NUS Orbital 2018
Mission Control #6a

Loh Zi Bin Robin
Year 3 Computer Science, NUS

Software Testing

23 June 2018 (Sat)

Food for Thought

- Why is software testing important in a project?
- How do you test a software? How far do you think random testing is effective?
- How can I generate more test cases?
- How can I use automated tools to test my software?

Importance of Software Testing – Quality Assurance

- The process to ensure that the designed software has the necessary levels of quality.

Quality Assurance = Validation + Verification

➢ Validation – Building the right system	(e.g. Are the requirements correct?)
➢ Verification – Building the system right	(e.g. Are the requirements implemented correctly?)

Quality Assurance from Various Perspectives

Student

- Obtain measurable confidence
  - Coverage based
  - Fault based
  - Failure based

Software Engineer

- Test for **comprehension** of requirements:

  For **correctness** and **completeness**

Business User

- Maximise user experience

Terminologies – Fault, Error and Failure

- Aim: Replace **all** elements in array by ‘X’.

```
for (int i = 0; i < 2; i++) {
    arr[i] = ‘X’;
}
```

Input Array	Actual Output
[R | L | Z |]	[X | X | X |] Program exits normally

Fault	Error	Failure
✓	✗	✗

Terminologies – Fault, Error and Failure

- Aim: Replace **all** elements in array by 'X'.

```
for (int i = 0; i < 2; i++) {
    arr[i] = 'X';
}
```

Input Array	Actual Output
R | L | Z | B	X | X | X | B Program exits normally

Fault	Error	Failure
✓	✓	✗

Terminologies – Fault, Error and Failure

- Aim: Replace **all** elements in array by 'X'.

```
for (int i = 0; i < 2; i++) {
    arr[i] = 'X';
}
```

Input Array	Actual Output	Fault	Error	Failure
R | L	X | X Program behaves incorrectly	✓	✓	✓

Terminologies - Testing

Test Case

- A group of input values that cause a program to take some defined action

Test Suite

- A collection of test cases

Test Oracle

- A mechanism for determining if the actual behaviour of a test case execution matches the expected behaviour

Test Effectiveness

- The degree to which testing reveals faults or achieves other objectives.

Test Plan

- A document describing the scope, approach, resources and schedule of intended activity.

Terminologies – Testing vs. Debugging

Testing	Debugging
Reveals faults	Used to remove a fault

Debugging is part of testing.

Terminologies – Random vs Systematic Testing

Random Testing

- Randomly pick possible inputs
- Minimises programmer bias
- Treat all inputs as equally valuable

Systematic Testing

- Attempts to select inputs which are very valuable
- Typically by picking representative values which are pertinent to fail / pass

Types of Testing

Types of Testing

Unit Testing

- **Testing a single, simple component in an isolated environment.**

  (e.g. procedure, class)

Integration Testing

- **Testing parts of the system by combining the modules**

System Testing

- **Testing based on requirements specifications**

Acceptance Testing

- **Testing based on use cases / use case scenarios.**

Regression Testing

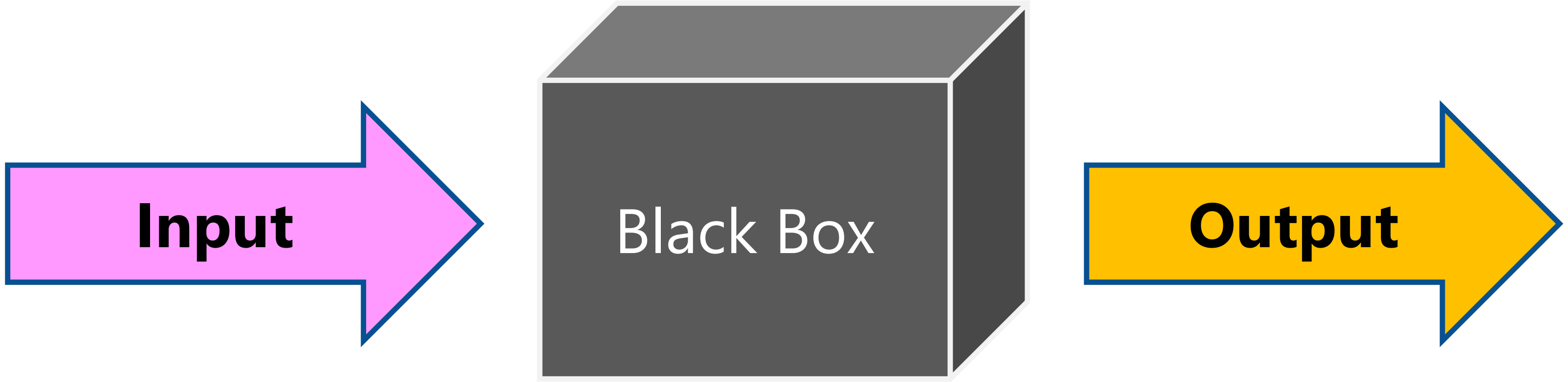
- **Testing based on previous preserved test cases, when a change is made.**

