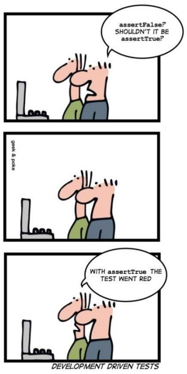


Software Testing

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
xUnit Patterns by Gerard Meszaros

Testing fundamentals

Why does software fail?

Why does software fail?

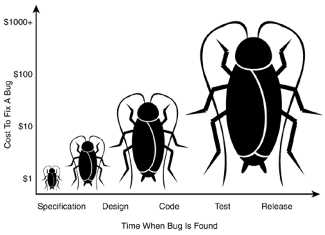
Bugs !!!

A more complete answer includes the famous triplet:

- Errors
- Faults
- Failures

What do we mean by error?

Errors



Error

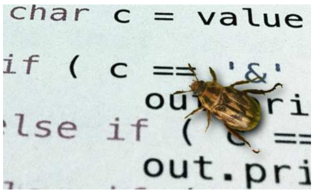
People make errors. A good synonym is mistake.

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**.

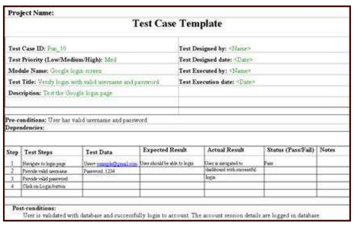
A test has two distinct goals:

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 Priority (Low/Medium/High): Med
Module Name: Google login screen
Test Title: Verify login with valid username and password
Description: Test the Google login page

Test Designed by: <Name>
Test Designed date: <Date>
Test Executed by: <Name>
Test Execution date: <Date>

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.

Why should we write tests?

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 …"

How do we write good tests?

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**.

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.

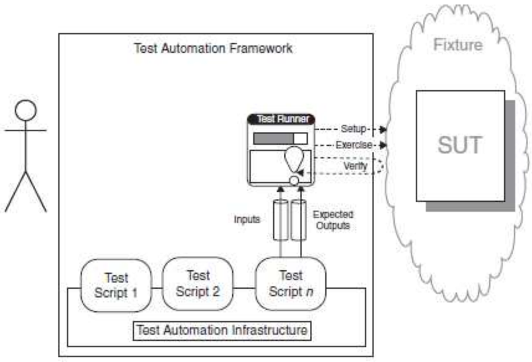
xUnit Basic Patterns

http://xunitpatterns.com/index.html

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

Test Automation Framework

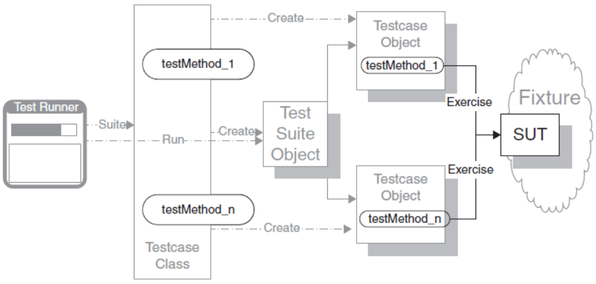
We use a framework that provides all the mechanisms needed to run the test logic so the test writer needs to provide only the test-specific logic.



Where do we put our test code?

Test Method

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



Variations:

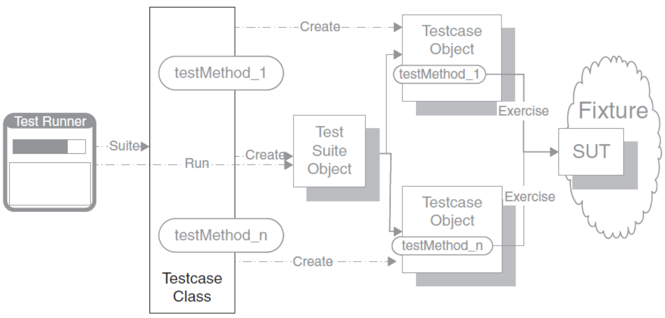
Simple success test (happy day)

Expected exception test

Constructor test

Test Case Class

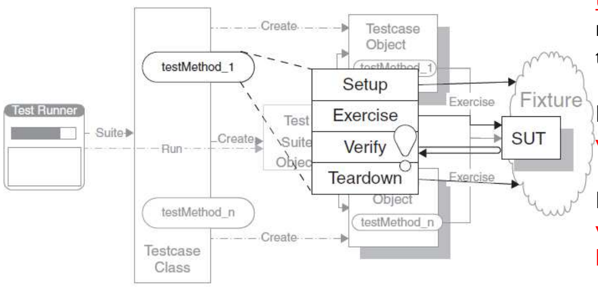
We group a set of related Test Methods on a single Testcase Class.



