

## Software Testing

[www.cs.uoi.gr/~zarras/http://www.cs.uoi.gr/~zarras/se.htm](http://www.cs.uoi.gr/~zarras/http://www.cs.uoi.gr/~zarras/se.htm)

Slides material sources:

Software Engineering - Theory & Practice, S. L. Pfleeger  
Introduction to Software Engineering, I. Sommerville  
SWEBOK v3: IEEE Software Engineering Body of Knowledge  
Working Effectively with Legacy Code, M. Feathers  
Software Testing – A Craftsman's Approach, P Jorgensen  
JUnit Patterns by Gerard Meszaros

1

## Testing fundamentals

2

## Why does software fail?

3

## Why does software fail?



**Bugs !!!**

A more complete answer includes the famous triplet:

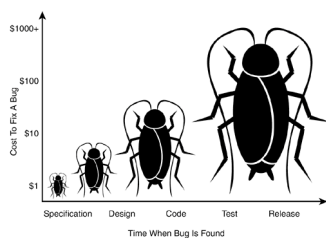
- Errors
- Faults
- Failures

4

## What do we mean by error?

5

### Errors



#### Error

People make errors. A good synonym is mistake.

Errors tend to **propagate**; a requirements error may be magnified during design and amplified still more during coding.

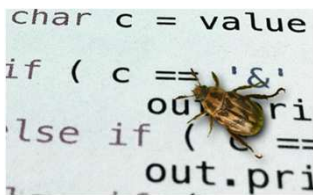
[International Software Testing Qualification Board \(ISTQB\)](#)

6

## What do we mean by fault?

7

### Faults



#### Fault

A **fault** is the **result** of an **error**. It is more precise to say that a fault is **the representation of an error**, where representation is the mode of expression, such as narrative **text**, UML **diagrams**, hierarchy **charts**, and **source code**.

**Defect** (see the ISTQB Glossary) is a good synonym for fault, as is **bug**.

[International Software Testing Qualification Board \(ISTQB\)](#)

8

## What do we mean by failure?

9

## Failures



### Failure

A failure **occurs** when the **code** corresponding to a **fault** executes.

In general a **failure** is the **manifestation** of a **fault**.

[International Software Testing Qualification Board \(ISTQB\)](#)

10

## What is software testing?

11

## Software testing



**Testing** is the act of exercising software with **test cases**.

A test has two distinct **goals**:

To find failures (**verification** aspect).

To demonstrate correct execution (**validation** aspect).

12

## What is a test case?

13

## Test cases

Test Case Template						
Project Name:						
Test Case ID: TC-01	Test Designed by: [Name]					
Test Priority: Low/Medium/High: Mid	Test Designed date: [Date]					
Module Name: Google login module	Test Executed by: [Name]					
Test Title: Verify login with valid username and password	Test Execution date: [Date]					
Description: Check the Google login page.						
Pre-conditions: User has valid username and password.						
Dependencies:						
Step	Test Steps	Test Data	Expected Results	Actual Results	Status (Pass/Fail)	Notes
1	Click on login page	Open <a href="https://accounts.google.com">https://accounts.google.com</a>	User should be able to login	User is redirected to dashboard with message	Pass	
2	Provide valid username	Username: admin				
3	Provide valid password					
4	Click on login button					
Post-conditions: User is validated with database and successfully login to account. The account session details are logged in database.						

The essence of software testing is to **determine** a **set of test cases** for the **item** to be **tested**.

A **test case** is (or should be) a recognized **work product**.

A **complete test case** will contain a test case **identifier**, a brief statement of **purpose**, a description of **preconditions**, the actual test case **inputs**, the **expected outputs**, a description of **expected post-conditions** (system state after **test execution**), and an execution **history**.

The execution **history** is primarily for test management use—it may contain the **date** when the test was **run**, the **person** who ran it, the **version** on which it was run, and the **pass/fail** result.

14

## Why should we write tests?

15

## Why should we write tests ?

### We need tests to improve software quality

- ▶ Tests as specification.
  - ▶ Insure that we build the right software.
- ▶ Defect localization.
  - ▶ Insure that the software is correct.
- ▶ Defect prevention.
  - ▶ Insure that bugs wont crawl back to the software.

### We need tests to improve software understanding

- ▶ Tests as documentation.
  - ▶ Allow the developer/maintainer to answer questions like “what should be the expected outcome of the software is the given input is ...”



16



## How do we write good tests?

17

## How do we write good tests ?

### Tests should not introduce new risks

- ▶ **Refrain from modifying the software** to facilitate the development of the tests as safety net.

### The tests that we write should be easy to run

- ▶ Fully automated.
  - ▶ Execute without any effort.
- ▶ Self checking.
  - ▶ Detect and report any errors without human intervention.
- ▶ Repeatable.
  - ▶ Can be run many times in a row and produce the same results without human interventions in between.
- ▶ Independent from each other.
  - ▶ Can be run by themselves and **NOT** depend on the execution **order**, **failure** or **success** of **other tests**.

18

## Which are the targets of testing?

19

## Testing targets



The target of the test can vary:

A **single module**, a **group** of such **modules** (related by purpose, use, behavior, or structure), or an entire **system**.

Three test stages can be distinguished: **unit**, **integration**, and **system**.