Approaches to Testing

Functional & Structural Testing

Black Box Testing (Functional Testing)

Testing which ignores the internal mechanism of a system / component, and focuses solely on the outputs generated in response to the selected inputs and execution conditions.

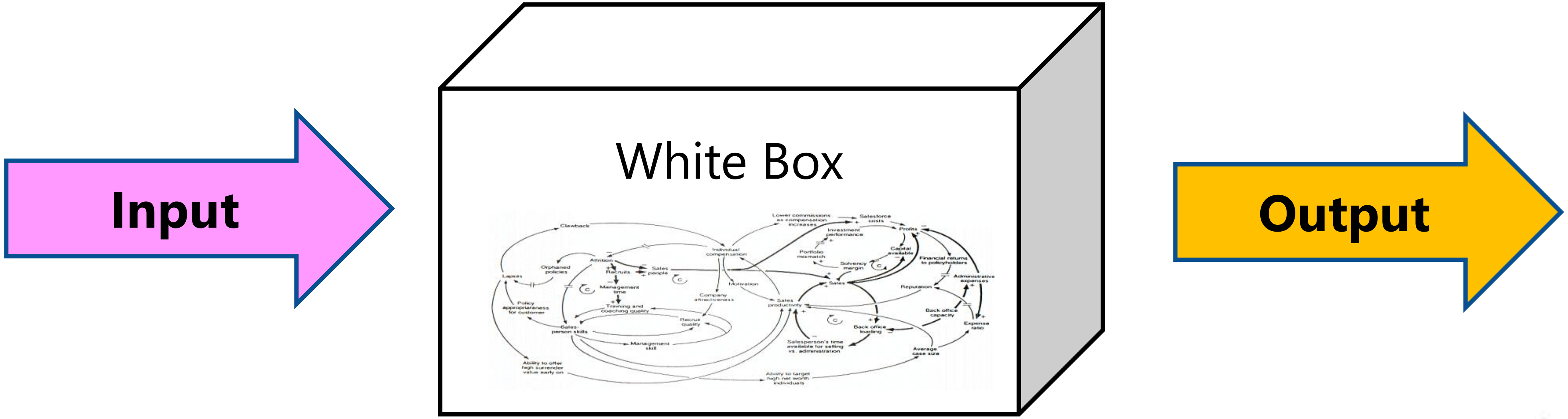


Internal mechanism is unknown

Functional & Structural Testing

White Box Testing (Structural Testing)

Testing which takes into account the internal mechanism of a system or a component, with respect to some well defined coverage criterion.