How do we structure our test code?

Four Phase Test

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



In the first phase, we set up the **test fixture** and anything we need to put in place to be able to observe the actual outcome .

In the second phase, we interact with the SUT.

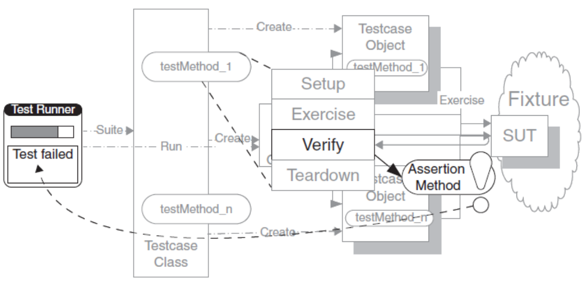
In the third phase, we determine whether the expected outcome has been obtained.

In the fourth phase, we tear down (clean up) the test fixture to put the world back into the state in which we found it.

How do we make tests self-checking?

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, …)

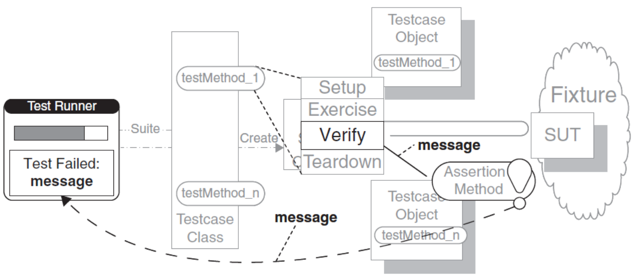
Expected exception assertions.

Single outcome assertions (fail)

How do we provide more information about a failed assertion?

Assertion Message

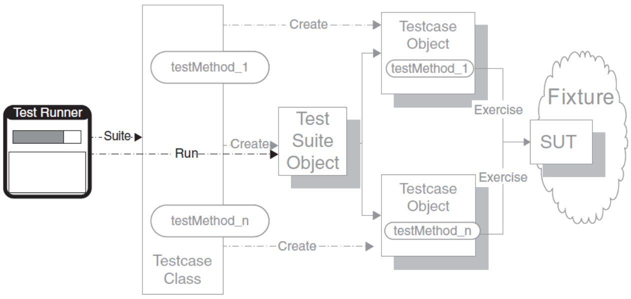
We include a **descriptive string argument** **in each call to an** **Assertion Method.**



How do we run the tests?

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.



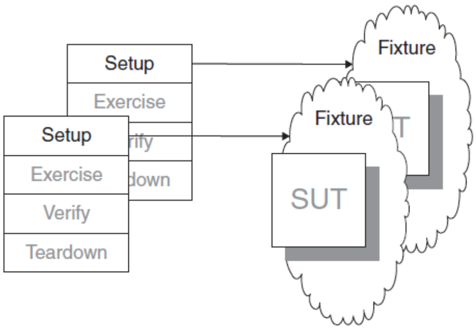
Fixture Setup/Teardown Patterns

What is a test fixture?

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

Fresh Fixture

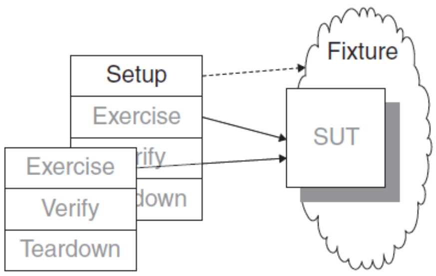
Each test constructs its own brand-new test fixture for its own private use.



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

Shared Fixture

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



If we want to avoid slow tests.

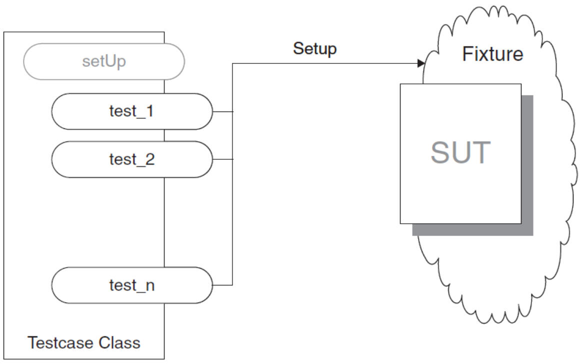
Or, when we have a long, complex sequence of actions, each of which depends on the previous actions. In customer tests, this may show up as a workflow; in unit tests, it may be a sequence of method calls on the same object.

