tests for the task they are currently working on

Test Driven Development (TDD)

- Use compilation and execution to drive development

Example - TDD in Java

- Specification:

  TextFormatter: it takes arbitrary strings and horizontally centers them in a line

- Methods:

  a. *setLineWidth()*

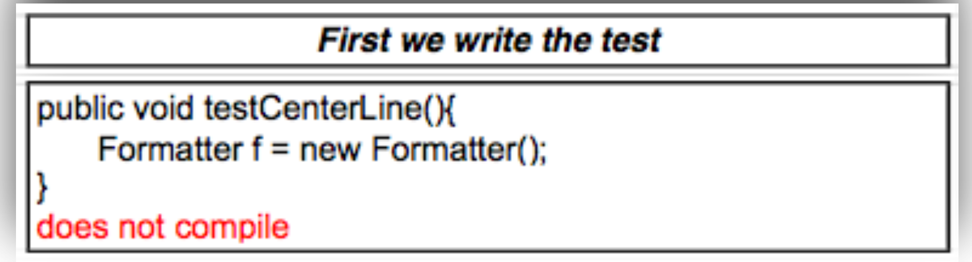
  b. *center()*

- Parameters

  a. **size** b. **string**

Example - TDD in Java

- Start by creating a test method and instantiate within it an object of the class you want to test



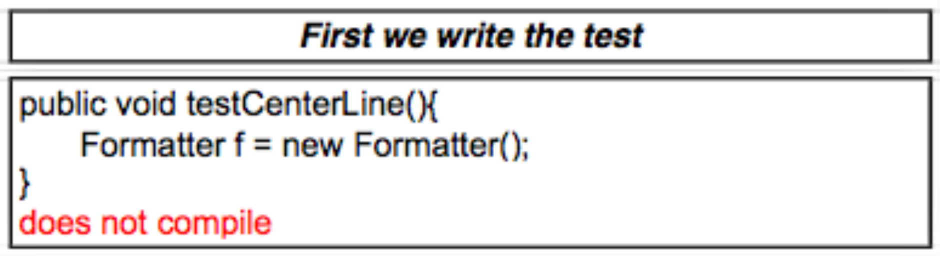
First we write the test

```
public void testCenterLine(){
      Formatter f = new Formatter();
}
```

does not compile

Example - TDD in Java

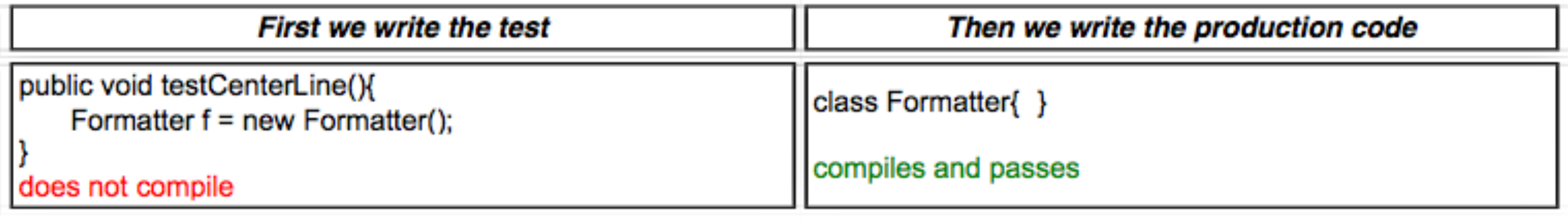
- Start by creating a test method and instantiate within it an object of the class you want to test



First we write the test

```
public void testCenterLine(){
      Formatter f = new Formatter();
}
```

does not compile



First we write the test

```
public void testCenterLine(){
      Formatter f = new Formatter();
}
```

does not compile

Then we write the production code

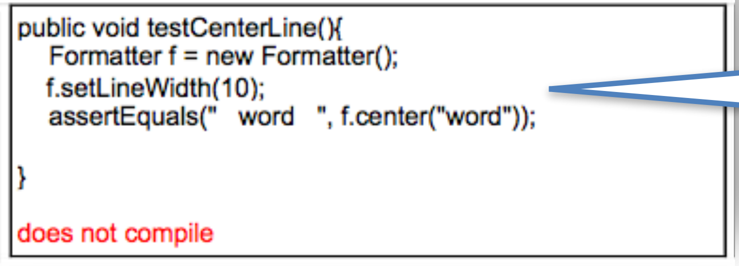
```
class Formatter{  }
```

compiles and passes

Example - TDD in Java

- Select one method to develop
- Choose input representative values to test it

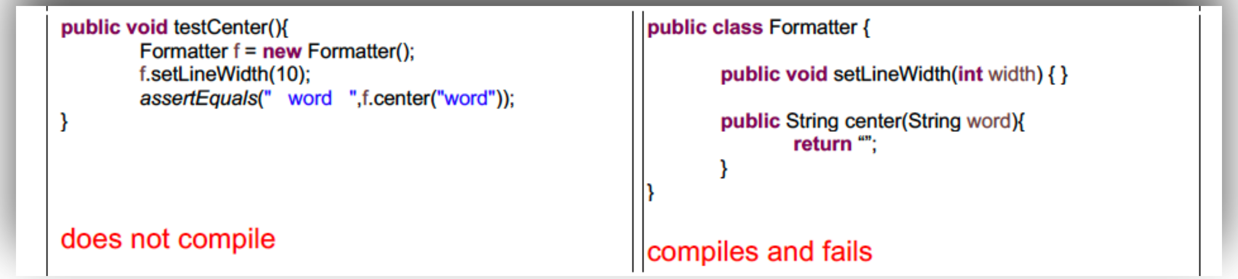
center



```
public void testCenterLine(){
    Formatter f = new Formatter();
    f.setLineWidth(10);
    assertEquals("   word   ", f.center("word"));

}
```

does not compile

width=10
line="word"



```
public void testCenter(){
        Formatter f = new Formatter();
        f.setLineWidth(10);
        assertEquals("   word   ",f.center("word"));
}
```

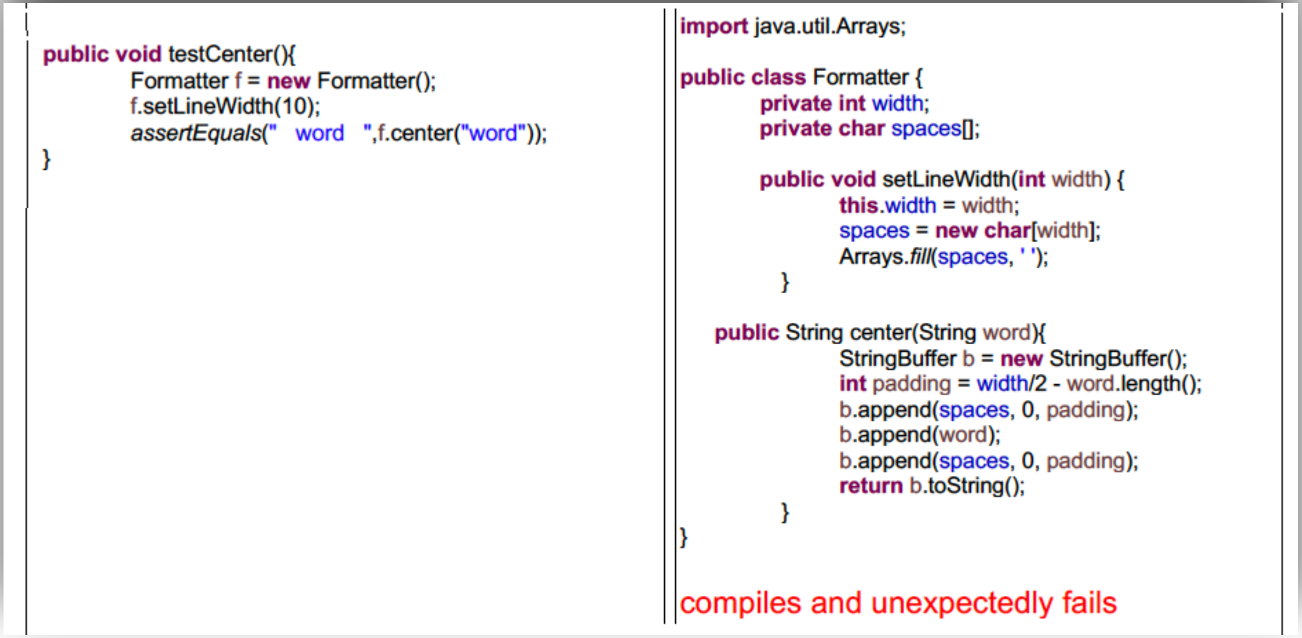
does not compile

```
public class Formatter {

        public void setLineWidth(int width) { }

        public String center(String word){
                return “”;
        }
}
```

compiles and fails

padding | term | padding

- Develop the method in a simple way to avoid it fails; thinking of the parameters’ values you have chosen; first attempt