Testing

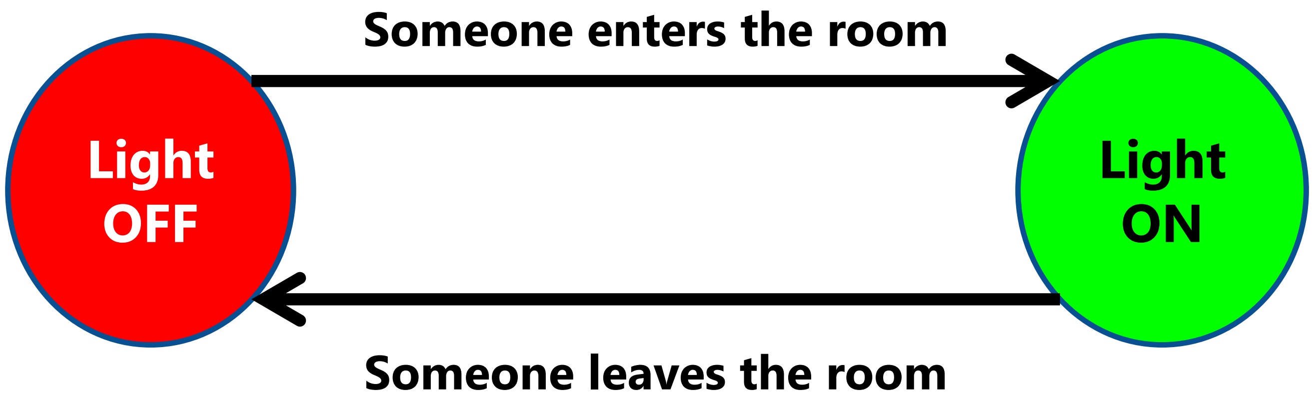
- Model Based
- Specification Based

Model Based Testing

Finite State Machines

- A type of model for describing behaviour that depends on sequences of events or stimuli.

- Example: Light Sensor



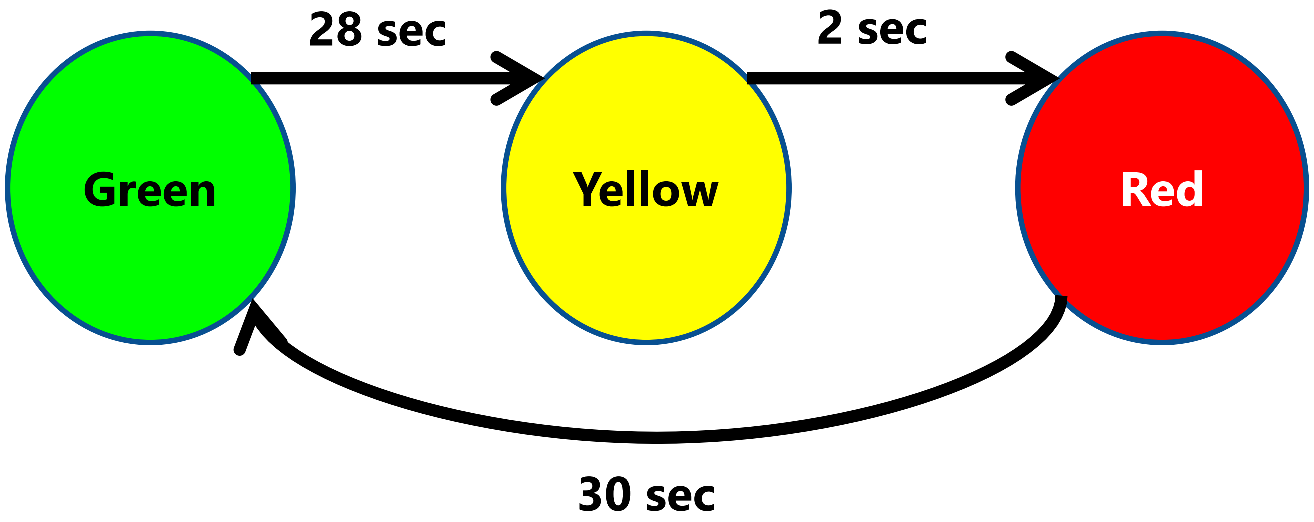
The **state** of an object is modified by methods.

The **transitions** model the methods.

The sequences of transitions that traverse the state machine can be tested!

Finite State Machines Exercise 1

- Draw a finite state machine for a traffic light.



Model Coverage

- **State Coverage**
  (Every state in the model is covered by at least 1 test case)

- **Transition Coverage** [most commonly used criterion]
  (Every transition between states in the model is covered by at least 1 test case)

Model Condition / Decision Coverage (MC/DC)

MC/DC – Example

- An aeroplane has 4 engines, with 2 engines on each side of the wings.

  Suppose this aeroplane can take off with a minimum of 1 engine operating on each side.

  - ➢ How many combinations (minimally) do you need to consider in this situation?

MC/DC – Answer (16 cases)

E1	E2	E3	E4	Out
T	T	T	T	True
T	T	T	F	True
T	T	F	T	True
T	T	F	F	False
T	F	T	T	True
T	F	T	F	True
T	F	F	T	True
T	F	F	F	False

E1	E2	E3	E4	Out
F	T	T	T	True
F	T	T	F	True
F	T	F	T	True
F	T	F	F	False
F	F	T	T	False
F	F	T	F	False
F	F	F	T	False
F	F	F	F	False

MC/DC – Answer (9 cases)