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

How do we construct (destroy) a fresh fixture?

Inline Setup (Teardown)

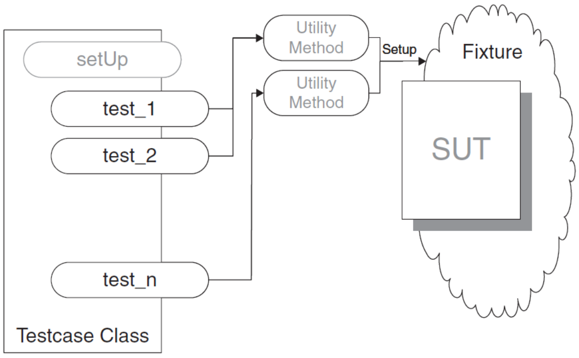
Each Test Method creates its own Fresh Fixture by calling the appropriate constructor methods to build exactly the test fixture it requires (the method destroys the fixture at the end).



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

Delegated Setup (Teardown)

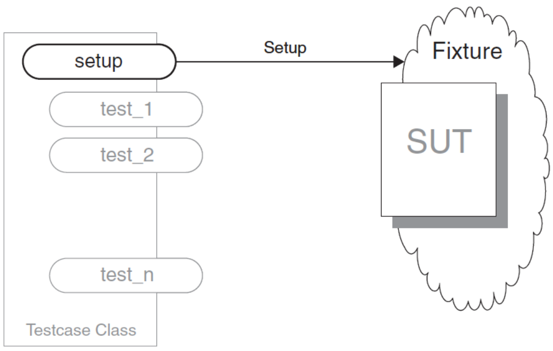
Each Test Method creates (destroys) its own Fresh Fixture by calling Creation/Destruction Methods from within the Test Methods.



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

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.

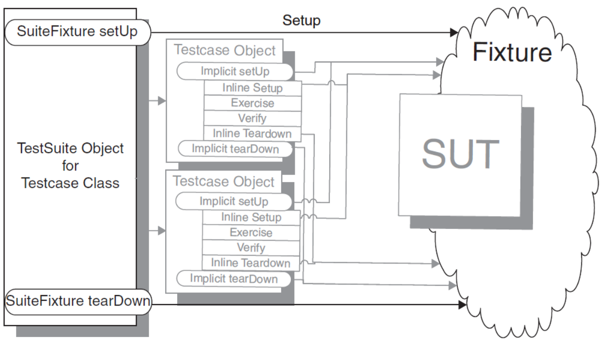


We can use Implicit Setup when several Test Methods on the same Testcase Class need an identical Fresh Fixture.

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

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

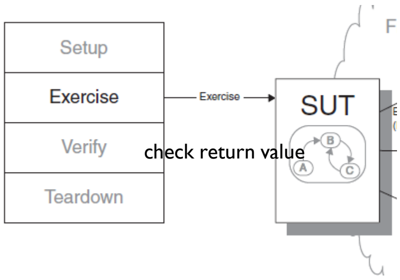


Result Verification Patterns

How do we verify a method that returns a value?

Return Value Verification

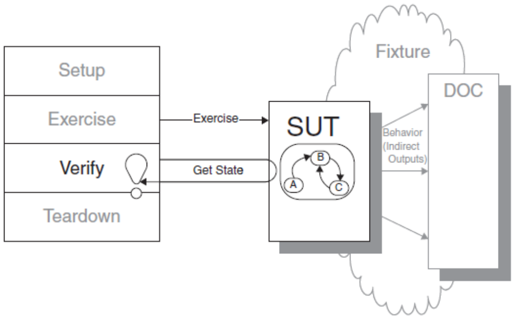
We inspect the returned value of the method and compare it with an expected return value.



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

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.

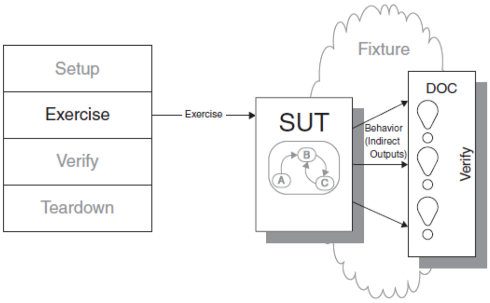
Expected State Specification, we construct a specification for the post-exercise state in the form of one or more objects populated with the expected attributes. We then compare the actual state with these objects.

