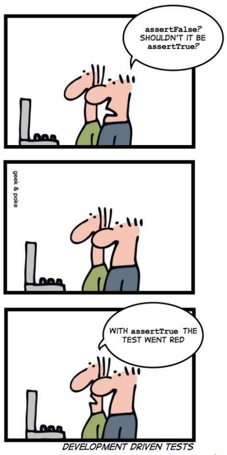


Software Testing

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

Testing fundamentals

What does software fail?

Why does software fail?

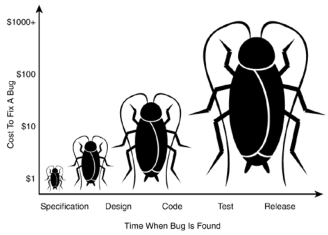
Bugs !!!

A more complete answer includes the famous triplet:

- Errors
- Faults
- Failures

What do we mean by error?

Errors



Error

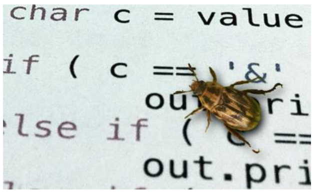
People make errors. A good synonym is mistake.

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

International Software Testing Qualification Board (ISTQB)

What do we mean by fault?

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)

What do we mean by failure?

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)

What is software testing?

Software testing

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

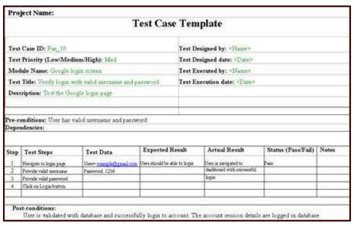
A test has two distinct goals:

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

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

What is a test case?

Test cases



Project Name:

Test Case Template

Test Case ID: Fun_10	**Test Designed by:** <Name>
Test Priority (Low/Medium/High): Med	**Test Designed date:** <Date>
Module Name: Google login screen	**Test Executed by:** <Name>
Test Title: Verify login with valid username and password	**Test Execution date:** <Date>
Description: Test the Google login page	

Pre-conditions: User has valid username and password

Dependencies:

Step	Test Steps	Test Data	Expected Result	Actual Result	Status (Pass/Fail)	Notes
1	Navigate to login page	User= example@gmail.com	User should be able to login	User is navigated to dashboard with successful login	Pass	
2	Provide valid username	Password: 1234				
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.

How do I develop good tests?

How do I develop good tests?

- Well-designed tests exhibit the following properties:

[OO Reengineering Patterns, DeMeyer, Ducasse, Nierstraz, 2013]

- Automation. Tests should run without human intervention.
  - Only fully automated tests offer an easy way to check after every change to the system whether it still works as it did before.
  - By minimizing the effort needed to run tests, developers will hesitate less to use them.
- Persistence. Tests must be stored to be automatable.
  - Each test documents its test data, the actions to perform, and the expected results.
- Efficiency and failure localization.
  - Tests should run fast for getting quick feedback about the program.
  - Tests should allow us to quickly find the points of the program that break.

This is why we typically write tests using XUnit and Mocking frameworks!!

Testing targets

Which are the targets of testing?

Testing targets

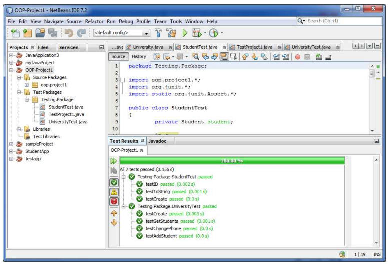
The target of the test can vary:

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.

What is unit testing?

Unit testing



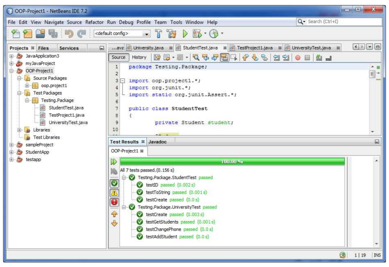
Common to most conceptions of unit tests is the idea that they are **tests in isolation** of individual components of software.

What are components?