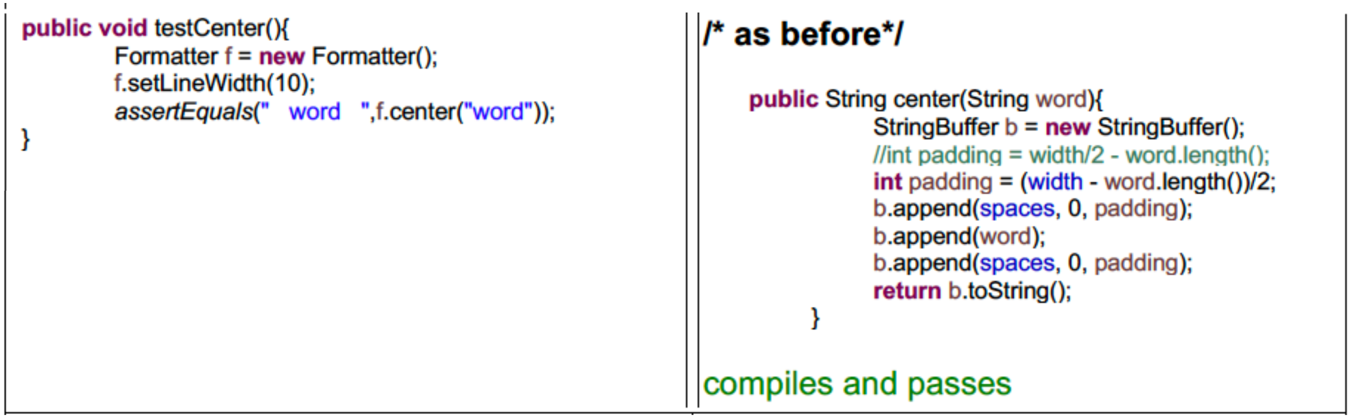
```
public void testCenter(){
        Formatter f = new Formatter();
        f.setLineWidth(10);
        assertEquals("   word   ",f.center("word"));
}
```

```
import java.util.Arrays;

public class Formatter {
        private int width;
        private char spaces[];

        public void setLineWidth(int width) {
                this.width = width;
                spaces = new char[width];
                Arrays.fill(spaces, ' ');
        }

    public String center(String word){
                StringBuffer b = new StringBuffer();
                int padding = width/2 - word.length();
                b.append(spaces, 0, padding);
                b.append(word);
                b.append(spaces, 0, padding);
                return b.toString();
        }
}
```

compiles and unexpectedly fails

Example - TDD in Java

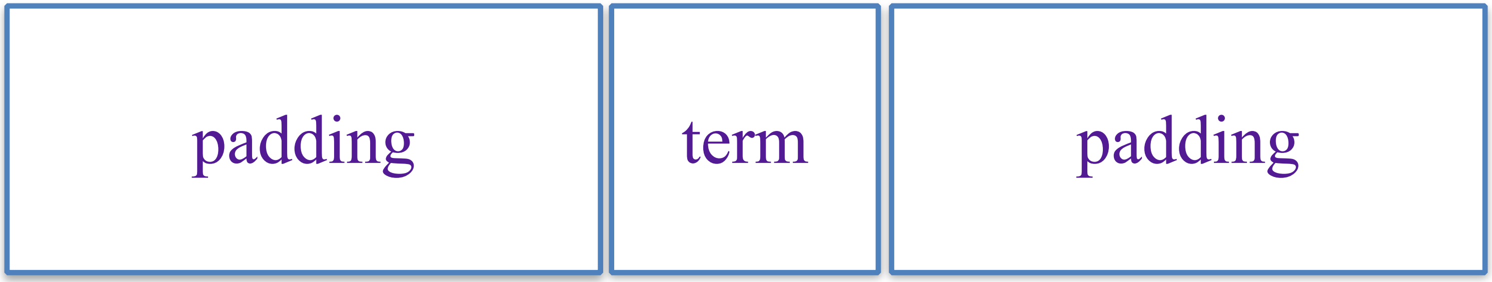
- Re-thinking of the logic



```
public void testCenter(){
        Formatter f = new Formatter();
        f.setLineWidth(10);
        assertEquals("   word   ",f.center("word"));
}
```

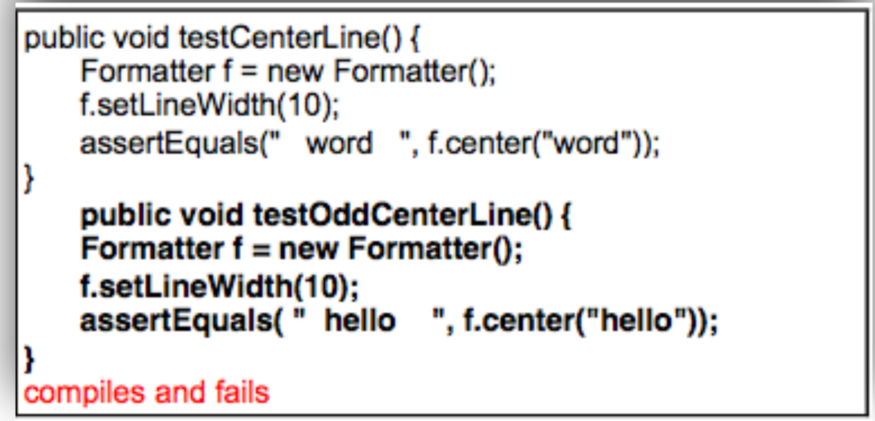
```
/* as before*/

    public String center(String word){
            StringBuffer b = new StringBuffer();
            //int padding = width/2 - word.length();
            int padding = (width - word.length())/2;
            b.append(spaces, 0, padding);
            b.append(word);
            b.append(spaces, 0, padding);
            return b.toString();
    }
```

compiles and passes



Example - TDD in Java

- Changed parameter value into “**hello**”



```
public void testCenterLine() {
	Formatter f = new Formatter();
	f.setLineWidth(10);
	assertEquals("   word   ", f.center("word"));
}
	public void testOddCenterLine() {
	Formatter f = new Formatter();
	f.setLineWidth(10);
	assertEquals( "  hello   ", f.center("hello"));
}
compiles and fails
```

Exercise

- How many test cases?
- Let’s reason using category partition testing!

How many tests w. brute force?

width	term.length
odd	odd
even	even

How many tests w. brute force?

- One-parameter problem!

width	term.length
odd	odd
even	even

- padding = (width-word.length)/2

How many tests w. brute force?

- One-parameter problem!