20

## xUnit Basic Patterns

<http://xunitpatterns.com/index.html>

Examples:

<https://github.com/zarras/myy803-junit-mockito-tutorials/tree/master/basic-xunit-patterns>

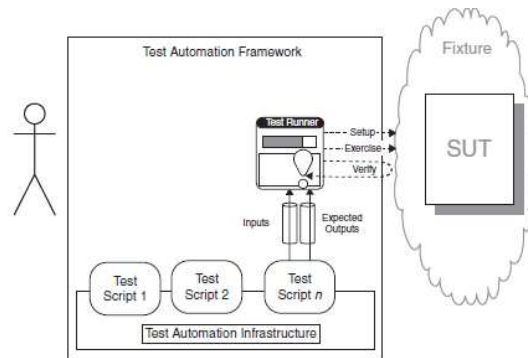
21

**How do we make it easy to write  
and run tests written by different  
people?**

22

## Test Automation Framework

We use a **framework** that **provides all the mechanisms** needed to **run the test logic** so the test writer needs to provide only the test-specific logic.



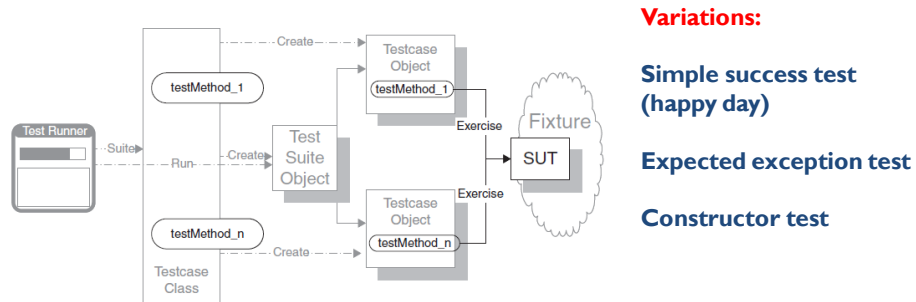
23

**Where do we put our test code?**

24

## Test Method

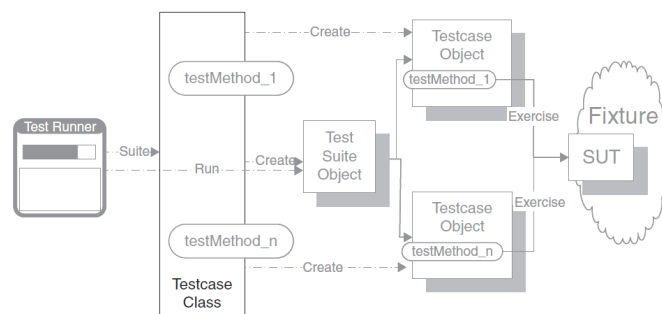
We encode **each test** as a **single Test Method** on some class.



25

## Test Case Class

We **group** a set of related **Test Methods** on a single **Testcase Class**.



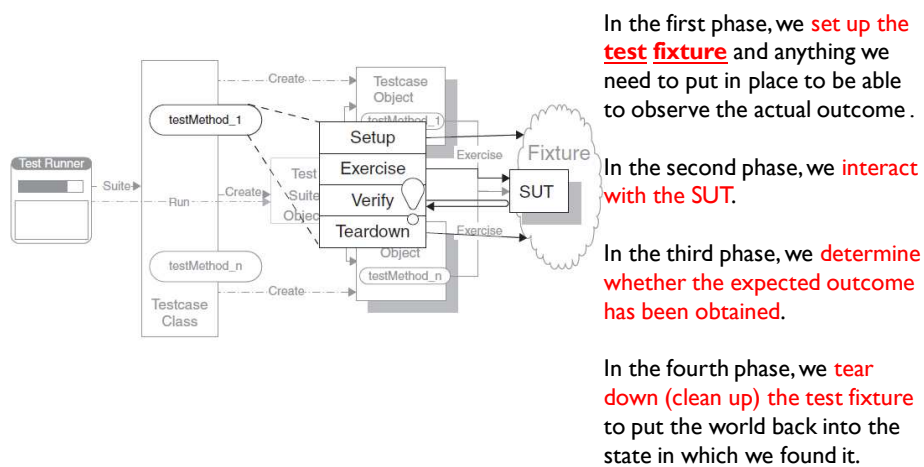
26

## How do we structure our test code?

27

## Four Phase Test

We structure each test with **four distinct parts** executed in sequence.



28

## How do we make tests self-checking?

29

## Assertion Method

**We call a `xUnit` assertion method to evaluate whether an expected outcome has been achieved.**

### Variations:

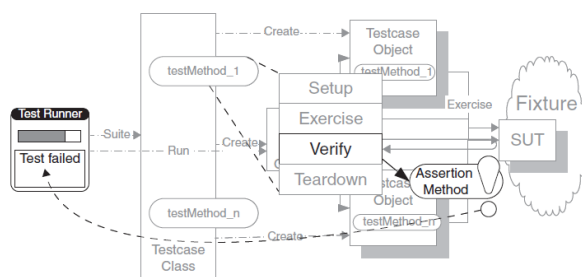
### (In) Equality assertions

### Fuzzy equality assertions (for floating point results with an error tolerance)

**Stated outcome assertions**  
(is null, is true, ...)

**Expected exception assertions.**

### Single outcome assertions (fail)



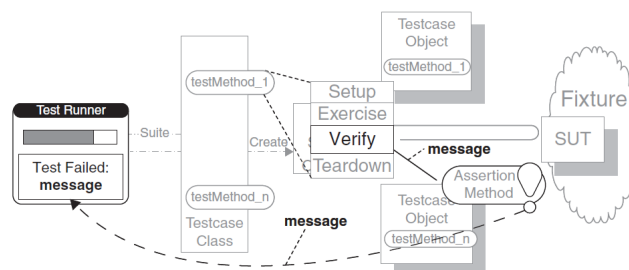
30

## How do we provide more information about a failed assertion?

31

## Assertion Message

We include a **descriptive string argument** in each call to an **Assertion Method**.