E1	E2	E3	E4	Out
T	F	T	F	True
T	F	F	T	True
T	F	F	F	False

T	F	T	F	True
F	F	T	F	False
F	T	T	F	True
T	F	F	F	False
T	F	F	T	True

E1	E2	E3	E4	Out
F	T	T	F	True
F	T	F	T	True
F	T	F	F	False
F	F	T	F	False
F	F	F	T	False
F	F	F	F	False

Methods to Generate Test Cases

Methods to Generate Test Cases

- **Functional Testing (Combinatorial)**
  - Category Partition Testing
  - Pairwise Testing
  - Catalog Based Testing

Category Partition Testing

Category Partition Testing – Example

- Consider this method which calculates the factorial (i.e. n!) of a number n.

```
int factorial(int n) {
    //  Negative inputs: return 0
    //  Out of range: return -1
}
```

- Identify independently testable features.
  - The method does only 1 thing → There is only 1 testable feature.
  - Testable feature: The method correctly determines & return the factorial of n, or returns an appropriate error number.

Category Partition Testing – Example

- Consider this method which calculates the factorial (i.e. n!) of a number n.

```
int factorial(int n) {
    //  Negative inputs: return 0
    //  Out of range: return -1
}
```

- Identify parameters & environment.
  - ➢ There is only 1 parameter → n
  - ➢ Environment: 32-bit & 64-bit computers (i.e. int has a limited range capacity)

Category Partition Testing – Example

- Consider this method which calculates the factorial (i.e. n!) of a number n.

```
int factorial(int n) {
    //  Negative inputs: return 0
    //  Out of range: return -1
}
```

- Identify categories.
  - Negative n
  - In Range (according to int range capacity)
  - Out of Range (according to int range capacity)

Category Partition Testing – Example

- Identify the representative values.
  - n < -1 [i.e. return 0]
  - n = -1 [i.e. return 0]
  - n = 0
  - n = 1
  - 0 < n < MAX
  - n = MAX - 1
  - n = MAX
  - n = MAX + 1 [i.e. return -1]
  - n > MAX + 1 [i.e. return -1]

Parameters: n

- $n < 0$ [i.e. return 0]
- $0 \le n \le$ MAX [i.e. return n]
- $n >$ MAX [i.e. return -1]

Environment

- 32 bit [i.e. MAX = 12]
- 64 bit [i.e. MAX = 20]

Category Partition Testing – Example

- Identify constraints.
  - Since there is only 1 parameter (i.e. n), there are no additional constraints to consider.

Category Partition Testing – Example

- Enumerate the values.
  - $n < -1$ [i.e. return 0]
  - $n = -1$ [i.e. return 0]
  - $n = 0$
  - $n = 1$
  - $0 < n < MAX$
  - $n = MAX - 1$
  - $n = MAX$
  - $n = MAX + 1$ [i.e. return -1]
  - $n > MAX + 1$ [i.e. return -1]

Environment
- 32 bit [i.e. MAX = 12]
- 64 bit [i.e. MAX = 20]

Category Partition Testing – Example

- Enumerate the values.

Test case	Size of int	*n*	Expected result	Notes
1	Any	-2	0	$n < -1$
2	Any	-1	0	$n = -1$
3	Any	0	1	$n = 0$
4	Any	1	1	n = 1
5	Any	4	24	$0 < n <$ MAX (for all environments)
6	Any	100	-1	$n >$ MAX + 1 (for all environments)
7	32-bit	11	39,916,800	n = MAX – 1, MAX = 12
8	32-bit	12	479,001,600	n = MAX, MAX = 12
9	32-bit	13	-1	n = MAX + 1, MAX = 12
10	64-bit	19	121,645,100,408,832,000	n = MAX - 1, MAX = 20
11	64-bit	20	2,432,902,008,176,640,000	n = MAX, MAX = 20
12	64-bit	21	-1	n = MAX + 1, MAX = 20

Pairwise Testing

Pairwise Testing – Example

- Consider the variety of side dishes to create a meal for a customer in a restaurant.

Main	Sides	Drinks
Burger	Corn	Cola
Fries	Salad	Juice
		Milk

  - The meal contains only 1 dish from each category.
  - Due to an internal policy, the corn has to be together with the burger.

- Generate the minimum number of meal combinations in the shortest time possible.

