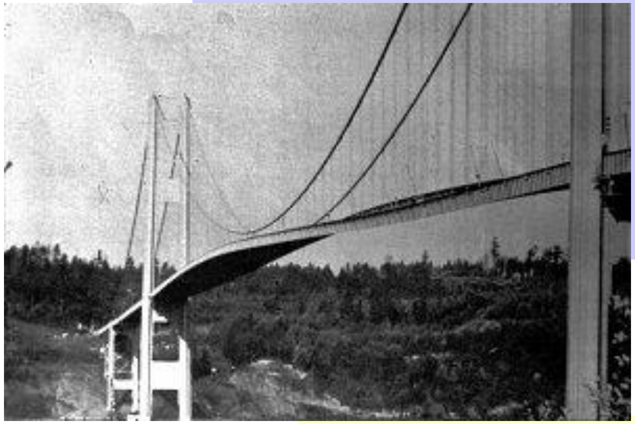
EC3301– Information Systems Design



Lecture 9 - Testing Principles

Lecture 10 - Testing Practice

- Dynamic testing
- Black box and White box testing
- Test planning
- Testing procedures

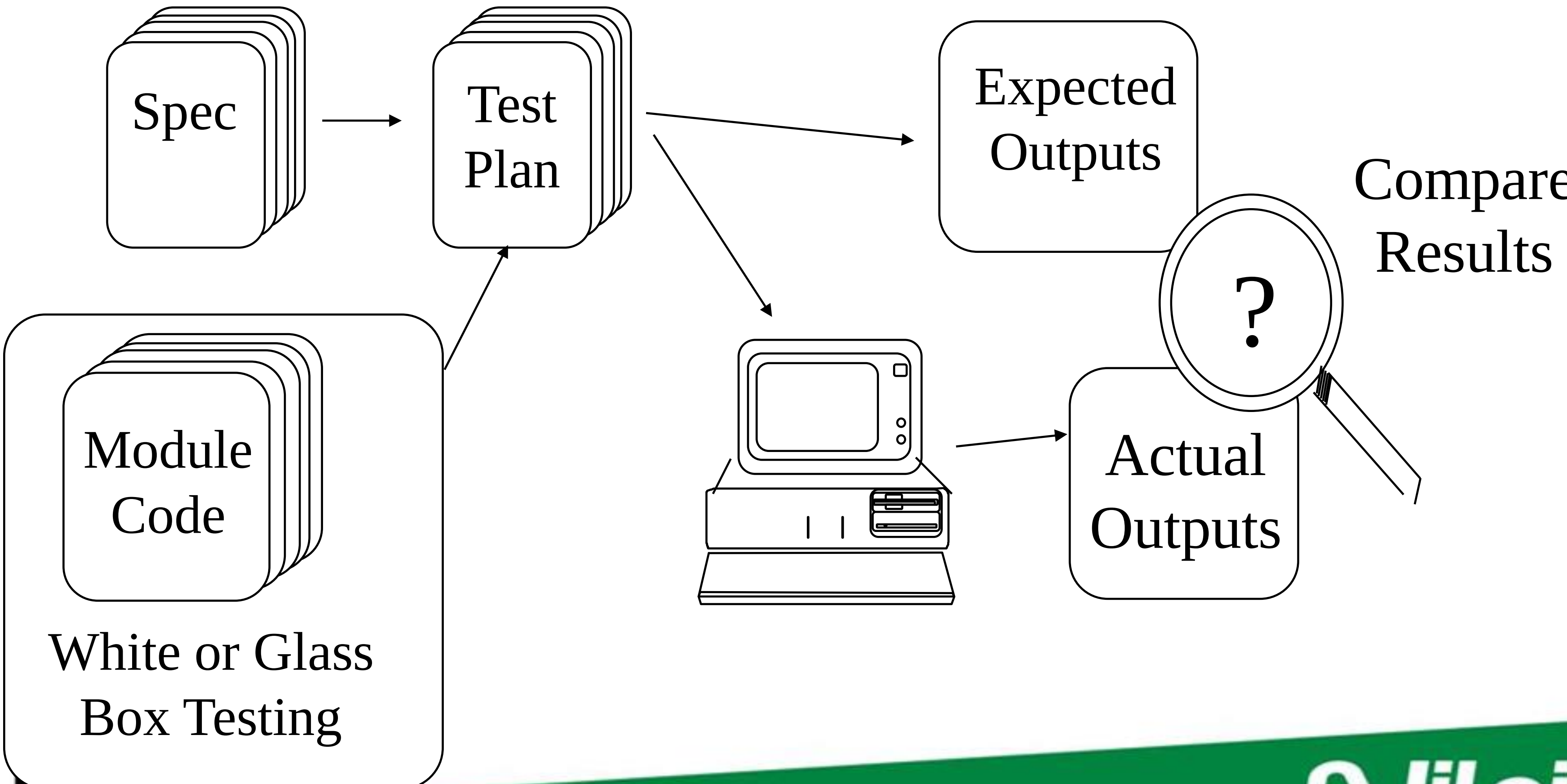
Types of Dynamic System Testing

- Function – modes of operation
- Load/Stress – robustness, reliability
- Volume/Performance – capacity, efficiency
- Configuration - portability
- Security – integrity, safety
- Installation – ease of installing, de-installed, upgraded
- Reliability – stability
- Recovery – fault tolerance
- Diagnostics - maintainability
- Human Factors – user friendliness

Dynamic Module testing

- **Test entire program, single module, procedure or code segment**
- **Test module code in isolation using a simulated environment provided by a test harness**
- **Test harness – a program designed specifically to test a module. The program inputs appropriate test inputs to and records outputs from the module under test.**
- **A separate test harness is required for each code module**

Dynamic Testing: General Procedure



Basic Elements of Dynamic Testing

The program under test

- Must be executable
- May need additional code to make it executable (e.g. libraries)

The test case

- The input data to run the program
- The expected output / dynamic behaviour (e.g. timing)

The observation

- The aspects of behaviour to be observed
- Means of observation (e.g. GUI or text file etc.)

The analysis of test results

- The correctness of behaviour
- The adequacy (e.g. coverage)

Black Box Dynamic Testing Techniques

- **Functional testing** – specific test cases defined to test each aspect of operation or system function, using a black box approach
- **Boundary value** – test performed at extremes of each input and output range, typically choosing values either side and on the boundary, to include both valid or invalid values.
- **Equivalence partitioning** – group sets of input and output ranges that can be treated in same way. Test performed on each set.