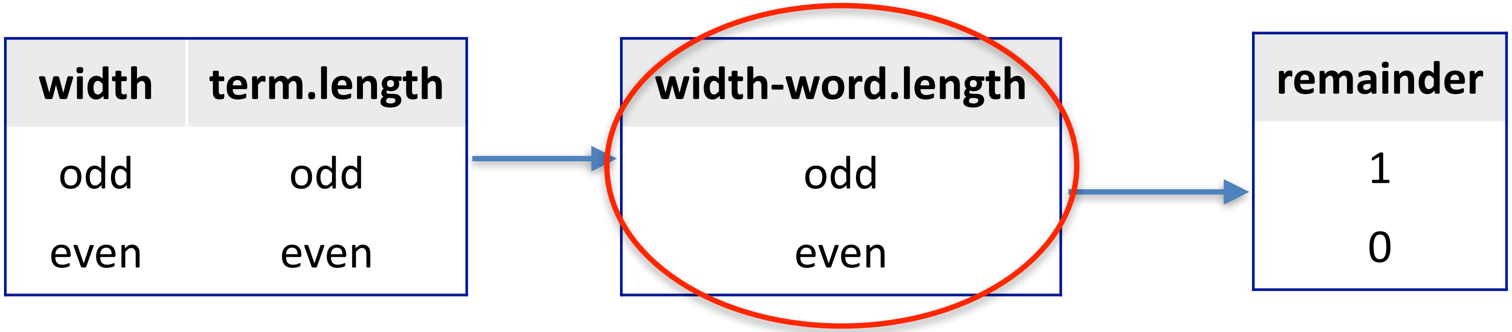
width	term.length
odd	odd
even	even

width-word.length
odd
even

- padding = (width-word.length)/2

How many tests w. brute force?

- One-parameter problem!



width	term.length
odd	odd
even	even

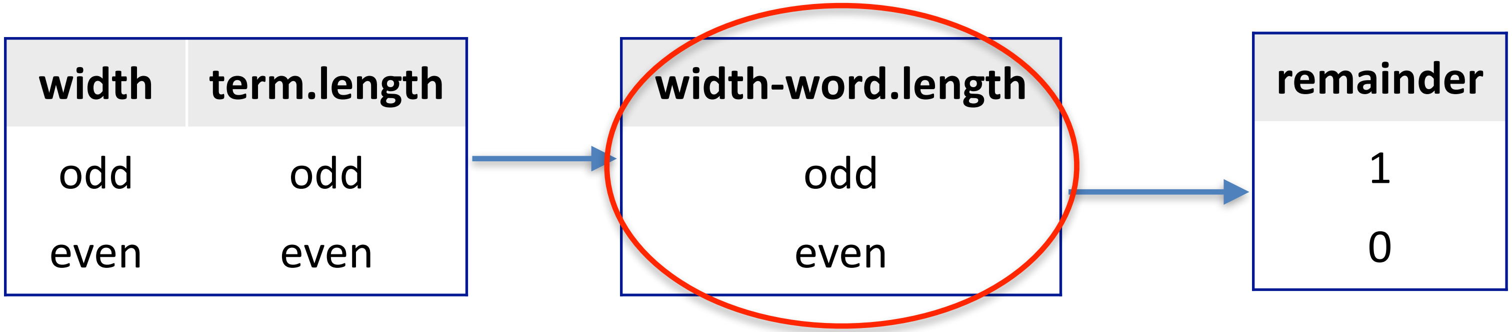
width-word.length
odd
even

remainder
1
0

- padding = (width-word.length)/2
- remainder = padding % 2

How many tests w. brute force?

- One-parameter problem!



width	term.length
odd	odd
even	even

width-word.length
odd
even

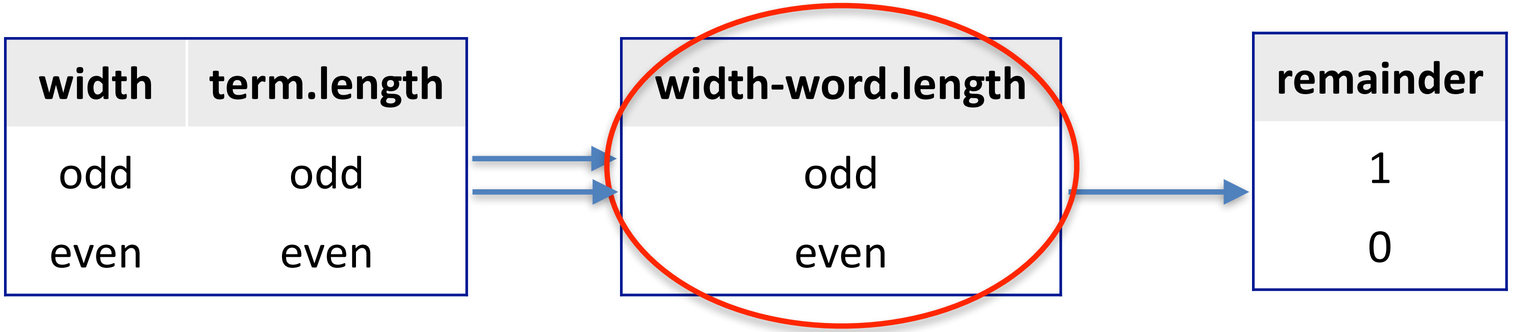
remainder
1
0

- padding = (width-word.length)/2
- remainder = padding % 2

padding	term	padding	remainder

How many tests w. brute force?

- One-parameter problem!



width	term.length
odd	odd
even	even

width-word.length
odd
even

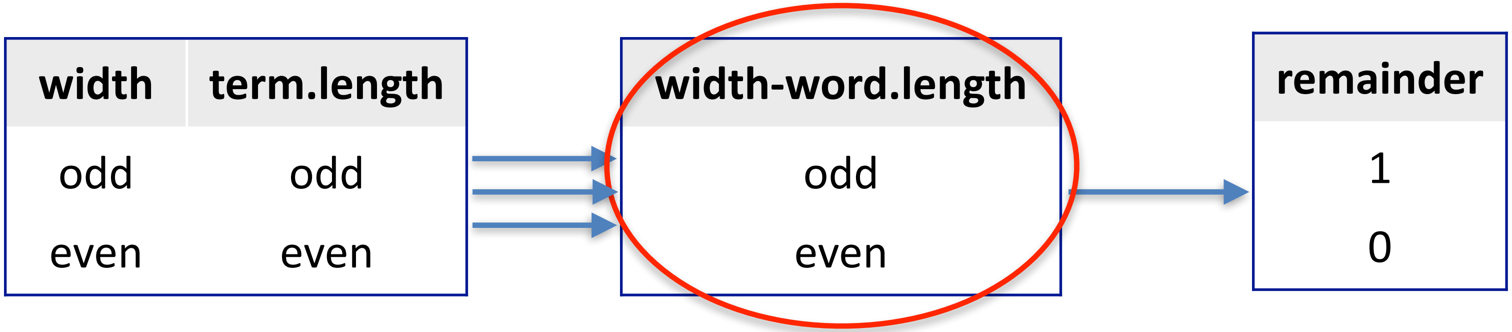
remainder
1
0

- padding = (width-word.length)/2
- remainder = padding % 2

padding	term	padding	remainder

How many tests w. brute force?