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Drinks	Main	Sides
Cola	Burger	Corn [if burger]
Juice	Fries	Salad
Milk		

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Main	Sides	Drinks
Cola		
Cola		
Juice		
Juice		
Milk		
Milk		

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Drinks	Main	Sides
Cola	Burger	
Cola	Fries	
Juice	Burger	
Juice	Fries	
Milk	Burger	
Milk	Fries	

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Drinks	Main	Sides
Cola	Burger	Corn
Cola	Fries	Salad
Juice	Burger	Corn
Juice	Fries	Salad
Milk	Burger	Corn
Milk	Fries	Salad

Missing combinations:

- Burger and salad
- Salad & corn?

Possible improvements:

- Swap Corn & Salad for Juice/Milk?

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Solution:

Add a missing combination for burger & salad.

Question:

What should the drink be?

Drinks	Main	Sides
Cola	Burger	Corn
Cola	Fries	Salad
Juice	Burger	Corn
Juice	Fries	Salad
Milk	Burger	Corn
Milk	Fries	Salad
??????	Burger	Salad

Pairwise Testing – Answer

Main	Sides	Drinks
Burger	Corn [if burger]	Cola
Fries	Salad	Juice
		Milk

Answer:

Don't care – Any cola, juice or milk is suitable.

Drinks	Main	Sides
Cola	Burger	Corn
Cola	Fries	Salad
Juice	Burger	Corn
Juice	Fries	Salad
Milk	Burger	Corn
Milk	Fries	Salad
–	Burger	Salad

Note: This is modified from CS4218 Mid-Term (AY2016/17 Semester 2)

Catalog Based Testing

Catalog Based Testing – Example

- Consider a program with this main method:

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

  - Identify the independently testable features.
  - What are the possible representative values for testing this main method? Enumerate these values.

Catalog Based Testing - Answer

- **Independently Testable Features:** Check that <input> is <output>
  - Check that a sequence of characters is a valid tag.

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing - Answer

- **Independently Testable Features:** Check that <input> is <output>
  - Check that a sequence of characters is a valid tag.

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing – Answer [Step 1]

- Pre-conditions: **str** must contain between 4 to 5 characters. [Validated]
  str is not null, not empty [Assumed]

- Post-conditions: **true** if first 3 characters are letters and followed by 1 digit.
  true if first 3 characters are letters and followed by 2 digits.
  false if first 3 characters are not all letters.
  false if first 3 characters are letters and not followed by a digit.
  false if first 3 characters are letters and followed by 1 digit and non-digit.

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing – Answer [Step 1]

- Definitions: First 3 characters are letters [A-Z, a-z]
  Followed by 1 to 2 digits [0-9]
- Variables: A string of alphanumeric characters
- Operations: NIL – No manipulation is done.

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing – Answer [Step 2]

- Preconditions:
  - **str** must contain between 4 to 5 characters
  - **str** contains less than 4 characters or **str** contains more than 5 characters
  - **str** is not null and not empty
  - **str** is null or **str** is empty

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing – Answer [Step 2]

- Post-conditions: **true** if first 3 characters are letters and followed by 1 digit.
  true if first 3 characters are letters and followed by 2 digits.
  false if first 3 characters are not all letters.
  false if first 3 characters are letters and not followed by a digit.
  false if first 3 characters are letters and followed by 1 digit and non-digit.

```
boolean isValidTag(String str) {
    //  Returns true if:
    //      First 3 characters are letters
    //      Followed by up to 2 digits (min. 1 digit is expected)
}
```

Catalog Based Testing – Answer [Step 3]

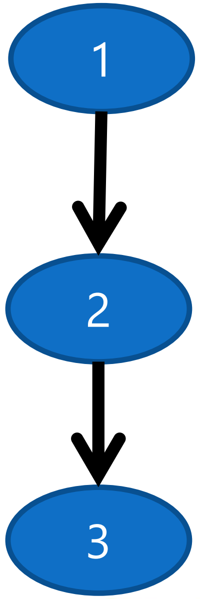
- Boolean
  - [in/out] true
  - [in/out] false
- Enumeration
  - [in/out] each enumerated value
  - [in] values outside of the enumerated set
- Range L..U
  - [in] L-1
  - [in/out] L
  - [in/out] A value between L and U
  - [in/out] U
  - [in] U+1
- Numeric Constant C
  - [in/out] C
  - [in] C - 1
  - [in] C + 1
  - [in] Any other constant in the same data type.
- Non-Numeric Constant C
  - [in/out] C
  - [in] Any other constant in the same data type
  - [in] Some other value of the same data type
- Sequence
  - [in/out] Empty
  - [in/out] A single element
  - [in/out] More than one element
  - [in/out] Maximum length (in bounded) or very large
  - [in] Longer than max length (if bounded)
  - [in] Incorrectly terminated
- Scan with action on element P
  - [in] P occurs at beginning of sequence
  - [in] P occurs in interior of sequence
  - [in] P occurs at end of sequence
  - [in] P appears twice in a row
  - [in] P does not occur in sequence