- **Performance testing** – examines the system behaviour in terms of resource utilization, e.g. cpu time, cpu complexity, memory or disk usage, network or I/O requirements - in normal and stressed processed conditions
- **Random testing** – is functional or structural testing in which it has been decided to test some random sample of tests or input vectors. An effective random test will match the inputs expected during system operation
- **Error seeding** – introduce error, as a check on the testing process
- **Error guessing** – predict error conditions where test cases based on possible operation situations

Black Box Dynamic Testing - example

- Test module against its (external) specification, i.e. check module outputs
- No knowledge of internal code

```
e.g. Function dodgy_product(x,y:integer):integer;
     { calculate product of integers x and y }
     Var product:integer;
     Begin
       product:=x*y;
       if product=42 then
          product:=24;  { sabotage !}
       dodgy_product:=product
     End
```

With black box testing we cannot see the module internal code

Black box testing (test plan) with random data values from input domain:

```
Writeln( dodgy_product(4,5));    { is ok: expected=20, observed=20 }
Writeln( dodgy_product(5,6));    { is ok : expected=30, observed=30 }
Writeln( dodgy_product(6,7));    { is NOT ok : expected=42, observed=24 }
...
```

Test scripts

Boundary Analysis

The purpose:

- To test if the boundaries implemented by the software are correct

The method:

- Select test cases on and around the borders

The basic assumptions:

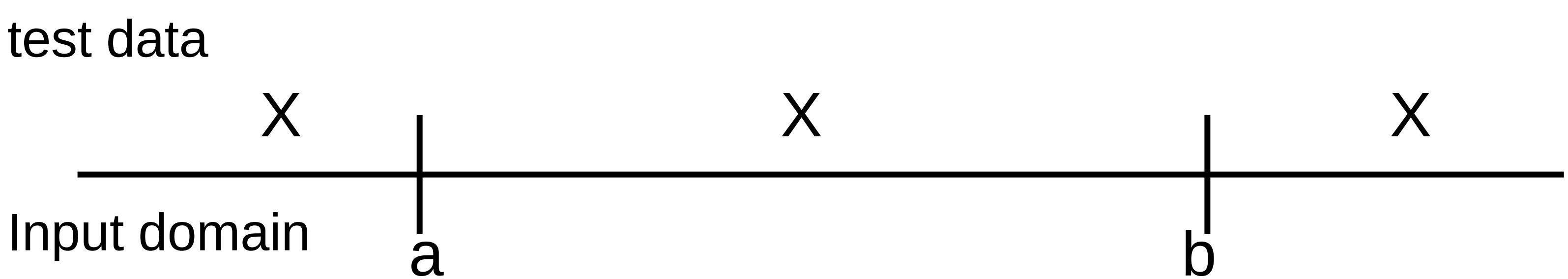
- The software computes different function on points inside the sub-domain from the points outside the sub-domain
- Domain is decomposed into sub-domains by borders, which are simple, such as straight lines and planes
- Boundary errors are simple, such as shift errors and rotation errors
- Errors arise frequently from >, >= and < <= confusion/ambiguity