32

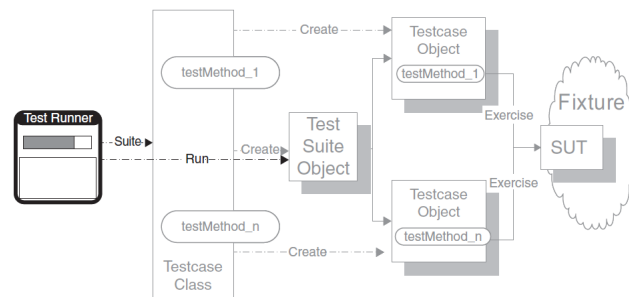


## How do we run the tests?

33

### Test Runner

We execute the xUnit framework's specific program that **instantiates** and executes the **Testcase Objects**. When we have many tests to run we can organize them in **Test Suites**.



34

## Fixture Setup/Teardown Patterns

<http://xunitpatterns.com/index.html>

Examples:

<https://github.com/zarras/myy803-junit-mockito-tutorials/tree/master/fixture-patterns/>

35

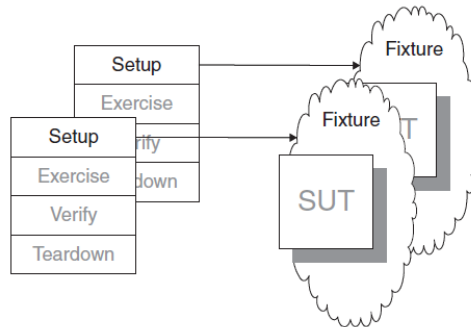
## What is a **test fixture**?

**A test fixture** is everything we need in place to be able to test the **System Under Test (SUT)**

36

## Fresh Fixture

**Each test constructs its own brand-new test fixture for its own private use.**

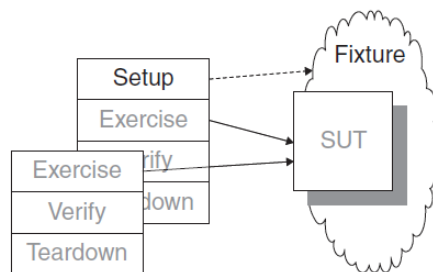


We should use a Fresh Fixture whenever we want to avoid any interdependencies between tests (which is in fact almost always the case .....)

37

## Shared Fixture

**We reuse the same instance of the test fixture across many tests.**



If we want to avoid slow tests.

Or, when we have a long, complex sequence of actions, each of which depends on the previous actions. In customer tests, this may show up as a workflow; in unit tests, it may be a sequence of method calls on the same object.

**With the big risk (!!!!) of introducing interdependencies between tests ....**

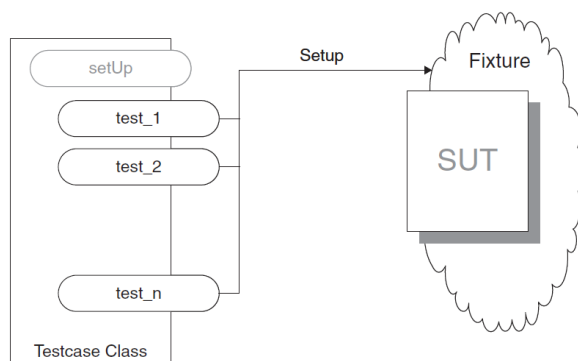
38

## How do we construct (destroy) a fresh fixture?

39

### Inline Setup (Teardown)

Each Test Method creates its own Fresh Fixture by calling the appropriate constructor methods to build exactly the test fixture it requires (the method destroys the fixture at the end).

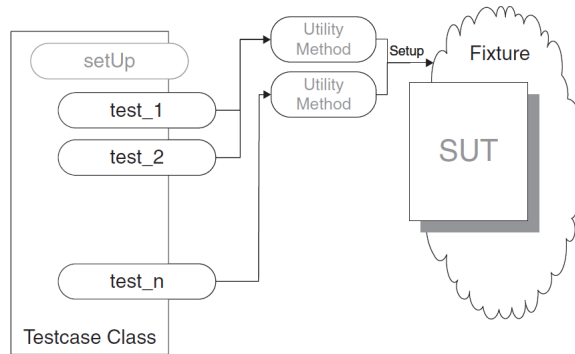


We can use In-line Setup when the fixture setup logic is **very simple** and straightforward.

40

## Delegated Setup (Teardown)

**Each Test Method creates (destroys) its own Fresh Fixture by calling Creation/Destruction Methods from within the Test Methods.**

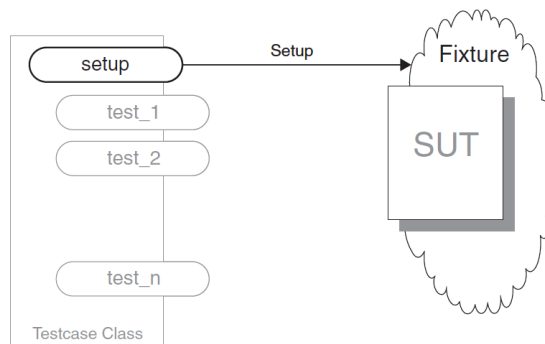


We can use a Delegated Setup when we want to **avoid the test code duplication** caused by having to set up similar fixtures for several tests and we want to **keep the nature of the fixture visible within the Test Methods**

41

## Implicit Fresh Fixture Setup (Teardown)

**We build (destroy) the test fixture common to several tests in set up/tear down methods called by the test framework.**



We can use Implicit Setup when **several Test Methods** on the same Testcase Class need an **identical Fresh Fixture**.