- In unit testing, we are usually concerned with the most atomic behavioral units of a system.
- In procedural code, the units are often functions.
- In object oriented code, the units are classes.

Test harness *is a generic term for the testing code that we* write to exercise some piece of software and the code that is needed to run it.

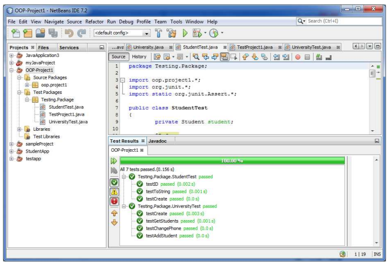
Unit testing



Testing in isolation is an important part of the definition of a unit test, but **why is it important**?

- **Error localization**
  - As tests get further from what they test, it is harder to determine what a test failure means.
  - Often it takes considerable work to pinpoint the source of a test failure.
- **Execution time**
  - Larger tests tend to take longer to execute.
  - This tends to make test runs rather frustrating.
  - Tests that take too long to run end up not being run.

Unit testing



Here are qualities of **good unit tests**:

- They run fast.
  - If they don't run fast, they aren't unit tests.
- They help us localize problems.

A test is **not a unit test** if:

- It talks to a database.
- It communicates across a network.
- You have to do special things to your environment (such as editing configuration files) to run it**.....**

A unit test that takes 1/10th of a second to run is a slow unit test !!!

Breaking Dependencies

To **put unit tests in place**, we often have to **deal with dependencies from one class to another.**

Mock objects for dependency breaking

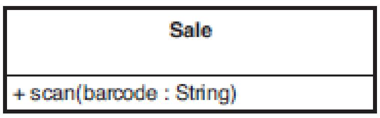
Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.

We can **create mock classes** and **objects manually**
OR
use a **Mocking framework** (e.g. Mockito, PowerMockito) **if there is one available** ….

For the example that follows see github code for JUnit and Mockito @ github.com/zarras

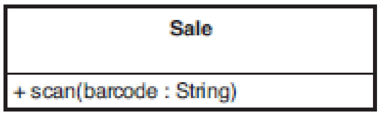
Mock objects for dependency breaking

In a point-of-sale system, we have a class called Sale. Whenever scan() is called, the Sale object needs to display the name of the item that was scanned, along with its price on a cash register display.



How can we test this to sense if the right text shows up on the display?

Mock objects for dependency breaking



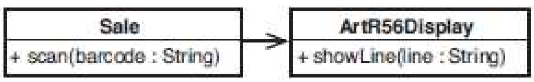
scan() is a void function so the expected results are visible only in the cash register display – it would be a good idea if we could mock the display with something that can be easily/automatically checked in a JUnit test

If the calls to the cash register's **display API are buried** deep in the **Sale** class, **faking is going to be hard**.

Mock objects for dependency breaking

We can move all of the display code from Sale over to ArtR56Display and have a system that does exactly the same thing that it did before.

Does that get us anything?



Mock objects for dependency breaking

- To use mock objects for dependency breaking:
  - Change the class of an irritating object such that it implements an interface.
  - Then change the TargetClass implementation (methods parameters, class attributes, etc.) such that it is based on objects that implement the interface, instead of the irritating objects.
  - This way you can create an object of the TargetClass for testing that uses **fake objects** that simply **implement the interface**, instead of real irritating objects.

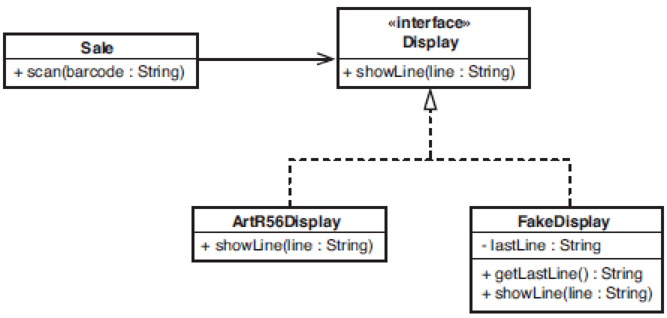
Instead of doing this work manually we can also use a **Mocking framework** (e.g. Mockito, PowerMockito) **if there is one available** ….