Boundary Value Testing

- Aim is to detect errors relating to the input domain
- Basis of the technique validity is that program errors are frequently associated with range boundary values, e.g:
  - Use of < where ≤ should be used
  - In C `int a[10];` defines elements 0 to 9, reference to `a[10]` is a common programming error

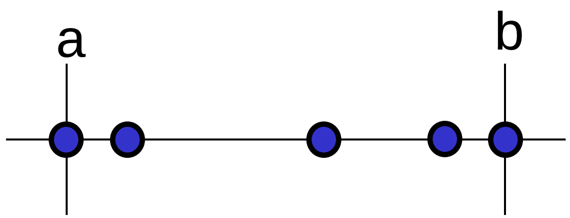
Equivalence partitioning

- Black box technique based on category sets of inputs
  e.g. input domain [a, b]



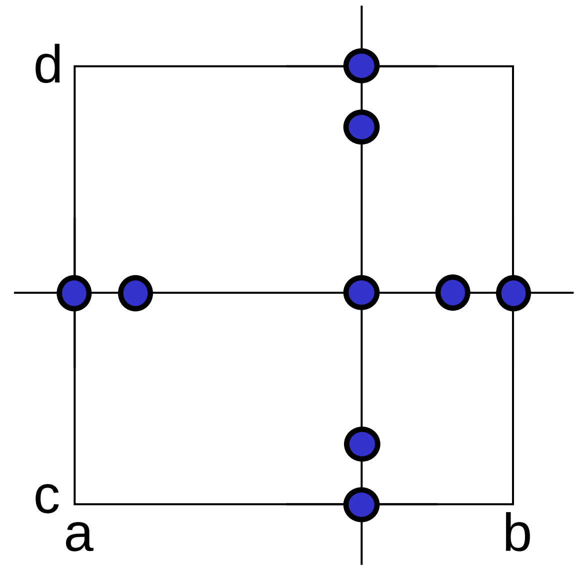
Basic Boundary Testing Model (1)

- Consider a data input x with domain range [a, b] and y from [c, d]



1-D

Number of tests: 5

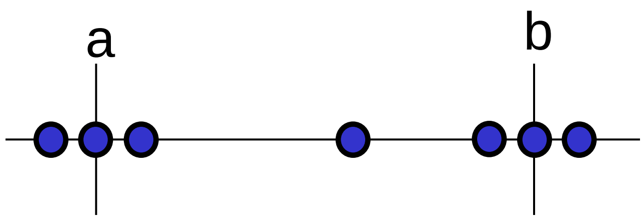


2-D

Number of tests: 9

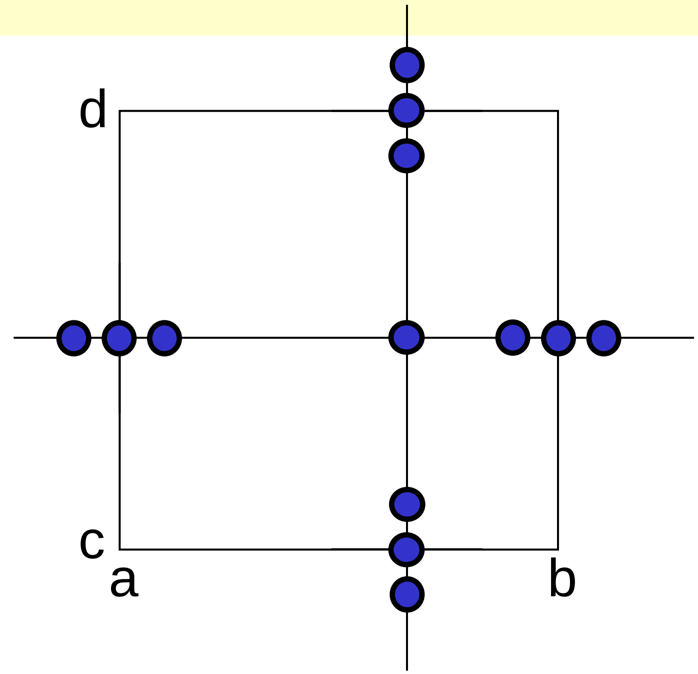
Robustness Boundary Testing Model (2)

- Consider a data input x with domain range [a, b] and y from [c, d]