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

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.



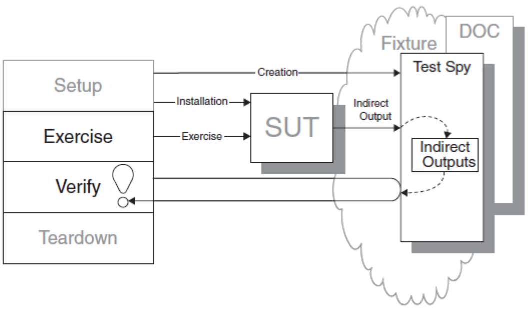
Typically to verify behavior we have to use some kind of a Test Spy or Test Mock (see Test Double patterns that follow)

Test Double Patterns

How do we verify the behavior of the SUT when it calls another component?

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:

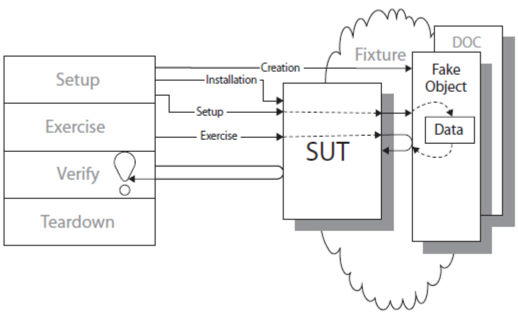
Use a mocking framework like mockito (spy() and verify() commands).

Subclass DOC and override the required methods to capture SUT calls. Configure the SUT with a test-specific object of the DOC subclass.

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

Fake Object

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



Fake Object implementation options:

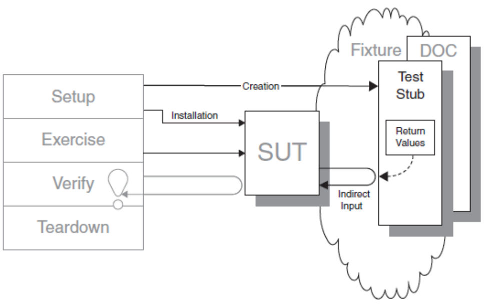
Use a mocking framework like mockito (mock() and when-then-return/throw commands).

If DOC implements an interface, configure the SUT with a **test-specific** DOC interface implementation.

How do we verify the behavior of the SUT when it gets indirect inputs from another component, independently from this component?

Test Stub

Use a test-specific object that feeds the desired indirect inputs into the system under test.



Variations:

Responder: a stub that feeds the SUT with valid (happy path) input.

Saboteur: a stub that feeds the SUT with invalid input.

Stub Implementation options:

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

If DOC implements an interface, configure the SUT with a **test-specific** DOC interface implementation.

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?

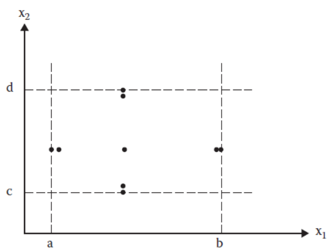
Input domain based techniques

Two widely know categories are:

Boundary value testing techniques.

Equivalence class testing techniques.

Normal boundary value technique

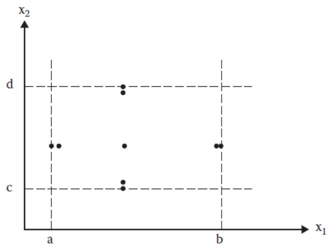


Boundary value analysis test cases for a function of two variables.

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.