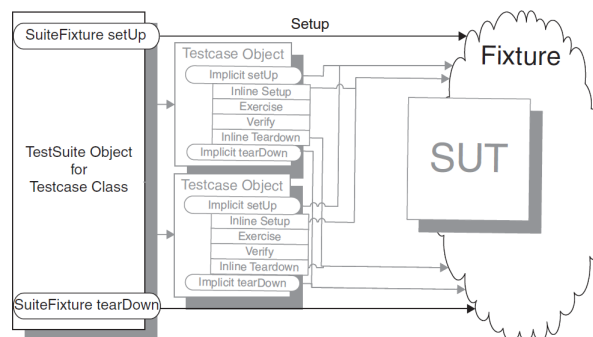
42

**How do we create (destroy) a shared fixture if the test methods that need it are in the same test class?**

43

## Implicit Shared Fixture Setup (Teardown)

**We build (destroy) the shared fixture in special methods called by the Test Automation Framework before/after the first/last Test Method is called**



44

## Result Verification Patterns

<http://xunitpatterns.com/index.html>

Examples:

<https://github.com/zarras/myy803-junit-mockito-tutorials/tree/master/result-verification-patterns>

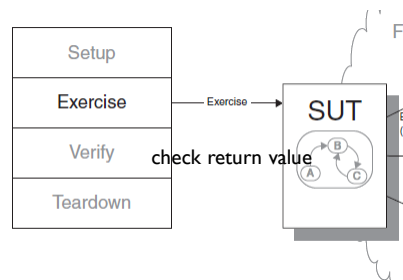
45

**How do we verify a method that  
returns a value?**

46

## Return Value Verification

We inspect the **returned value** of the method and compare it with an **expected return value**.



47

**How do we verify a method that changes the state of the SUT?**

48



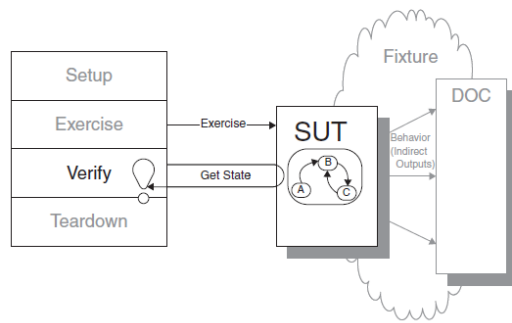
## State Verification

We **inspect the state** of the system under test after it has been exercised and compare it to the **expected state**.

### Variations:

**Procedural State Verification**, we simply write a series of calls to Assertion Methods that pick apart the state information into pieces and compare to individual expected values.

**Expected State Specification**, we construct a specification for the post-exercise state in the form of one or more objects populated with the expected attributes. We then compare the actual state with these objects.



49

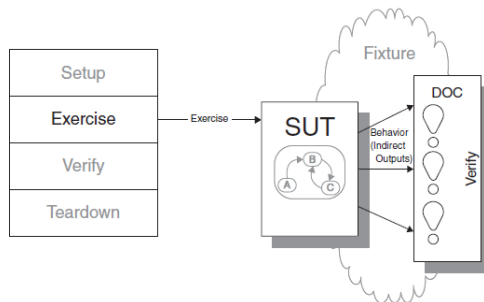
**How do we verify a method of the SUT that interacts with other Depend-On-Components (DOC)?**

50

## Behavior Verification

**We capture the indirect outputs/interactions of the SUT with DOC as they occur and compare them to the expected behavior.**

Usually this is done with the help of the test framework.

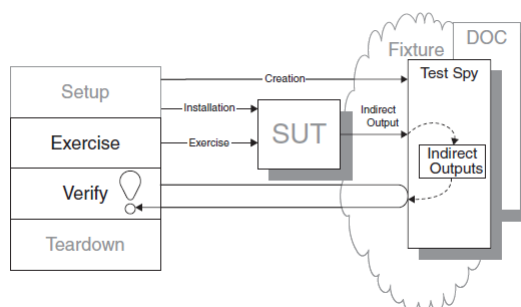


Typically to verify behavior we have to use some kind of a **Test Spy or Test Fake or Test Stub** (see Test Double patterns that follow)

51

## Test Spy

**We use a Test Spy to wrap the DOC to capture the indirect output calls made to DOC by the SUT for later verification by the test.**



**Spy Implementation options:**

Use a **mocking framework** like mockito (spy() and verify() commands).

Examples

<https://github.com/zarras/myy803-junit-mockito-tutorials/tree/master/test-double-patterns/>

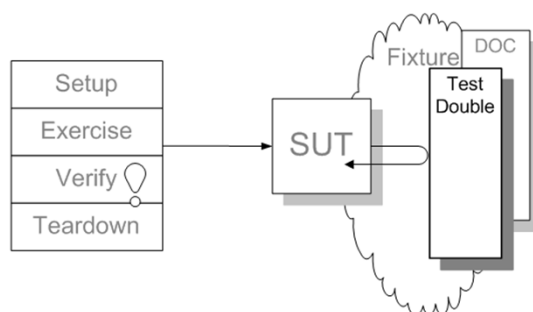
52

**How do we verify the behavior of the SUT when it calls another component, independently from this component?**

53

## Test Double

We **replace** the **DOC** that the **SUT** depends on with a much lighter-weight implementation.



**Fake Object** – a test double that we just call.

**Stub Object** – a test double that we call to get some predetermined values

**Mock Object** – a test double that we use to check the interaction of the SUT with the DOC

Examples  
<https://github.com/zarras/myy803-junit-mockito-tutorials/tree/master/test-double-patterns/>

**Implementation:**  
 Use a **mocking framework** like mockito (mock()) and when-then-return/throw commands).

54

## Testing techniques

55

**How do we create test cases?**

56

## How do we create test cases?



There are several ways:

Based on the software engineer's **intuition** and **experience**, the **specifications**, the **code structure**, the **real** or **imagined** faults to be discovered, predicted usage, models, or the **nature** of the application.