- One-parameter problem!



width	term.length
odd	odd
even	even

width-word.length
odd
even

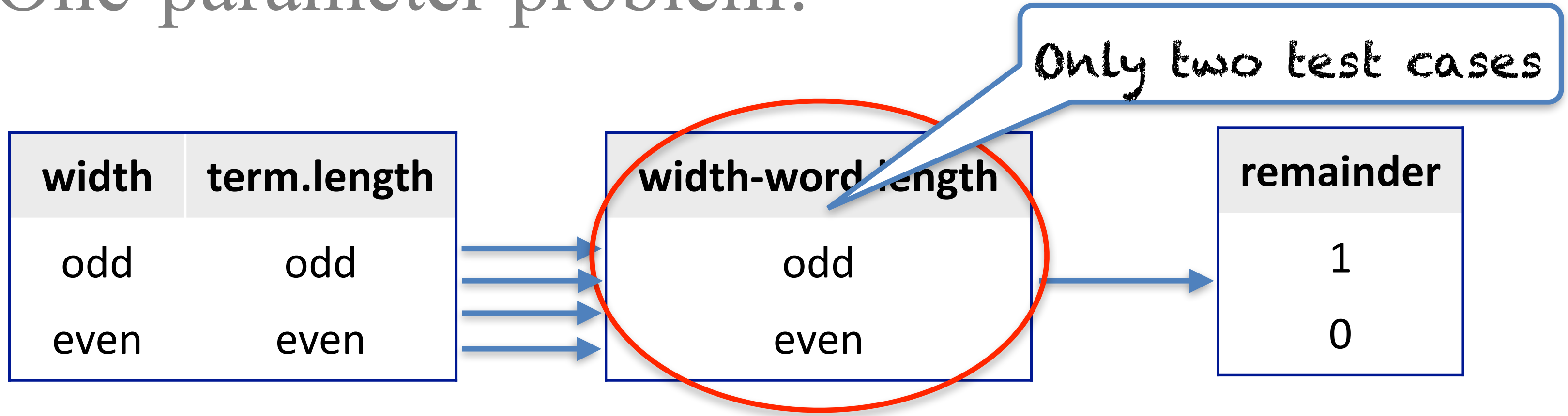
remainder
1
0

- padding = (width-word.length)/2
- remainder = padding % 2

padding	term	padding	remainder

How many tests w. brute force?

- One-parameter problem!



Only two test cases

width	term.length
odd	odd
even	even

width-word.length
odd
even

remainder
1
0

- padding = (width-word.length)/2
- remainder = padding % 2

padding	term	padding	remainder

Test cases

- **width-string.length**
- odd = 2k+1, even =2k with k>0
- k != r
  - odd - odd = 2(k-r) : even
  - even - even = 2(k-r) : even
  - even - odd or odd - even = 2(k-r) +/- 1 : odd
- **In addition**
  - padding=0 or
  - string.length=width=0
  - padding<0

Test cases

- **width-string.length**
- odd = 2k+1, even =2k with k>0
- k != r
  - odd - odd = 2(k-r) : even
  - even - even = 2(k-r) : even
  - even - odd or odd - even = 2(k-r) +/- 1 : odd

Only two test cases

- **In addition**
  - padding=0 or
  - string.length=width=0
  - padding<0

other default test cases

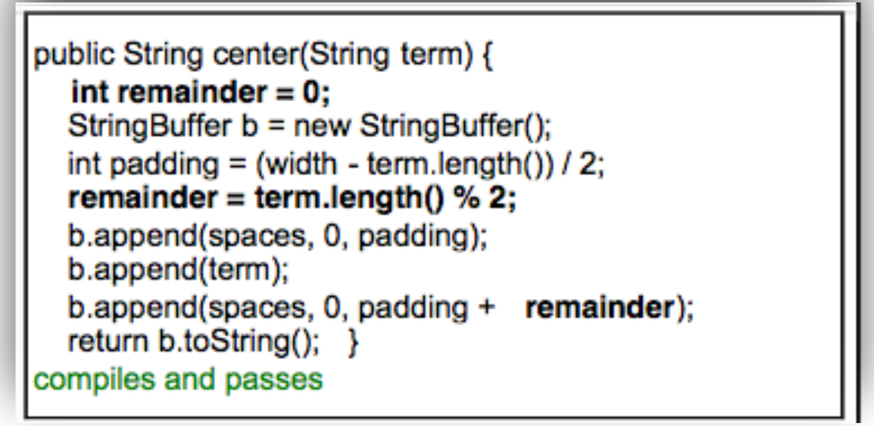
Combinatorial partition testing

width	word.length
even [Property: evenL]	even [Property: evenS] if([evenL])
~~odd [Property: oddL]~~	odd [Property: oddS] if([evenL])
0[single]	>word.length if([evenL]) [error]
	=word.length if([evenL])
	<word.length if([evenL])
	0 [single]

- line = 10
- string = “word” and “hello”
- string= “circumstances”
- string = “challenges”
- width = 0
- word.length=0

width=0 is used with 0 for word.length; cannot be used anywhere else

Solution



```
public String center(String term) {
    int remainder = 0;
    StringBuffer b = new StringBuffer();
    int padding = (width - term.length()) / 2;
    remainder = term.length() % 2;
    b.append(spaces, 0, padding);
    b.append(term);
    b.append(spaces, 0, padding +   remainder);
    return b.toString();   }
```

compiles and passes

Models of program execution

Barbara Russo

SwSE - Software and Systems Engineering Research Group

A model of program execution

A model of program execution is a representation of a software execution simpler but that preserves some key attributes of it

- This representation will help to define a strategy for testing

State Space

- Representation of the program execution with a sequence of states and transitions
- **The state space is a set of possible states and transitions**
- For almost all programs, the state space is potentially infinite

Abstraction function

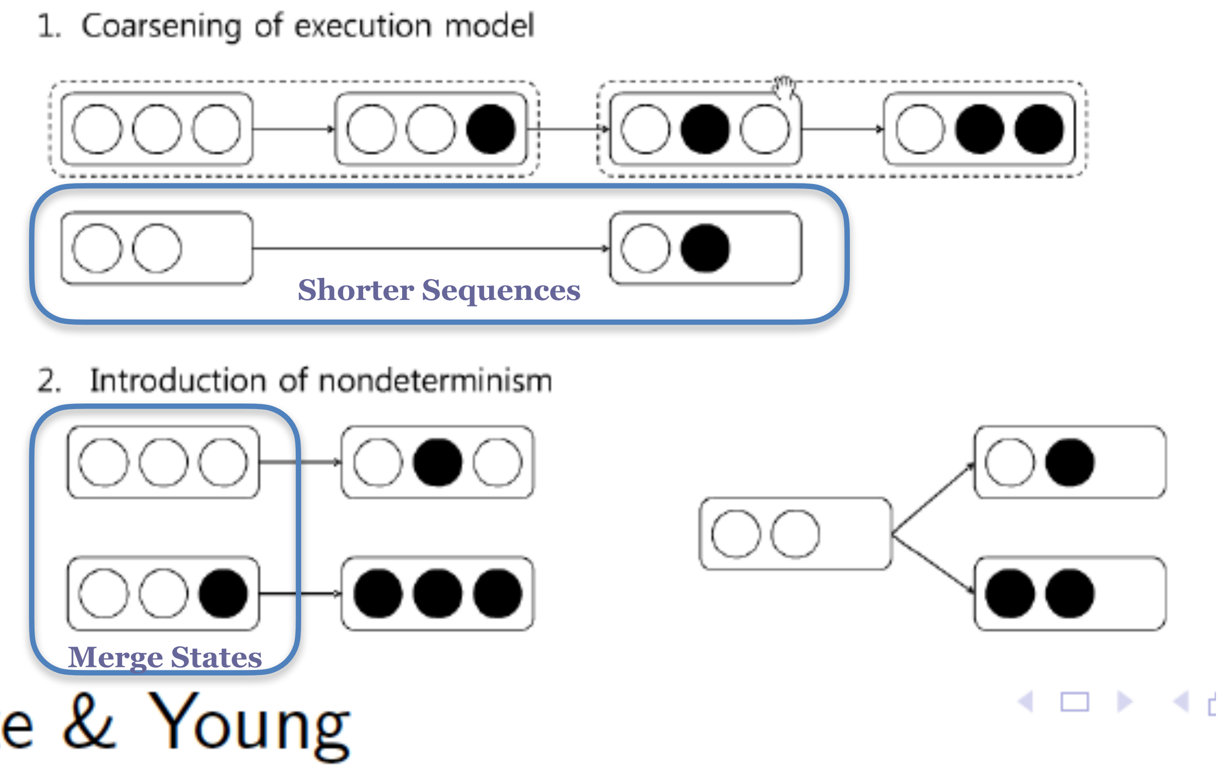
- The states are represented in the space by an abstraction function
- The abstraction function might suppress some states to create the finite model