1-D

Number of tests: 7

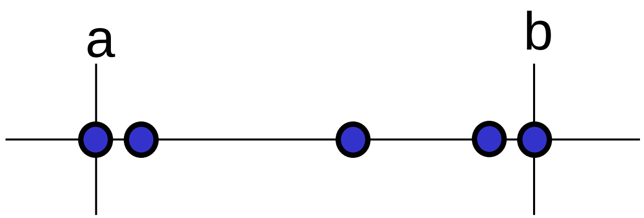


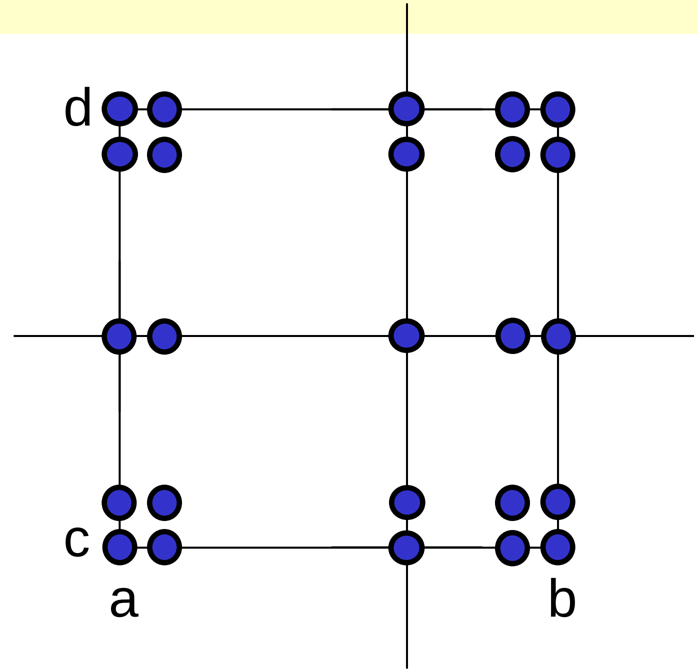
2-D

Number of tests: 13

Worst Case Boundary Testing Model (3)

- Consider a data input x with domain range [a, b] and y from [c, d]





1-D

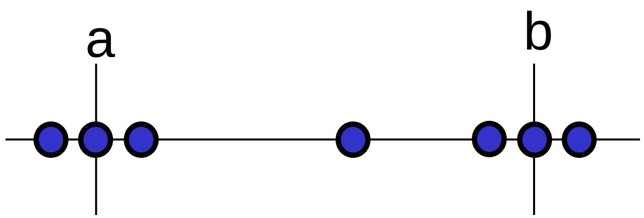
Number of tests: 5 (5^1)

2-D

Number of tests: 25 (5^2)

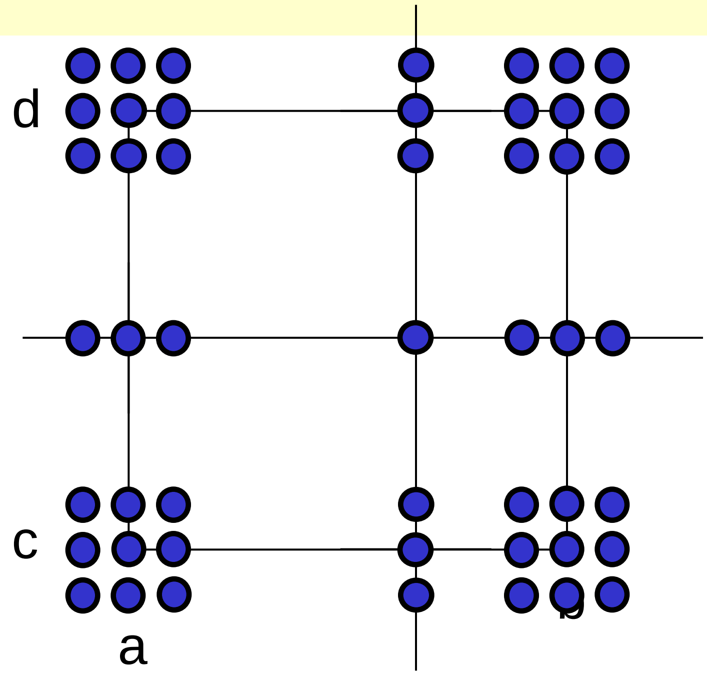
Worst Case Robust Boundary Testing Model (4)

- Consider a data input x with domain range [a, b] and y from [c, d]



1-D

Number of tests: 7 (7^1)