Sometimes these techniques are classified as **white-box** (also called **glass-box, code based**), if the tests are based on information about how the software has been **designed** or **coded**, or as **black-box** (**input domain based**) if the test cases rely only on the **input/output behavior** of the software.

57

**Which is the most widely  
practiced technique?**

58

## Ad-hoc testing



Perhaps the most widely practiced technique **is ad hoc testing**:

Tests are derived relying on the software engineer's **skill, intuition, and experience** with **similar programs**.

Ad hoc testing can be useful for **identifying test cases** that **not easily generated** by more **formalized techniques**.

**Test (good, bad) scenarios, use cases, user stories, .....**

59

**How about input domain based techniques?**

60

## Input domain based techniques



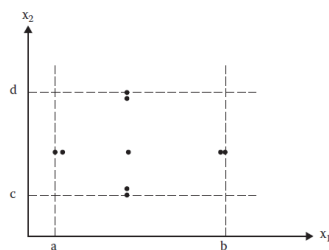
Two widely know categories are:

**Boundary value** testing techniques.

**Equivalence class** testing techniques.

61

## Normal boundary value technique



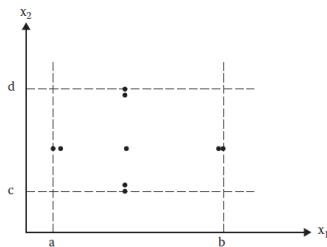
Boundary value analysis test cases for a function of two variables.

The **rationale** behind **boundary value testing** is that **errors** tend to **occur** near the **extreme values** of an input parameter.

The basic idea of **normal boundary value analysis** is to use input parameter values at their **minimum**, **just above the minimum**, a **nominal value**, **just below their maximum**, and at their **maximum**.

62

## Normal boundary value technique



Boundary value analysis test cases for a function of two variables.

### Generalization

If we have a function of  $n$  parameters\*\*, we hold all but one at the nominal values and let the remaining variable assume the

min, min +  $t$ , nom, max -  $t$ , and max

Where  $t$  is an appropriate threshold we chose for the parameter

To create all the test cases we repeat this for each parameter. Thus, for a function of  $n$  parameters, boundary value analysis yields  $4n + 1$  unique test cases.

\*\* In general when we talk about parameters we refer to the test input data. So these could also be method parameters, object attributes, etc....

63

```
class Triangle {
    public void checkType(int sideA, int sideB, int sideC){
        if((sideA < 1) || (sideA > 200) || (sideB < 1) ||
            (sideB > 200) || (sideC < 1) || (sideC > 200)) {
            System.out.println("Wrong input");
            return;
        }
        if(
            // then check if the triangle inequality holds
            (sideA >= sideB + sideC) ||
            (sideB >= sideA + sideC) ||
            (sideC >= sideA + sideB)){
            System.out.println("Not a Triangle");
            return;
        }
        // check if it is equilateral
        if((sideA == sideB) && (sideA == sideC) && (sideB == sideC)){
            System.out.println("The triangle is equilateral");
            return;
        }
        // if not equilateral, check if it is isosceles
        if((sideA == sideB) || (sideA == sideC) || (sideB == sideC)){
            System.out.println("The triangle is isosceles");
            return;
        }
        // otherwise it is scalene
        System.out.println("The triangle is scalene");
        return;
    }
}
```

64

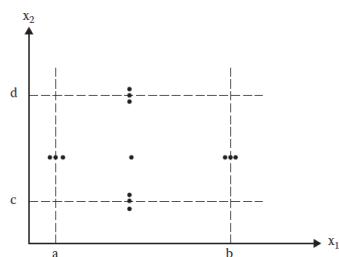


sideA: [1, 200] sideB: [1, 200] sideC: [1, 200]  $t = 1$

Test Case	sideA	sideB	sideC	Expected output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a triangle

65

## Robust boundary value technique



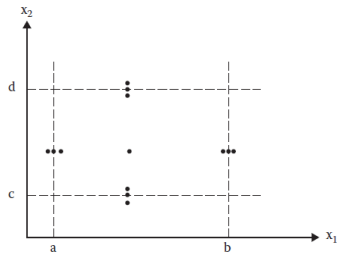
Robust boundary value analysis test cases for a function of two variables.

**Robust boundary value** testing is a simple **extension** of **normal boundary value** testing:

In addition to the five boundary value analysis values of a variable, we see **what happens when the extremes are exceeded** with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-).

66

## Robust boundary value technique



Robust boundary value analysis test cases for a function of two variables.

### Generalization

If we have a function of  $n$  variables, we hold all but one at the nominal values and let the remaining variable assume the

$\min - t, \min, \min + t, \text{nom}, \max - t, \max, \max + t$

Where  $t$  is an appropriate threshold we chose for the variable

To create all the test cases we repeat this for each variable. Thus, for a function of  $n$  variables, boundary value analysis yields  $6n + 1$  unique test cases.