Structural Testing

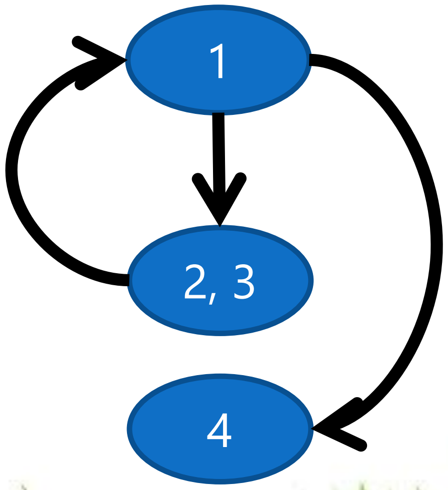
Program Representation – Control Flow Graph (CFG)

- A program can be represented using a CFG.

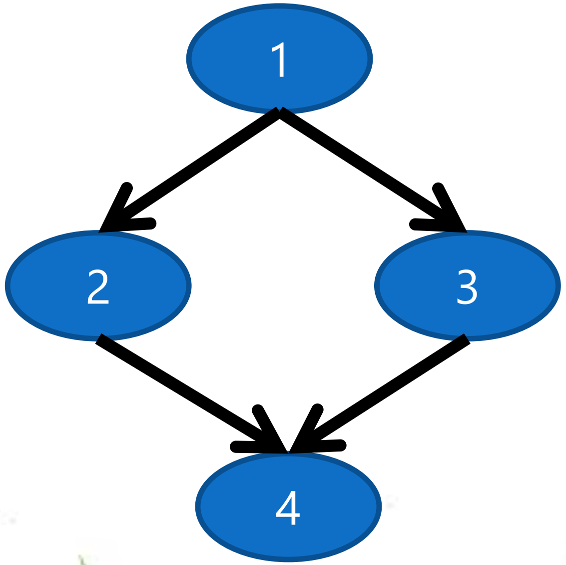
```
1.    a = b;
2.    b = c – d;
3.    b = a;
```




```
1. while (c < d) {
2.        d++;
3.        c = d;
   }
3. c++;
```



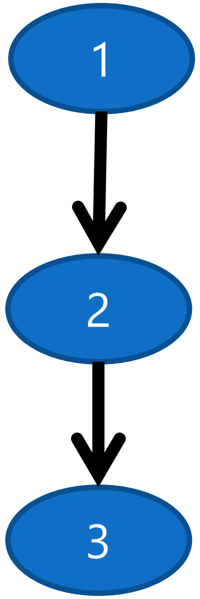

```
1.    if (a < b) {
2.          a = c;
      } else {
3.          b = a;
      }
4.    c = a;
```



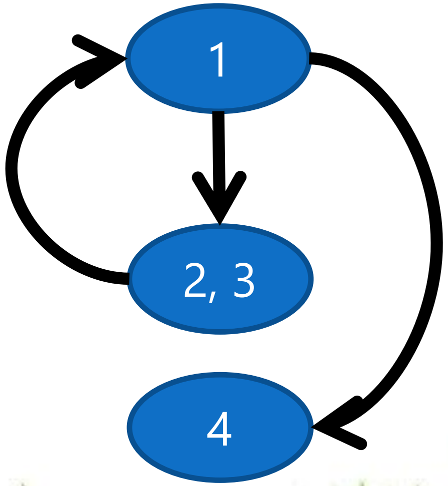
Program Representation – Control Flow Graph (CFG)

- A program can be represented using a CFG.