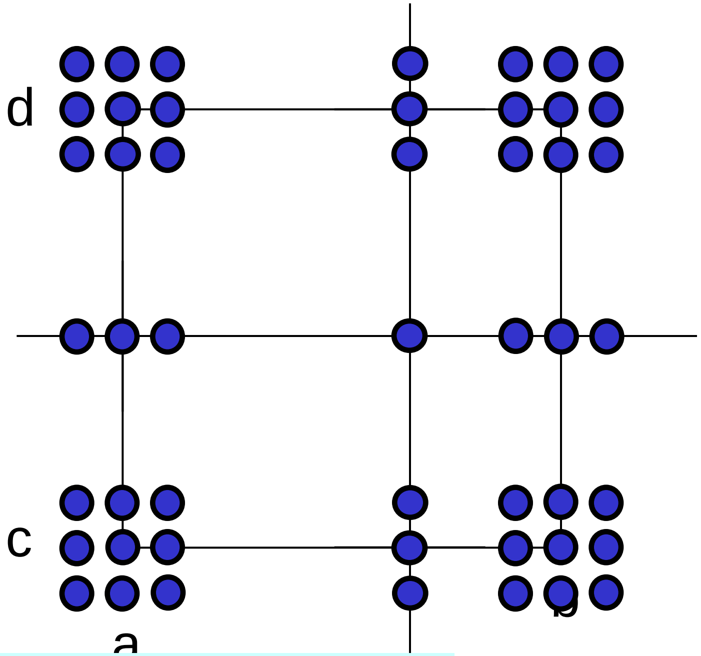


2-D

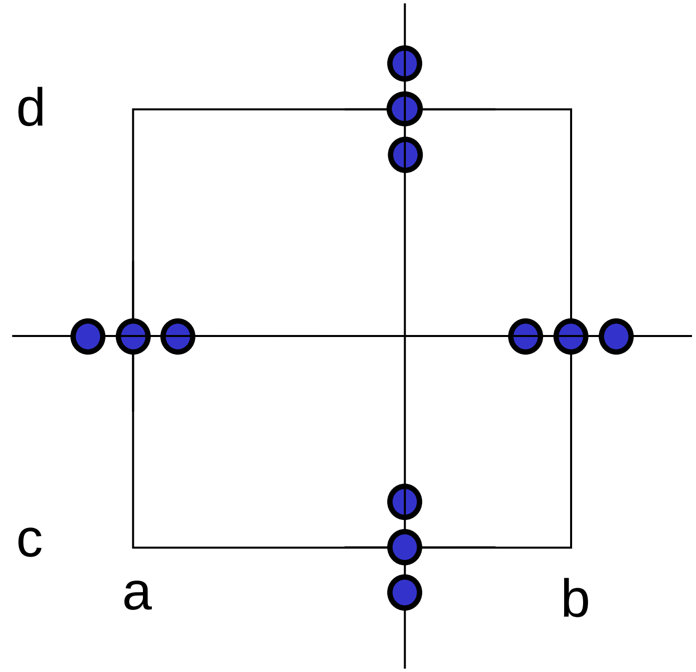
Number of tests: 49 (7^2)

Adequacy Criteria (using white box static testing)

- Previous basic, boundary, robustness and worst case testing strategies are intuitive but are over often adequate – not all the tests are required
- For Adequate Testing consider the minimum subset of tests that cover each boundary condition
- So in the case of the previous example, e.g. for testing for point inside a rectangle:



Worst Case Robust Boundary Testing no. of tests: 49

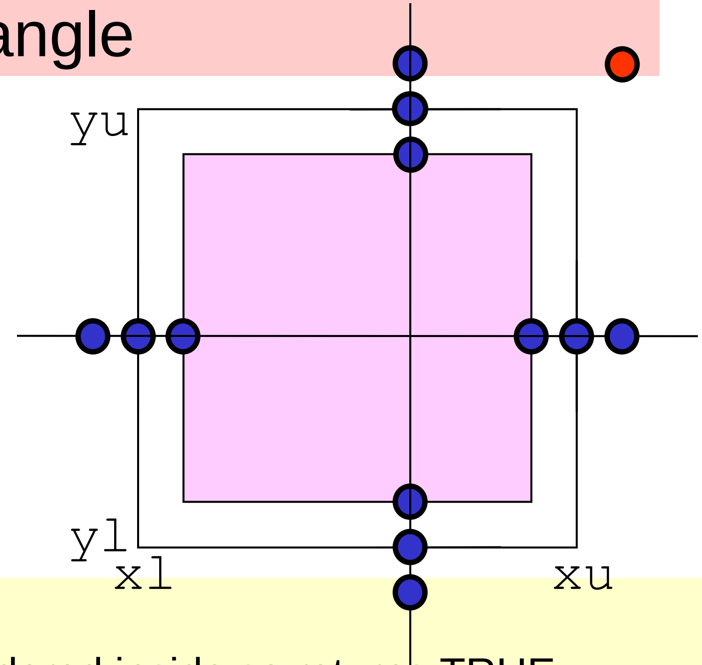


Adequacy Criteria - no of tests: 12

Where x and y testing is independant

Adequacy Criteria for point inside rectangle

```
Integer coordinates
Inside_X = x>xl AND x<xu
Inside_Y = y>yl AND y<yu
Inside = Inside_X AND Inside_Y
```