**\*\* In general when we talk about parameters we refer to the test input data. So these could also be method parameters, object attributes, class static attributes, etc....**

67

```
class Triangle {
    public void checkType(int sideA, int sideB, int sideC){
        if((sideA < 1) || (sideA > 200) || (sideB < 1) ||
           (sideB > 200) || (sideC < 1) || (sideC > 200)) {
            System.out.println("Wrong input");
            return;
        }
        if(
            // then check if the triangle inequality holds
            (sideA >= sideB + sideC) ||
            (sideB >= sideA + sideC) ||
            (sideC >= sideA + sideB)){
            System.out.println("Not a Triangle");
            return;
        }
        // check if it is equilateral
        if((sideA == sideB) && (sideA == sideC) && (sideB == sideC)){
            System.out.println("The triangle is equilateral");
            return;
        }
        // if not equilateral, check if it is isosceles
        if((sideA == sideB) || (sideA == sideC) || (sideB == sideC)){
            System.out.println("The triangle is isosceles");
            return;
        }
        // otherwise it is scalene
        System.out.println("The triangle is scalene");
        return;
    }
}
```

68

sideA: [1, 200] sideB: [1, 200] sideC: [1, 200] t = 1

Test Case	sideA	sideB	sideC	Expected output
1	100	100	0	Wrong input
2	100	100	1	Isosceles
3	100	100	2	Isosceles
4	100	100	100	Equilateral
5	100	100	199	Isosceles
6	100	100	200	Not a triangle
7	100	100	201	Wrong input
8	100	0	100	Wrong input
9	100	1	100	Isosceles
10	100	2	100	Isosceles
11	100	199	100	Isosceles
12	100	200	100	Not a triangle
13	100	201	100	Wrong input
14	0	100	100	Wrong input
15	1	100	100	Isosceles
16	2	100	100	Isosceles
17	199	100	100	Isosceles
18	200	100	100	Not a triangle
19	201	100	100	Wrong input

69

## Issues and limitations

Boundary value analysis works well with a function of several independent parameter that represent **bounded physical quantities**.

The parameters need to be described by a **true ordering relation**, in which, for every pair  $\langle a, b \rangle$  of values of a parameter, it is possible to say that  $a \leq b$ .

Test values for alphabet characters, for example, would be  $\{a, b, m, y, \text{ and } z\}$ .

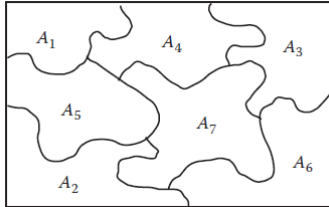
When no explicit bounds are present, we usually have to create **"artificial" bounds** (e.g., language specific Integer.MAX\_VALUE, Integer.MIN\_VALUE, etc).

Boundary value analysis **does not make much sense** for **boolean variables**; we can use as the extreme values TRUE and FALSE.



70

## Equivalence class testing



### Math preliminaries

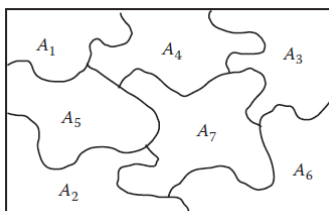
Given a set  $B$ , and a set of subsets  $A_1, A_2, \dots, A_n$  of  $B$ , the subsets are a **partition** of  $B$  iff

$A_1 \cup A_2 \cup \dots \cup A_n = B$ , and  $i \neq j \Rightarrow A_i \cap A_j = \emptyset$ .

71

## Equivalence class testing

### Math preliminaries



Suppose we have a partition  $A_1, A_2, \dots, A_n$  of  $B$ .

Based on this partition **two elements,  $b_1$  and  $b_2$**  of  $B$ , are **related** if  $b_1$  and  $b_2$  are in the **same partition element**.

This is an **equivalence relation** because:

It is **reflexive** (any element is in its own partition),

It is **symmetric** (if  $b_1$  and  $b_2$  are in a partition element, then  $b_2$  and  $b_1$  are)

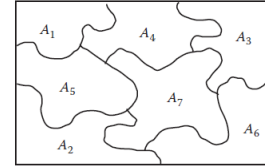
It is **transitive** (if  $b_1$  and  $b_2$  are in the same set, and if  $b_2$  and  $b_3$  are in the same set, then  $b_1$  and  $b_3$  are in the same set).

72

## Equivalence class testing

### Equivalence class testing

**Assume we** test a function  $f$  with  $n$  inputs:  $v_1, v_2, \dots, v_n$   
Each input  $v_i$  has a domain  $\text{dom}(v_i)$



Our **target** set  $B$  is the possibly infinite set of **input tuples**, i.e.  $B = \text{dom}(v_1) \times \text{dom}(v_2) \times \dots \times \text{dom}(v_n)$

Our goal is to define a **partition** of  $B$ .

This partition **would be useful for testing** if for all the **related** input tuples (i.e., the tuples that belong to each  $A_i$ ) the **expected behavior** of  $f$  is the **same** (although the **exact outputs may differ**).

➔ In a sense we try to define classes ( $A_1, A_2, \dots$ ) of expected outputs

Then the idea is to select **at least one test case** from each partition element  $A_i$ .

73

```
class Triangle {
    public void checkType(int sideA, int sideB, int sideC){
        .....
    }
}
```

For the triangle problem we can have the following partition:

$$B = A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5$$

$A_1 = \{a, b, c \mid \text{wrong input}\}$   $A_2 = \{a, b, c \mid \text{not a triangle}\}$   $A_3 = \{a, b, c \mid \text{equilateral}\}$

$A_4 = \{a, b, c \mid \text{isosceles}\}$   $A_5 = \{a, b, c \mid \text{scalene}\}$

Test Case	sideA	sideB	sideC	Expected output
1	100	100	0	Wrong input
2	100	100	200	Not a triangle
3	100	100	100	Equilateral
4	100	199	100	Isosceles
5	100	200	45	Scalene

74

## How about code based techniques?

75

### Code based techniques



Two widely know categories are:

**Control flow** testing techniques.

**Data flow** testing techniques.

76

## Which are the fundamental concepts of control flow techniques?

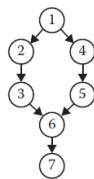
77

### Control flow techniques

If-Then-Else

```

1 If <condition>
2 Then
3   <then statements>
4 Else
5   <else statements>
6 End If
7 <next statement>
  
```



Pretest loop

```

1 While <condition>
2   <repeated body>
3 End While
4 <next statement>
  
```



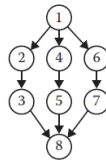
The **control flow** testing techniques are based on the concept of program graphs.