Effects of abstraction

- **Coarsening**: execution sequences are collapsed into shorter sequences
- **Non determinism**: states are merged

Effects of abstraction

For example, assume the third state is neglected



1. Coarsening of execution model

2. Introduction of nondeterminism

Pezze & Young

Example: Control Flow Graphs

Barbara Russo

SwSE - Software and Systems Engineering Research Group

Control Flow Graphs

- It is a directed graph
  - **Node (state)**= portion of code
  - **Directed Edge** = flow of execution between two portions

Control flow structure

- The control flow structure is modeled with **direct graphs**
- A direct graph is a set of arcs and nodes with one defined direction
  - A set of statements without branch corresponds to a **node**, a flow of control from a statement to another to an arch
  - There is a **start node** and an **end node**
  - Each other node resides on a path between these two
  - Each node has an **in-degree** and an **out degree**
  - The start/end node has zero in - degree/out - degree

Control flow structure

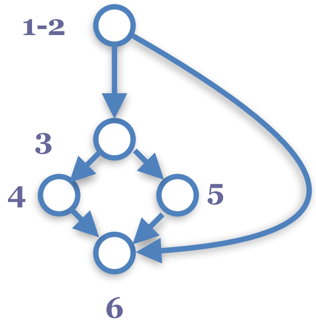
- A program is transformed in a direct graph called **control flow graph** that depicts the **execution control** of a program and the instruction to be executed
- It is a **static representation** of the program
- It makes visible the control structure
- Out-degree = 1 defines **procedural nodes** all the other nodes are called **predicate nodes**

Control Flow Graphs

- CFG keeps information of instructions to be executed and **ignores values of variables or data structures**
- Example of **non deterministic abstraction**

```
1 boolean z = FALSE;
2 if(z && y<=2){
3   if(z){
4     y++;
5   }else{y--;}
6 }
```

It does not depend on the value of z or the data structures in the branches



- CFG also models the non- feasible path!

CFG to design test cases

- We can use this information to design test cases
- Let’s see how to do it …
  - First let’s introduce the **McCabe complexity** measure which will help us to limit the number of test cases

McCabe Cyclomatic Complexity

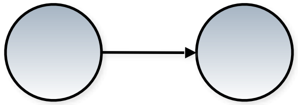
- Map codes to flow graphs
- Map flow graphs to numbers

CC definition

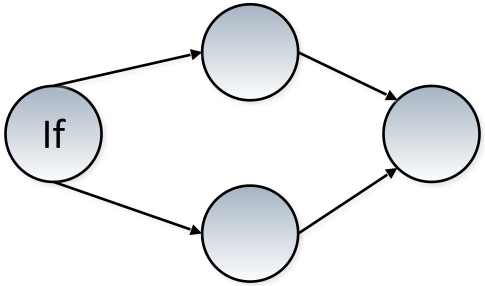
- CC= # of connected regions
- CC=# branches+1
- CC=# elements in a base
- CC=# decision point +1
- CC=#arcs-#nodes+2 (Euler characteristic)

Examples

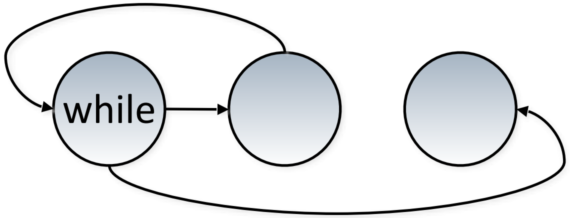
- Sequence



- If … then … else



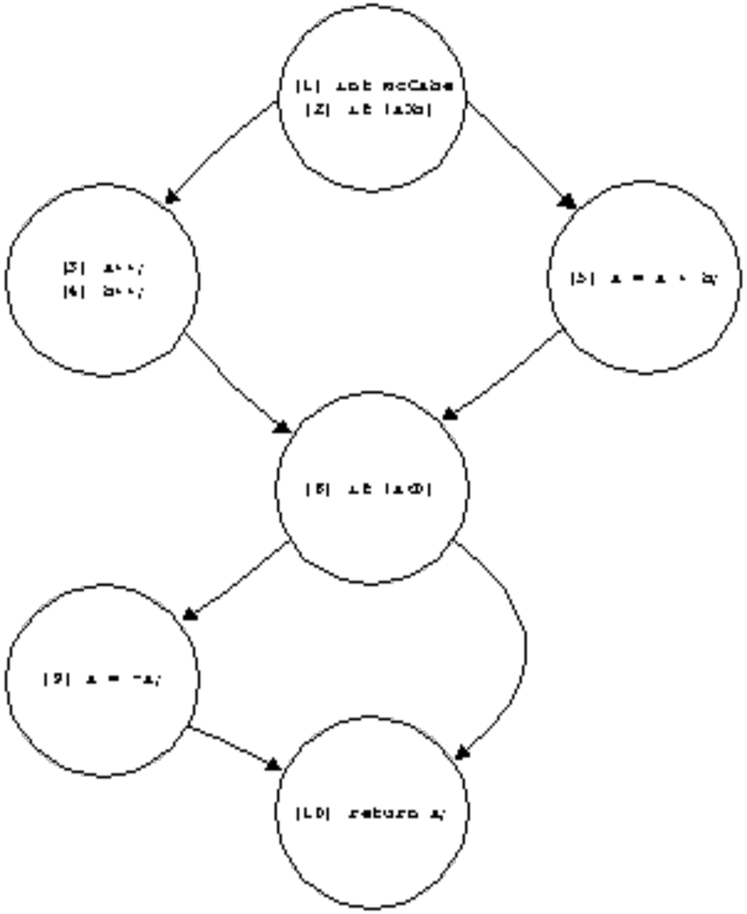
- While



Example

```
int mcCabe(int a, int b) {
  if (a >b) {
    a++;
    b--;
  } else {
    a=a + b;
  }
  if (a < 0) a=-a;
  return a;
}
```

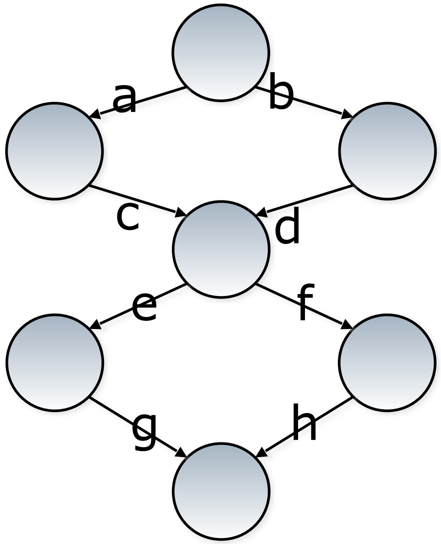
Exercise



CC as independent paths

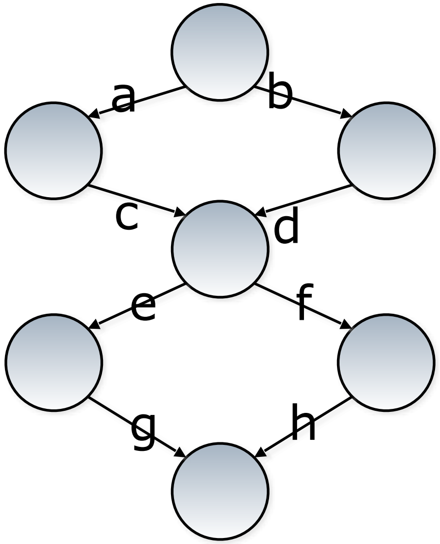
- A **complete path** is a path starting from the starting node and ending to the end node
- One complete path is **linearly independent** from the others if it does not exist a combination of the other complete paths to which is equal
- How to combine paths ...