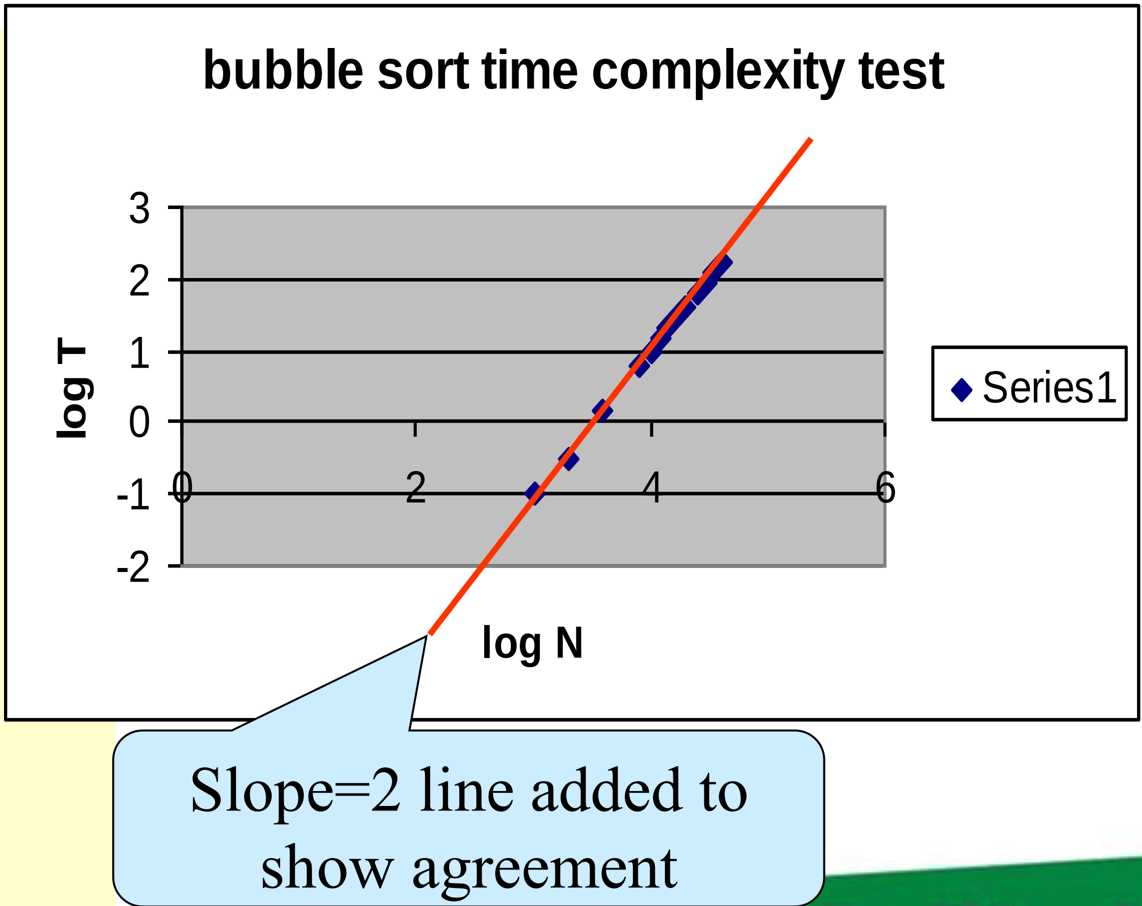


- Rectangle is (`xl,yl,xu,yu`), point is (`x,y`)
- Rectangle region (`xl+1,yl+1,xu-1,yu-1`) is considered inside so returns TRUE
- Other regions are either on the edge, or outside so return FALSE
- Rationale for tests: e.g. `x>xl` could be miscoded in several ways. It is important that this predicate returns F when `x<xl`, F when `x=xl` and T when `x>xl`
- Similar for the other 3 predicates, so 3*4=12 (blue spot) tests
- `x` any `y` expressions are linked by the AND so each of the 4: FF,FT,TF,TT combinations needed. FT,TF and TT already considered but additional FF (red spot) not needed as the result will be identical for AND or OR coding.
- So full worst case robustness test combination is not needed (in this case)

Performance testing example: CPU time performance of C bubble program where $T = AN^2$