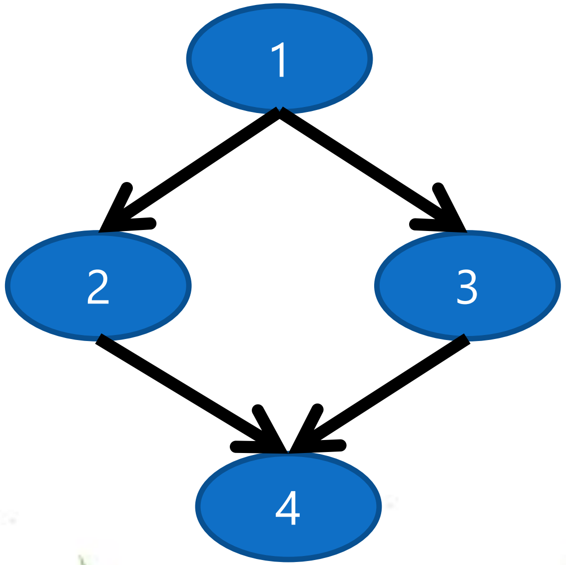
```
1.    a = b;
2.    b = c – d;
3.    b = a;
```




```
1. while (c < d) {
2.       d++;
3.       c = d;
   }
3. c++;
```




```
1.    if (a < b) {
2.          a = c;
      } else {
3.          b = a;
      }
4.    c = a;
```



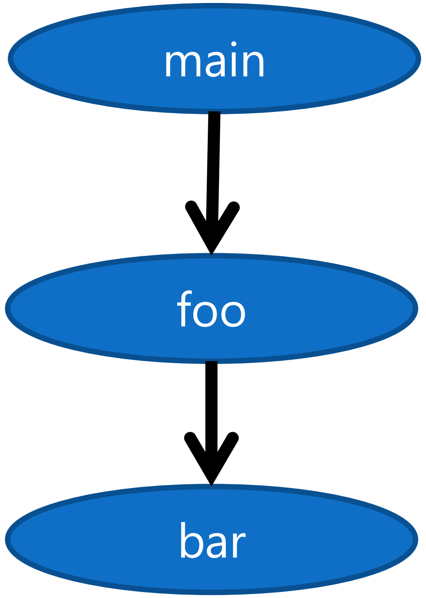
Program Representation – Control Flow Graph (CFG)

- A program can be represented using a CFG.

```
public static void main(String[] args) {
    foo();
}

void foo() {
    bar();
}

void bar() {
}
```



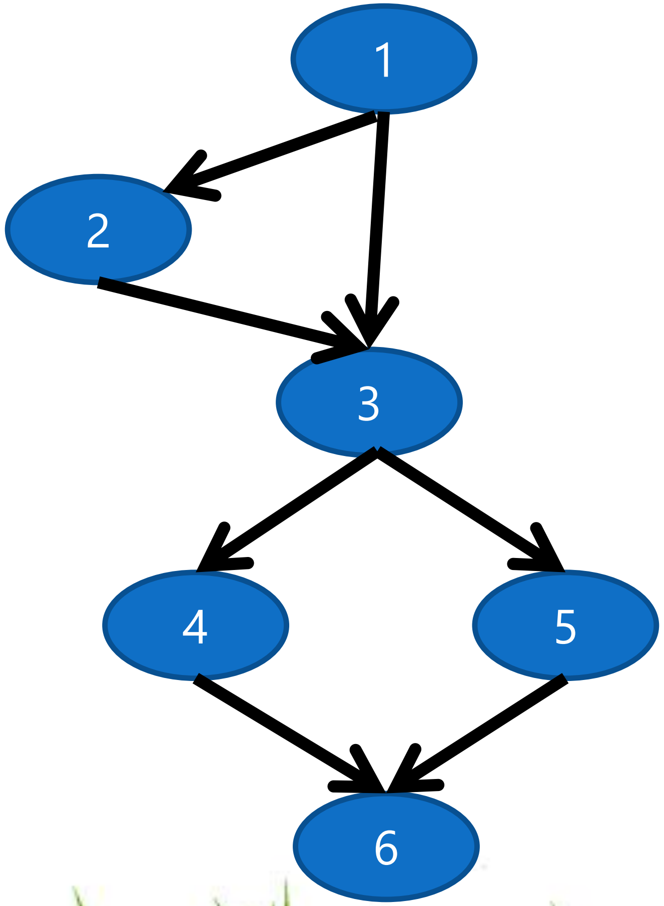
Control Graph Characteristics

	Intra-Procedural CFG	Inter-Procedural CFG (Call Graphs)
Nodes	Maximum code region with 1 entry point & 1 exit point	Procedures (e.g. methods, functions)
Directed Edges	Flow from 1 code region to another	Call relations

CFG Exercise 1

- Draw the control flow graph based on the code fragment below.