Mock objects for dependency breaking

The **Sale** class can now hold on to either an ArtR56Display object or something else, **a FakeDisplay object** that is used only for test purposes.

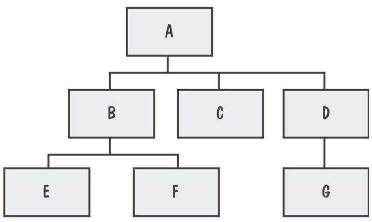
Instead of doing the work manually we can use **Mockito**.

See github code examples for **JUnit** and **Mockito** @ **github.com/zarras**



What is integration testing?

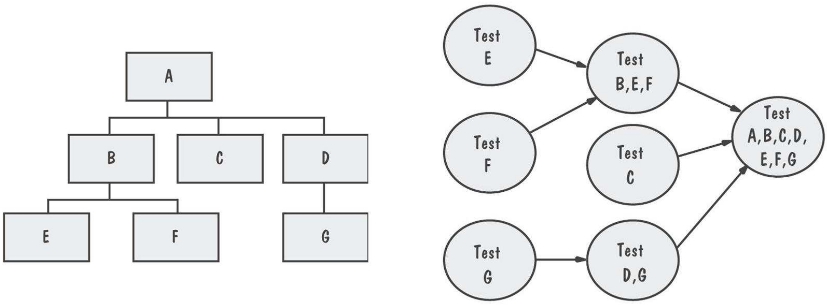
Integration testing



Integration testing is the process of verifying the interactions among software modules.

Classical integration testing strategies, such as top-down and bottom-up, are often used with hierarchically structured software to facilitate **error localization**.

Integration testing

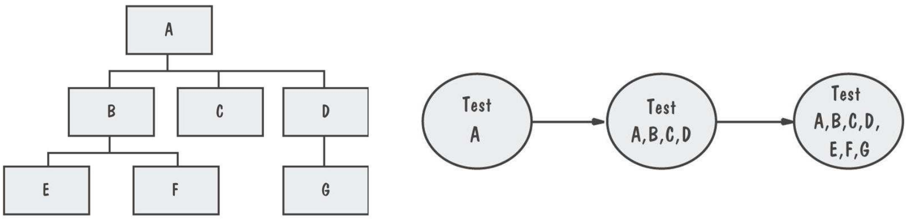


Bottom up strategy – does not require many fake objects

We test in **phases**:

We start testing in isolation simple – low level - components that do not depend on many others and we progressively add the ones that depend on them.

Integration testing



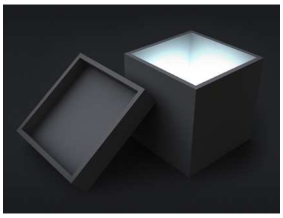
Top down strategy – requires more fakes but we find more complex problems early (e.g. faults in the core algorithm/process/ mechanism)

We test in **phases**:

We start testing in isolation the complex – high level - components and we progressively add the components that are used by them.

What is system testing?

System testing



System testing is concerned with testing the behavior of an entire system.

System testing is usually considered appropriate for assessing the non-functional requirements—such as security, performance, reliability, availability, usability

Testing techniques

How do we create test cases?

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.

Which is the most widely practiced technique?

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, …..

How about input domain based techniques?

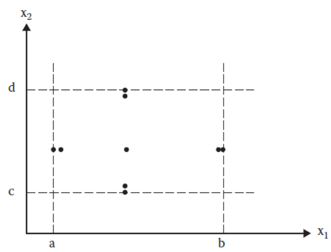
Input domain based techniques

Two widely know categories are:

Boundary value testing techniques.

Equivalence class testing techniques.

Normal boundary value technique

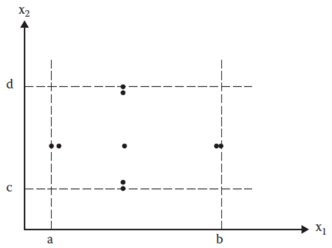


Boundary value analysis test cases for a function of two variables.