Describe the base of the following graph



Rule to combine paths:
The arcs go from top to bottom
-a : is the arc in the opposite direction
-aceg: is the opposite complete path of aceg
ab: is first a and then b

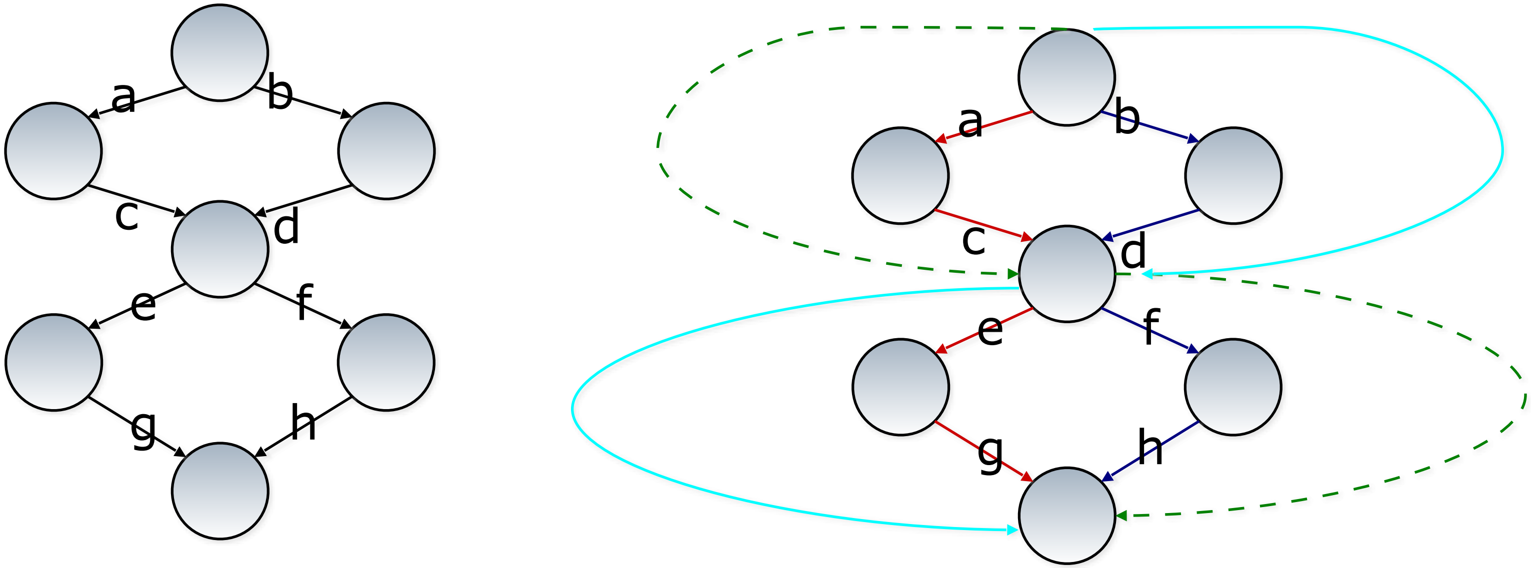
Describe the base of the following graph



Rule to combine paths:
The arcs go from top to bottom
-a : is the arc in the opposite direction
-aceg: is the opposite complete path of aceg
ab: is first a and then b

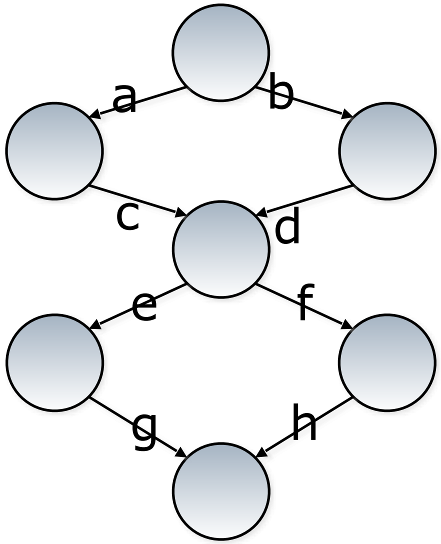
Complete paths:
aceg
bdfh
bdeg
acfh

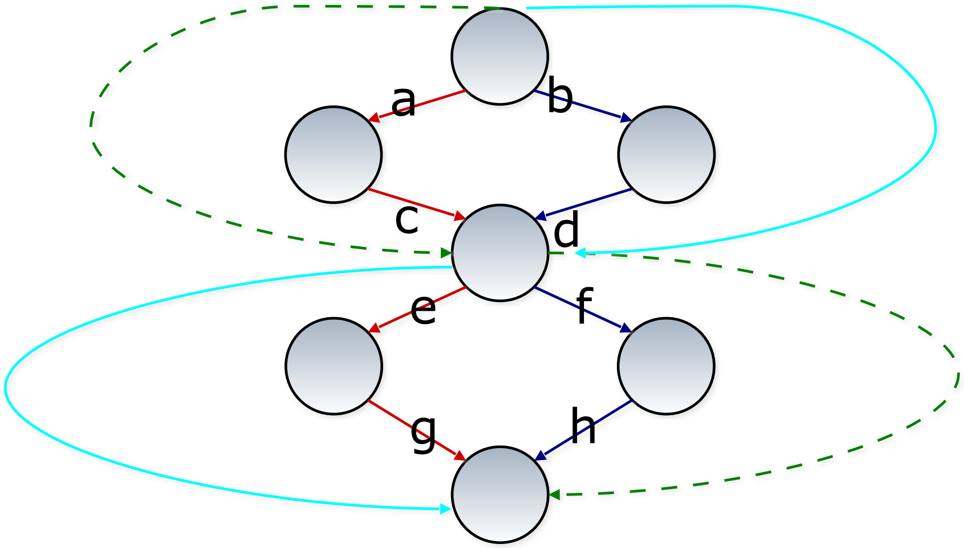
Describe the base of the following graph



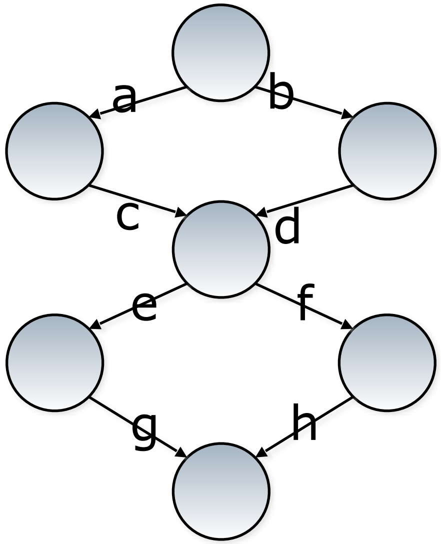
The path acfh=aceg-bdeg+bdfh

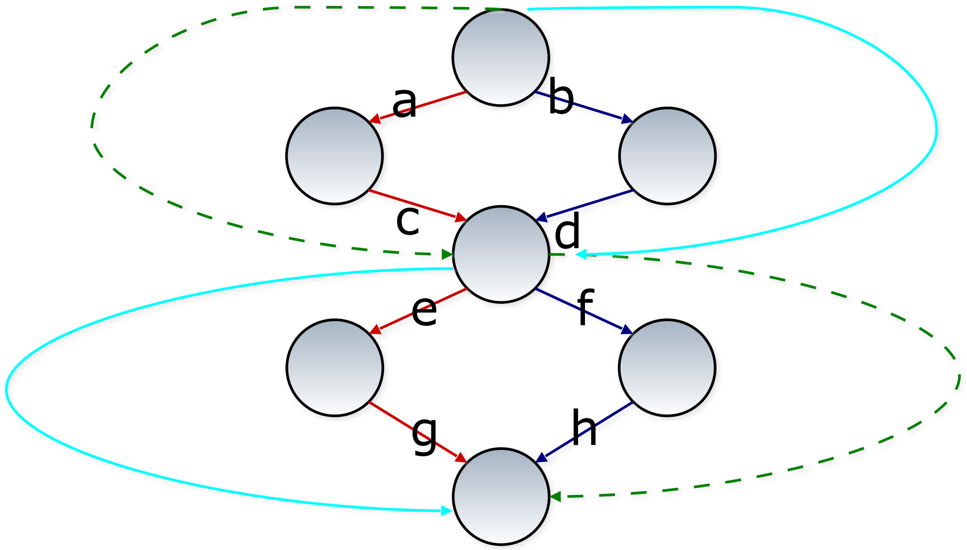
Describe the base of the following graph