Given a program/function, its **program graph** is a **directed graph** in which **nodes** are **statement fragments**, and **edges** represent **flow of control**.

Case/Switch

```

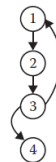
1 Case n of 3
2   n=1:
3     <case 1 statements>
4   n=2:
5     <case 2 statements>
6   n=3:
7     <case 3 statements>
8 End Case
  
```



Posttest loop

```

1 Do
2   <repeated body>
3 Until <condition>
4 <next statement>
  
```



**Program graphs of structured programming constructs**

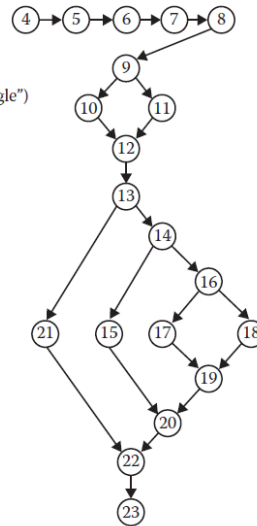
78

## Control flow techniques

```

1 Program triangle2
2 Dim a,b,c As Integer
3 Dim IsATrinagle As Boolean
4 Output("Enter 3 integers which are sides of a triangle")
5 Input(a,b,c)
6 Output("Side A is", a)
7 Output("Side B is", b)
8 Output("Side C is", c)
9 If (a < b + c) AND (b < a + c) AND (c < a + b)
10 Then IsATrinagle = True
11 Else IsATrinagle = False
12 EndIf
13 If IsATrinagle
14 Then If (a = b) AND (b = c)
15 Then Output ("Equilateral")
16 Else If (a≠b) AND (a≠c) AND (b≠c)
17 Then Output ("Scalene")
18 Else Output ("Isosceles")
19 EndIf
20 EndIf
21 Else Output("Nota a Triangle")
22 EndIf
23 End triangle2

```



79

## Control flow techniques

Simplified views (**DD-graphs**) of program graphs make things easier.

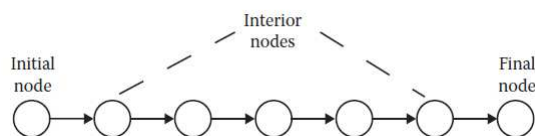
Starting for the program graph we produce the DD-graph as follows:

We keep the initial and the final nodes as is

We keep (decision) nodes with out degree  $\geq 2$  as is

We keep nodes with in degree  $\geq 2$  as is

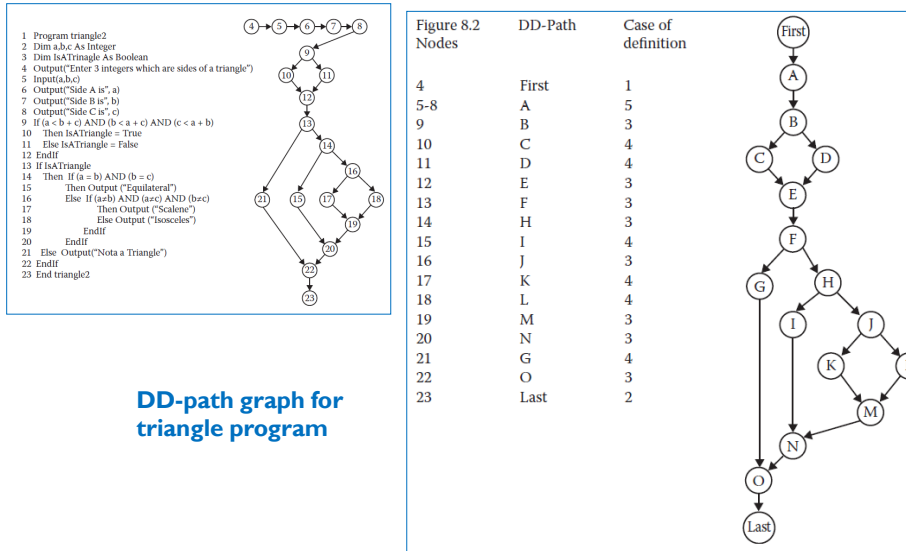
We replace chains (with length  $\geq 2$ ) of nodes with indeg = 1 and outdeg = 1 with a single node



80

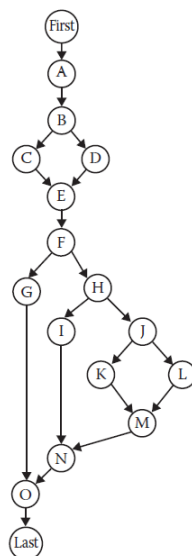


## Control flow techniques



81

## Statement testing



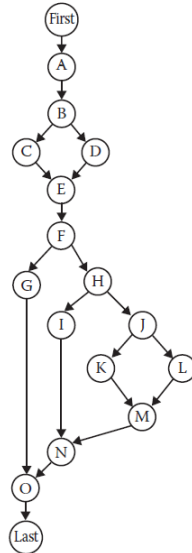
In **statement testing** our goal is to select a **set of test cases T** that satisfies the **node coverage** criterion.

A set of test cases T for a program/function, **satisfies node coverage** if, when executed on the program/function, **every node** in the program graph is traversed.

Denote this level of coverage as  $G_{node}$ , where the **G** stands for program graph.

82

## Branch testing



In **branch testing** our goal is to select a **set of test cases T** that satisfies the **edge coverage** criterion.