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.

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.

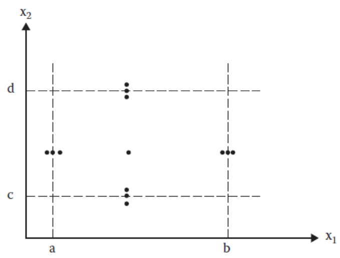
**** In general when we talk about parameters we refer to the test input data. So these could also be method parameters, object attributes, etc.…**

```
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;
    }
}
```

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

Robust boundary value technique

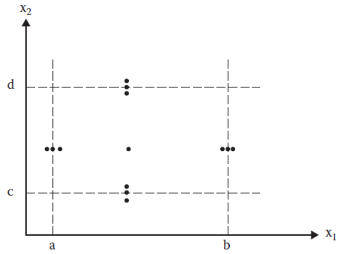


Robust boundary value analysis test cases for a function of two variables.

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–).

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, 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….**

```
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;
    }
}
```

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

Issues and limitations

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

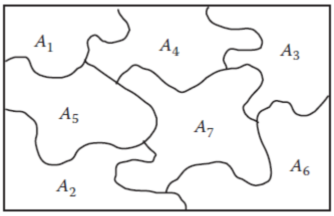
The paramters need to be described by a **true ordering relation**, in which, for every pair <a, b> of values of a parameter, it is possible to say that a ≤ *b.*

Test values for alphabet characters, for example, would be {a, b, m, y, 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.

Equivalence class testing



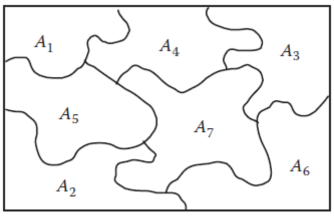
Math preliminaries

Given a set B, and a set of subsets A1, A2, …, An of B, the subsets are a partition of B iff

A1 ∪ A2 ∪ … ∪ An = B, and i ≠ j ⇒ Ai ∩ Aj = ∅.

Equivalence class testing

Math preliminaries



Suppose we have a partition A1, A2, …, An of B.

Based on this partition two elements, b1 and b2 of B, are related if b1 and b2 are in the same partition element.

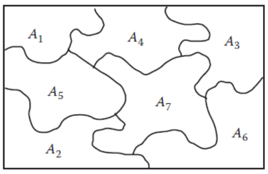
This is an equivalence relation because:

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

It is symmetric (if b1 and b2 are in a partition element, then b2 and b1 are)

It is transitive (if b1 and b2 are in the same set, and if b2 and b3 are in the same set, then b1 and b3 are in the same set).

Equivalence class testing



Equivalence class testing

Assume we test a function f with n inputs: v1, v2, …vn
Each input vi has a domain dom(vi)

Our target set B is the possibly infinite set of input tuples, i.e. B = dom(v1) x dom(v2) x … dom(vn)

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 Ai) the expected behavior of f is the same (although the **exact outputs may differ**).
➔In a sense we try to define classes (A1,A2, …) of expected outputs

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

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

For the triangle problem we can have the following partition:
B = A1 ∪ A2 ∪ A3 ∪ A4 ∪ A5
A1 = {a, b, c | wrong input} A2 = {a, b, c | not a triangle} A3 = {a, b, c | equilateral}
A4 = {a, b, c | isosceles} A5 = {a, b, c | 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

How about code based techniques?

Code based techniques

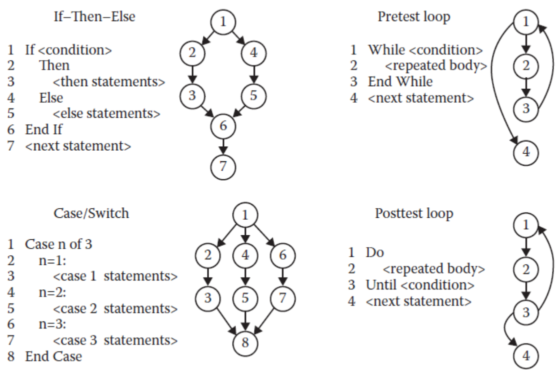
Two widely know categories are:

Control flow testing techniques.