Measure bubble sort cpu time

```
rainbow% /usr/bin/time ./sort 1000
user      0.1
rainbow% /usr/bin/time ./sort 2000
user      0.3
rainbow% /usr/bin/time ./sort 4000
user      1.5
rainbow% /usr/bin/time ./sort 8000
user      6.4
 . . .
 . . .
rainbow% /usr/bin/time ./sort 40000
bubble sort - N=40000
 real    2:44.0
user    2:43.6
sys       0.0
```



Performance Testing

- Load testing – testing under realistic or worst case or projected load conditions
- Failure Testing – test system, redundancy mechanisms in the case of individual or multiple component failure
- Soak Test – run system at high load for extended period
- Stress Test – determine work load for system to fail (load can be ramp, step or accelerated)
- Benchmarking – determine CPU memory or other system statistic as a function of job size (benchmarking often used for comparison purposes)
- Volume Testing – testing to assess transaction, message or response rate)

Random Testing

Random testing uses test data selected at random according to a probability distribution over the input space

- **Representative random testing**
  - The probability distribution use to sample the input data represents the operation of the software, e.g. data obtained in the operation of the old system or similar systems
- **Non-representative random testing**
  - The probability distribution has no-relationship with the operation of the system
- **Advantages**
  - Reliability can be estimated especially when representative random testing is used
  - Low cost in the selection of test cases, which can be automated to a great extent
  - Can achieve a high fault detection ability
- **Disadvantages**
  - Less confidence can be obtained from the testing
  - Still need to validate the correctness of output, which may be more difficult than deliberately selected test cases.

White Box Dynamic Testing

- **Statement coverage (every line)** - aim is to create enough tests to ensure every statement is executed at least once
- **Decision coverage (every decision)** - aim is to generate tests to execute each decision statement branch and module exit path. For example provide tests to exercise both **true** and **false** branches in IF statements. More rigorous than statement coverage
- **Structural analysis (every control path) -** tests the complete program's structure. It attempts to exercise every entry-to-exit control path but in large and complex programs the number of different control paths makes this approach prohibitive. In this case statement and decision coverage must be considered
- **Data value analysis (every data value)** - Identify numerical problems: entry of incorrect data type or value, divide by zero, overflow etc.

Problem: exhaustive (100% coverage) testing often impractical, e.g. 32 bit binary inputs tested at 1 test/ms will take 46 days

White Box Dynamic Testing Example

```
e.g.   If (A>1) AND (B=0)
         C:=A
       Else
         C:=B
```

For a full structural analysis consider the IF statement branch including the two predicate conditions. Consider true and false possibilities for each predicate.
4 tests as follows:

	B=0	B<>0
A>1	Test 1	Test 2
A<=1	Test 3	Test 4

Choose data values for each test

Test 1 A=2 B=0 Expected output C=A (=2)

Test 2 A=2 B=3 Expected output C=B (=3)

Test 3 A=-1 B=0 Expected output C=B (=0)

Test 4 A=-1 B=3 Expected output C=B (=3)

forms the test plan

Functional Testing

- Derive test cases from the system or component (black box) specification
- Check for correctness by executing each system function and examining the output or behaviour
- Generally a black box approach is taken
- Specification can be formal (e.g. Z, CSP etc.) or informal (e.g. UML, natural language)

Function of a software system:

- Required functions
- Specified functions
- Designed functions
- Implemented functions

Are these equivalent?

Functional Testing - Basic Concepts

Function:

- The relationship between the input and its output / behaviour

Domain (the input space)

- The set of valid input values

Codomain (the output space)

- The set of possible output values.