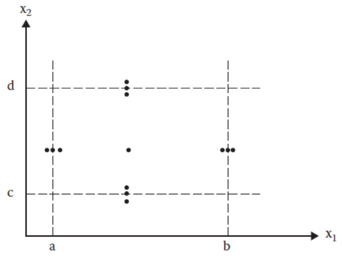
**** In general when we talk about parameters we refer to the test input data. So these could also be method parameters, object attributes, etc….**

```
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

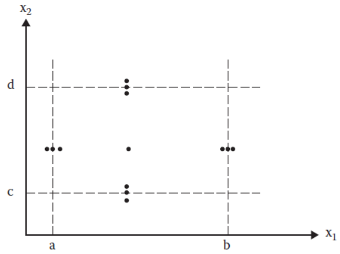


Robust boundary value analysis test cases for a function of two variables.

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.

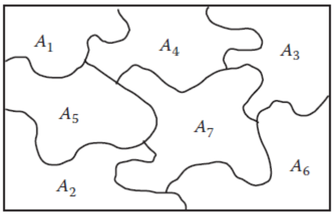
The parameters need to be described by a **true ordering relation**, in which, for every pair <a, b> of values of a parameter, it is possible to say that a ≤ *b.*

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



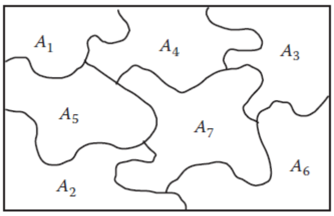
Math preliminaries

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 <u>related</u> if b1 and b2 are in the same partition element.

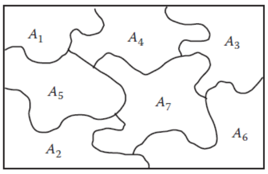
This is an equivalence relation because:

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

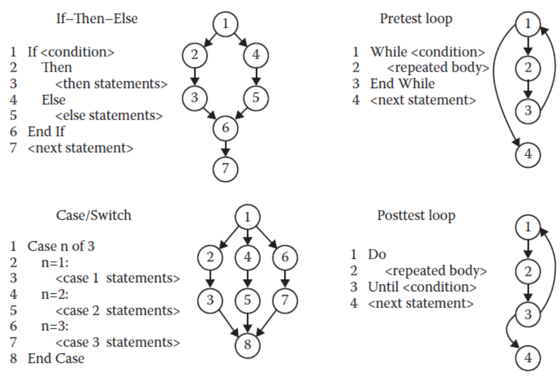
Two widely know categories are:

Control flow testing techniques.

Data flow testing techniques.

Which are the fundamental concepts of control flow techniques?

Control flow techniques



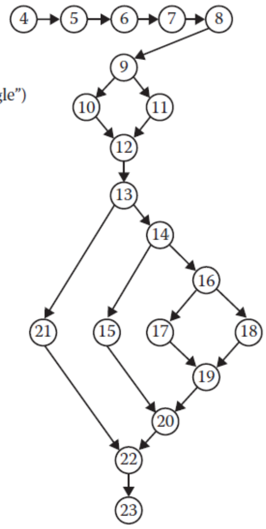
Program graphs of structured programming constructs

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

```
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
```

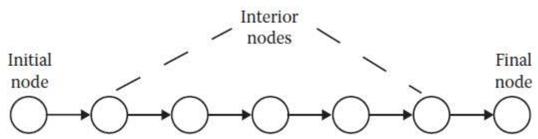


Control flow techniques

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

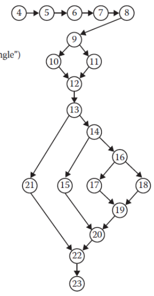
Starting for the program graph we produce the DD-graph as follows:

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



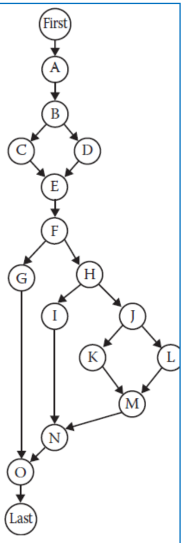
Control flow techniques

```
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
```

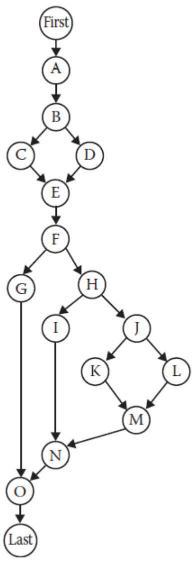


DD-path graph for triangle program

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

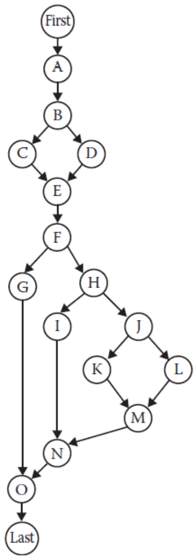


In statement testing our goal is to select a set of test cases T that satisfies the node coverage criterion.

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



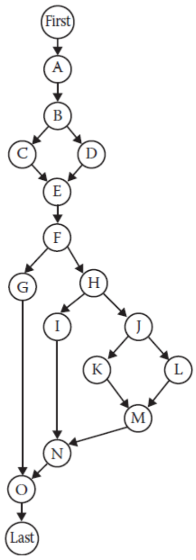
In branch testing our goal is to select a set of test cases T that satisfies the edge coverage criterion.

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.