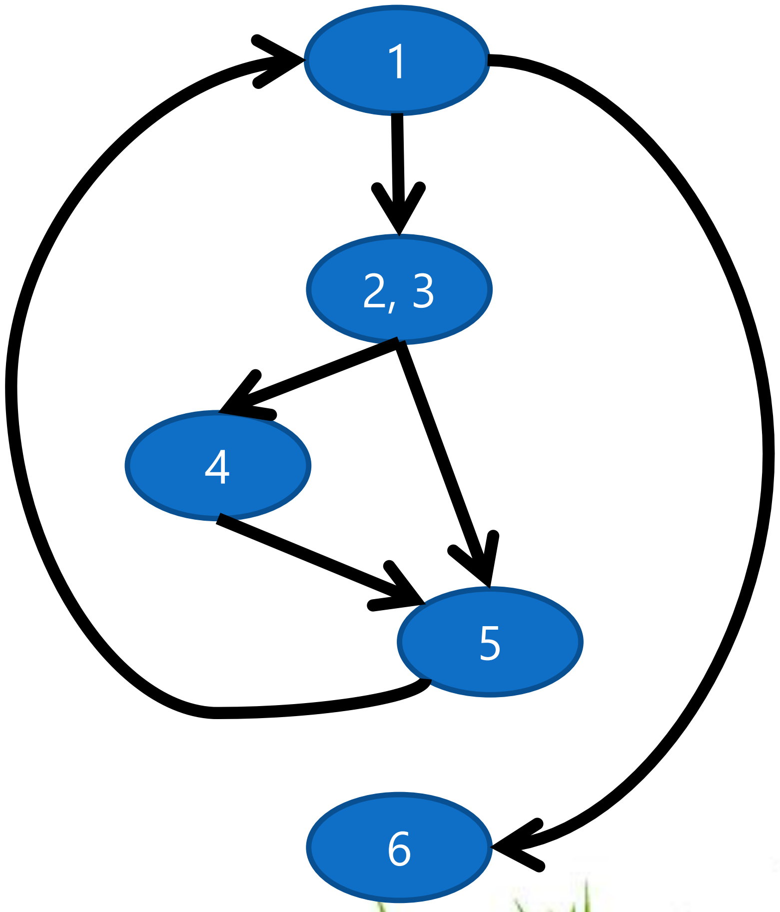
```
1.   if (a < b) {
2.       a = c;
     }
3.   if (b < a) {
4.       c = b;
     } else {
5.       a = b;
     }
6.   c = a;
```



CFG Exercise 2

- Draw the control flow graph based on the code fragment below.

```
1.    sum = 0; i = 0;
2.    while (i <= 99) {
3.        if (i % 2 == 0)
4.            sum = sum + i;
5.        i = i + 1;
      }
6.    return sum;
```



Number of Paths: 200

- **Feasible: 100**
- **Infeasible: 100**

Purpose of CFG

- Identifies cases that may not be identified from specifications alone.
  - Natural differences between specifications and implementation
  - Flaws in the software or its development process

- Determine various coverages manually.

Note: Executing all control flow statements does not guarantee all faults are found – Errors might be masked in the code!

Advice: Create functional test suite first, then measure structural coverage to identify missing cases.

Coverage Criteria

	Coverage	Rationale
Statement Testing	$$\frac{\text{No. of executed statements}}{\text{Total no. of statements}}$$	A fault in a statement can only be revealed by executing the faulty statement.
Branch Testing	$$\frac{\text{No. of executed branches}}{\text{Total no. of branches}}$$	Traversing all edges → All nodes are visited. (Converse is not true)
Condition Testing	$$\frac{\text{No. of truth values consumed by all basic conditions}}{2 \times \text{No. of basic conditions}}$$	Apply MC-DC where necessary
Path Testing	$$\frac{\text{No. of executed paths}}{\text{Total no. of paths}}$$ The total number of paths might be infinitely huge. • Limit the number of loop traversals • Limit the path length to be traversed • Limit the dependencies among selected paths	Sequences of branches might cause faults.

Test Driven Development (TDD)

TDD Cycle

- Decide what behaviour to implement
- Write test cases to exhibit the behaviour → Run these test cases to see them fail
- Implement code behaviour
- Run the test cases again → Keep modifying the code & re-run test cases until they all pass
- Refactor code to improve code quality
- Repeat the cycle for each small unit of behaviour

Final Advice

Demo – Debugging using an IDE

Automated Testing

Build Automation

Continuous Integration

Breakpoints in IDE

Coverage Report