Describe the base of the following graph



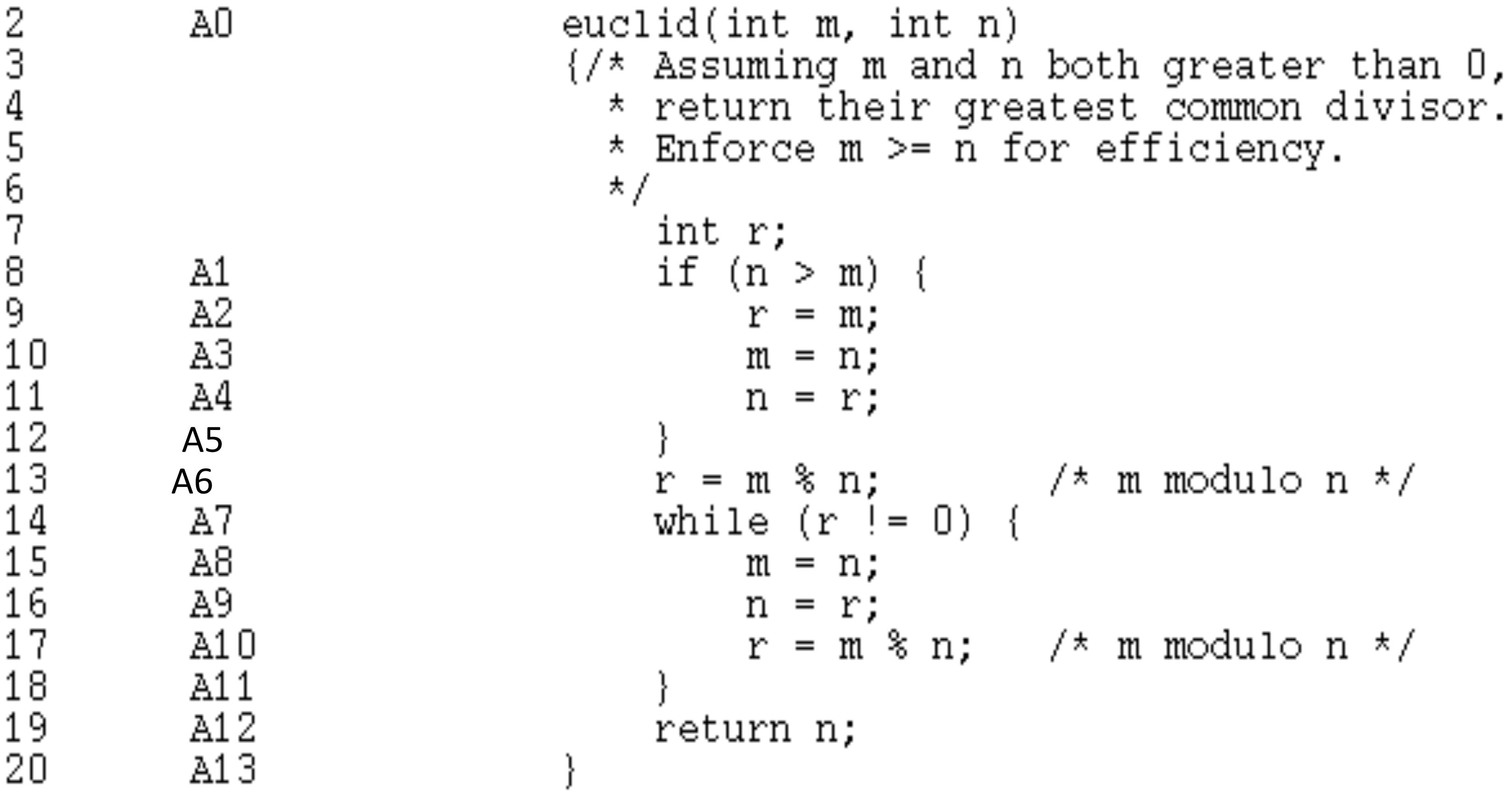


The path acfh=aceg-bdeg+bdfh

Base

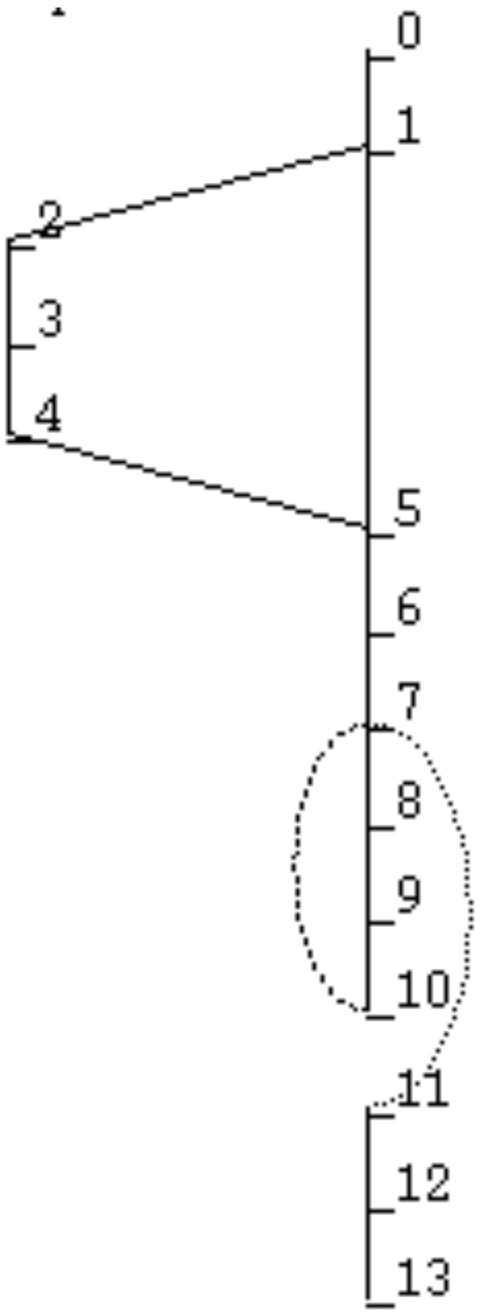
- Minimum number of independent complete paths

Draw the flow graph and Compute the CC



```
A0      euclid(int m, int n)
        {/* Assuming m and n both greater than 0,
           * return their greatest common divisor.
           * Enforce m >= n for efficiency.
           */
             int r;
A1           if (n > m) {
A2               r = m;
A3               m = n;
A4               n = r;
A5           }
A6           r = m % n;          /* m modulo n */
A7           while (r != 0) {
A8               m = n;
A9               n = r;
A10              r = m % n;     /* m modulo n */
A11          }
A12          return n;
A13     }
```

Result



Use CFG in testing

Barbara Russo

SwSE - Software and Systems Engineering Research Group

Path coverage

- Path coverage is every possible path through the program taken by some test case
- McCabe complexity is used to determine how many complete execution paths (i.e. test cases designed from them) a tester need to consider
- As with code coverage this is a measure that approximates completeness

Statement and Path coverage

- Reformulate statement coverage: Design test cases so that every node lies on at least one complete path
- Path coverage: Design test cases such that every possible arc is executed at least once