Data flow testing techniques.

Which are the fundamental concepts of control flow techniques?

Control flow techniques



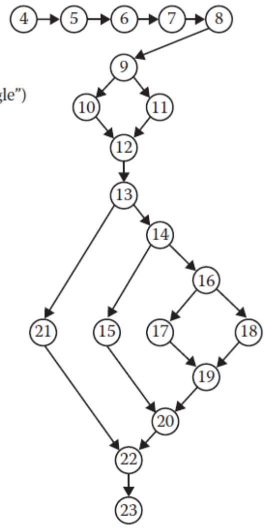
Program graphs of structured programming constructs

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.

Control flow techniques

```
Program triangle2
Dim a,b,c As Integer
Dim IsATrinagle As Boolean
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is", a)
Output("Side B is", b)
Output("Side C is", c)
If (a < b + c) AND (b < a + c) AND (c < a + b)
  Then IsATriangle = True
  Else IsATriangle = False
EndIf
If IsATriangle
  Then  If (a = b) AND (b = c)
        Then Output ("Equilateral")
        Else  If (a≠b) AND (a≠c) AND (b≠c)
                Then Output ("Scalene")
                Else Output ("Isosceles")
              EndIf
        EndIf
  Else  Output("Nota a Triangle")
EndIf
End triangle2
```

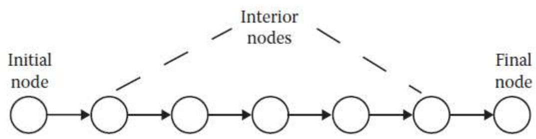


Control flow techniques

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

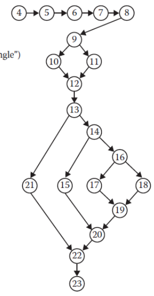
Starting for the program graph we produce the DD-graph as follows:

We keep the initial and the final nodes as is
We keep (decision) nodes with out degree >=2 as is
We keep nodes with in degree >=2 as is
We replace chains (with length >=2) of nodes with indeg = 1 and outdeg = 1 with a single node



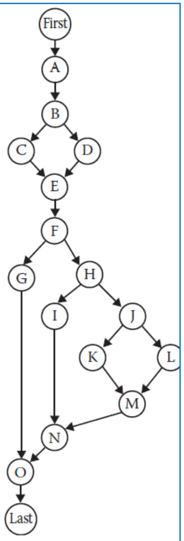
Control flow techniques

```
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 IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
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
```

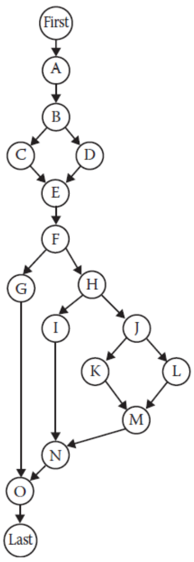


DD-path graph for triangle program

Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2



Statement testing

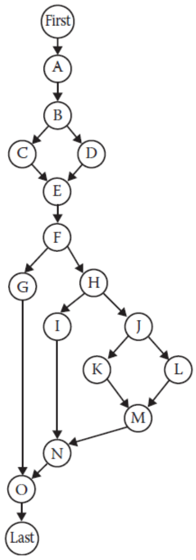


In statement testing our goal is to select a set of test cases T that satisfies the node coverage criterion.

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.

Branch testing



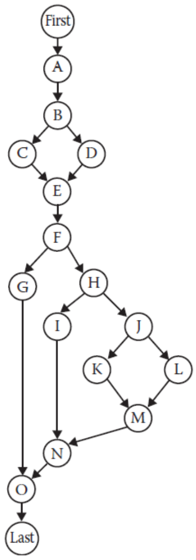
In branch testing our goal is to select a set of test cases T that satisfies the edge coverage criterion.

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.

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.