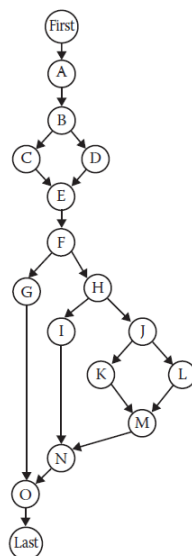
A set of test cases T for a program/function **satisfies edge coverage** if, when executed on the program/function, every edge in the program graph is traversed.

Denote this level of coverage as  $G_{edge}$ .

The **difference** between  $G_{node}$  and  $G_{edge}$  is that, in the latter, we are assured that all outcomes of a decision-making statement are executed.

83

## Path testing



In **path testing** our goal is to select a **set of test cases T** that satisfies the **path coverage** criterion.

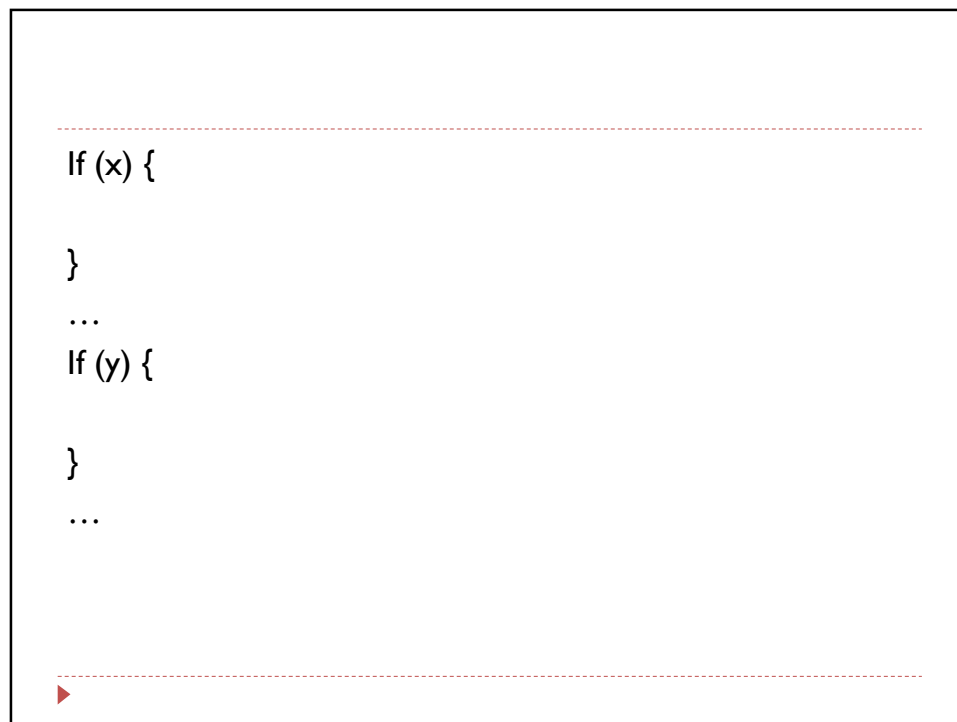
A set of test cases T for a program/function **satisfies path coverage** if, when executed on the program, every **feasible** path from the source node to the sink node in the program graph is traversed.

Denote this level of coverage as  $G_{path}$ .

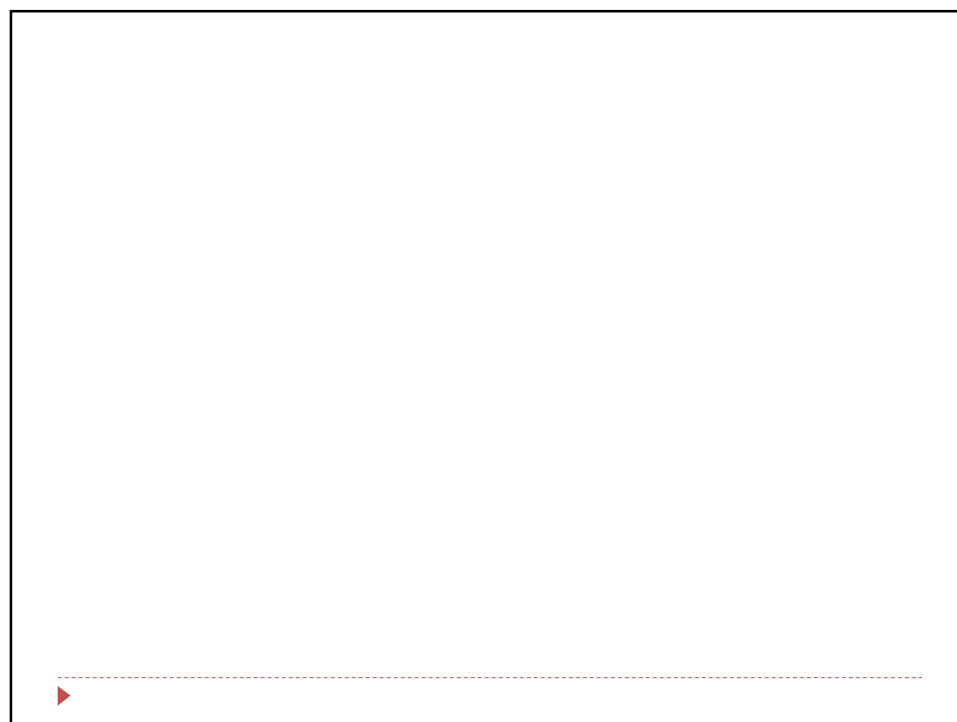
The **difference** between  $G_{edge}$  and  $G_{path}$  is that, in the latter, we are assured that all **possible combinations** of outcomes of decision-making statements are executed.

**Keep in mind the possibility of infeasible paths and dependent decision points.**

84



85



86