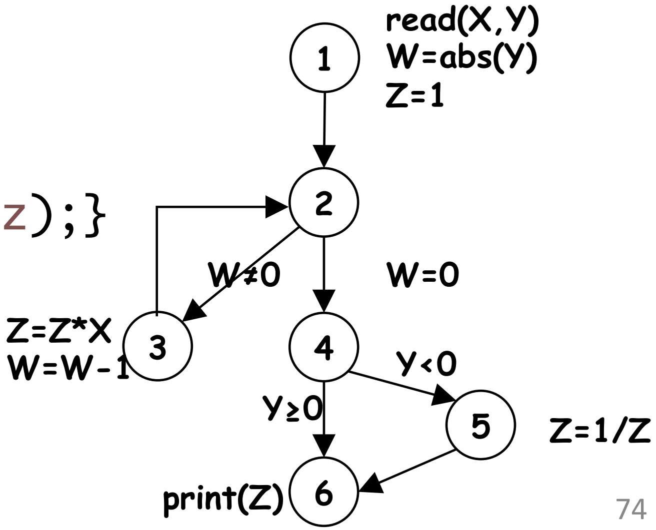
Template for path coverage

- Draw the CFG
- Count the possible independent complete paths
- Create a table with all the possible arcs as column headers
- Create a test case per execution of an arc in an independent path

The power function

Program computing Z=X^Y

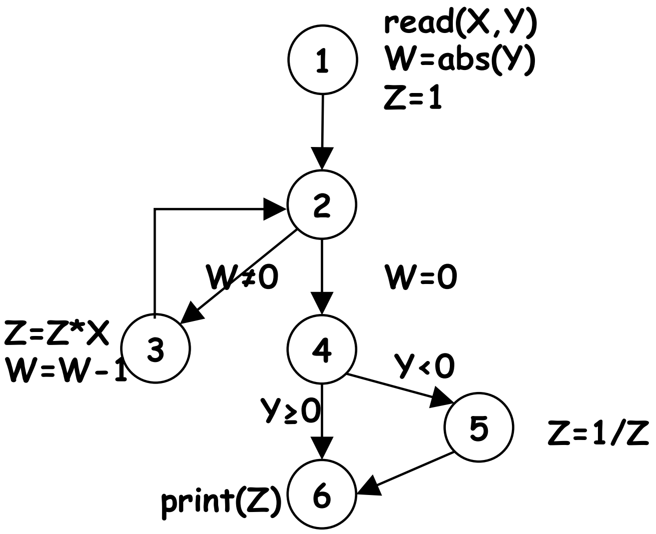
```
public class PowerFunction {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int w = Math.abs(y);
        int z = 1;

        while(w!=0){
            z=z*x;
            w=w-1;
        }
        if(y<0){z=1/z;}
        System.out.println("result is "+z);}
}
```



Path coverage

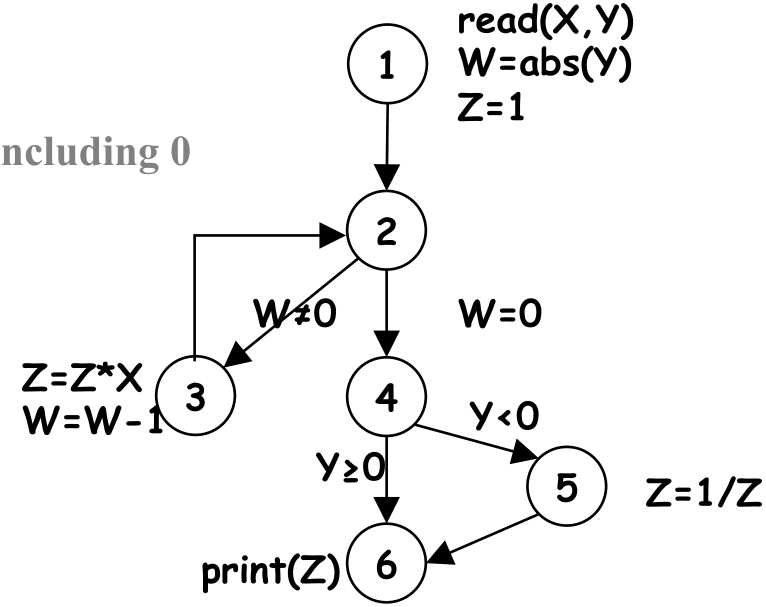
- All arcs are executed in at least one path
  - **Infeasible path**
  - 1 ->2 -> 4 -> 5-> 6
  - **As many ways to iterate as values of abs(Y) including 0**
  - 1 -> 2 -> (3 -> 2)* -> 4 -> 6
  - 1 -> 2 -> (3 -> 2)+ -> 4 -> 5 -> 6

  - w=0,1,-1 what for the infeasible path?



Issues

- Path coverage (CC=3, 4 complete paths)
  - **Infeasible path**
  - 1 ->2 -> 4 -> 5-> 6
  - **As many ways to iterate as values of abs(Y) including 0**
  - 1 -> 2 -> (3 -> 2)* -> 4 -> 6
  - 1 -> 2 -> (3 -> 2)+ -> 4 -> 5 -> 6
- Branch coverage
  - Three test cases:
    - **Y<0** : 1 -> 2 -> (3 ->2)+ -> 4 -> 5 -> 6
    - **Y≥0** : 1 -> 2 -> (3 -> 2)* -> 4 -> 6
- Statement coverage
  - One test case is enough:
    - **Y<0** : 1 -> 2 -> (3 ->2)+ ->4 -> 5 -> 6



Subsumption

- 100% path coverage subsumes both 100% statement coverage and branch coverage

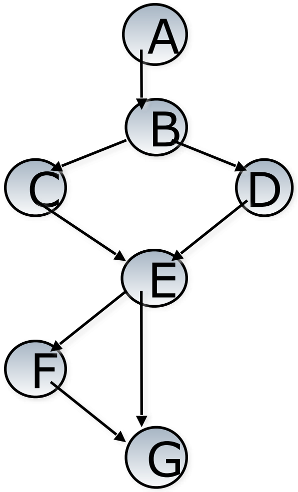
CFG and issues with coverage

- Some paths are infeasible
- Some edges are hidden
-

Some complete paths may be infeasible

- Infeasible path: a program path that cannot be executed for any input

```
A input(score)
B if score<45
C     then print ('fail')
D     else print ('pass')
E if score >70
F     then print ('with distinction')
G end
```

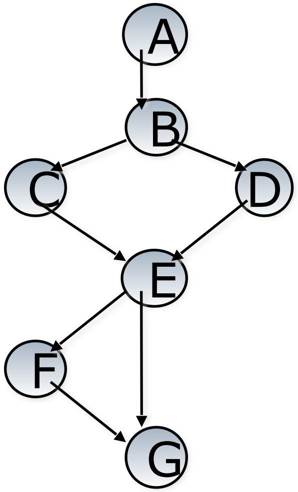


Some complete paths may be infeasible

- Infeasible path: a program path that cannot be executed for any input

```
A input(score)
B if score<45
C     then print ('fail')
D     else print ('pass')
E if score >70
F     then print ('with distinction')
G end
```

Which path is not feasible

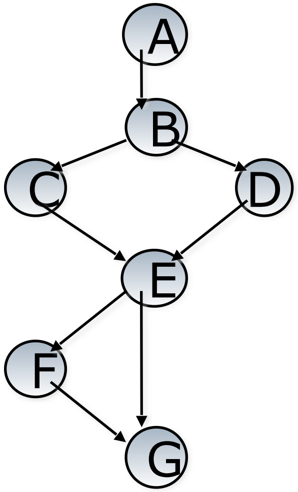


Some complete paths may be infeasible

- Infeasible path: a program path that cannot be executed for any input

```
A input(score)
B if score<45
C     then print ('fail')
D     else print ('pass')
E if score >70
F     then print ('with distinction')
G end
```

Which path is not feasible



- The path A-B-C-E-F-G is infeasible and
- It will be never executed
- We create a test case for the non-feasible path: wasting time

Some paths are implicit

```
if x < 0 then
   x := -x;
end if
z := x;
```

The else condition is implicit

```
else
   null;
```

- A test case exercising only x<0 reaches the 100% statement coverage, but it does not prevent a bug to occur if x > = 0

- With CGF we can create a test case also x > = 0. Good!