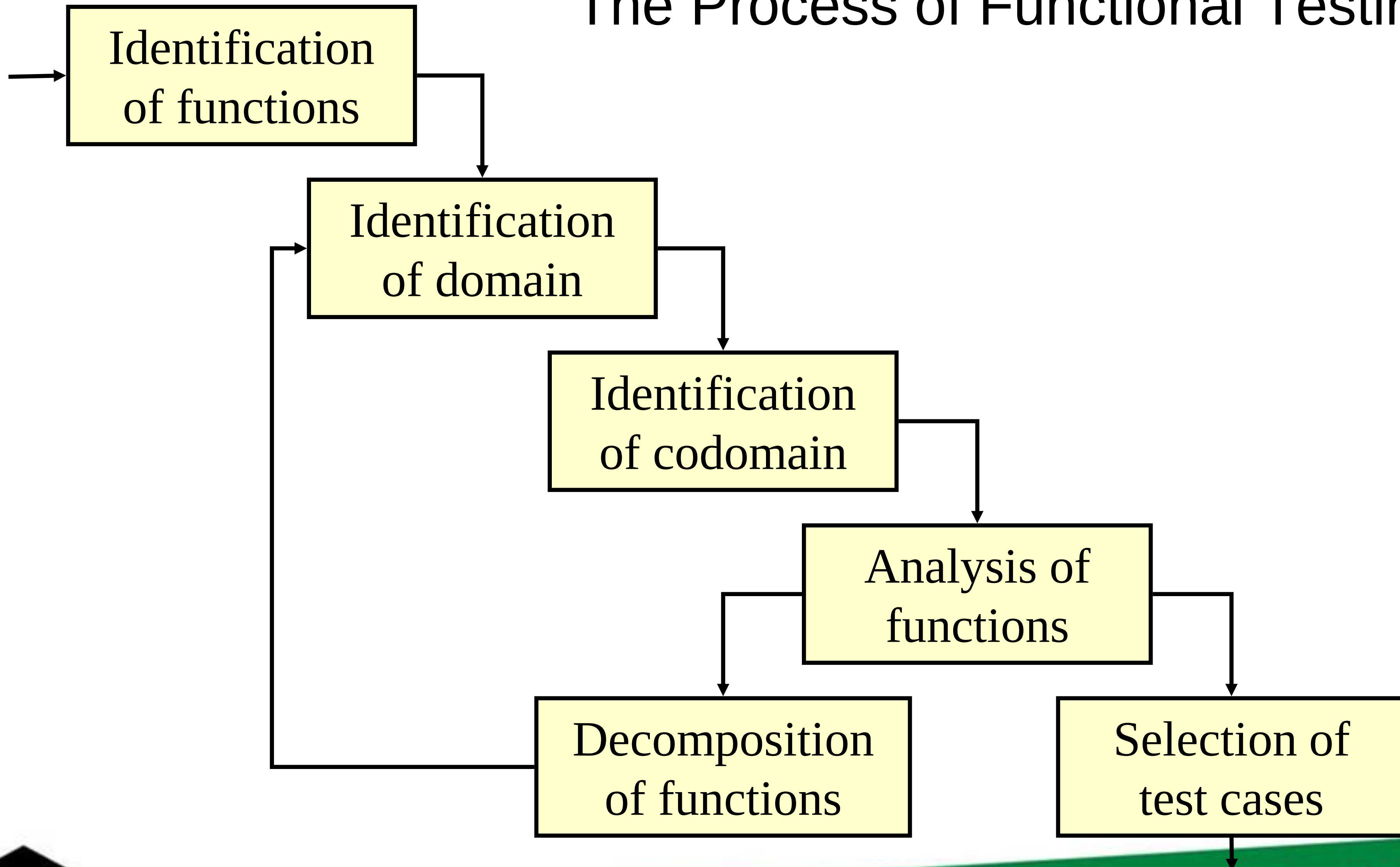
Dimension of domain

- The number of independent input variables

Boundary

- The lines/planes that specify the domain space when the inputs are in a continuous data set

The Process of Functional Testing



Activities of Functional Testing

Identification of functions

- What is the function to be tested?

Identification of domain

- What is the input space for each function?
- What is the dimension of the input space?
- What are the boundaries for each function?

Identification of the codomain

- What is the output space?

Analysis of the function

- What is the relationship between the domain and codomain?
- Can it be decomposed into simpler functions?

Decomposition of the function

- What are the components of the function?
- How are the components organised?

Selection of test cases

- What input data can prove or disprove that the software implements the boundary correctly?
- What input data can prove or disprove that the software implements the relationship correctly?

Example: Discount Invoice

A company produces two items, X and Y, with prices £5 for each X purchased and £10 for each Y purchased. An order consists of a request for a certain number of X's and a certain number of Y's.

The cost of the purchase is the sum of the costs of the individual items discounted as follows:

- If the total is greater than £200 a discount of 5% is given,
- If the total is greater than £1000 a discount of 20% is given.
- The company wishes to encourage sales of X and offers a further discount of 10% if more than thirty X's are ordered.

Note: Only one discount rate will apply per order, and non-integer final costs are rounded down to give an integer value, e.g. Int(3.6) returns 3.

Discount Invoice: problem analysis

Identification of function

- The function to be tested is the computation of the total invoice amount for any given order

Identification of domain

- The input space consists of two inputs:

  x: the number of product X ordered

  y: the number of product Y ordered

- Both inputs are non negative integers

Identification of codomain

- The output (sum) is an integer that represents the order cost in pounds

Analysis of functions

- The relationship between the input and output is ……, too complicated, hence we need to decompose it! . . . .

Discount Invoice: Decomposition

Case 1: If inputs x and y have the property that ($x \leq 30$ and $5x+10y \leq 200$), the output should be 5x + 10y.

Sub-function 1:

- Sub-Domain: $A = \{(x, y) \mid x \leq 30, 5x+10y \leq 200\}$
- Relationship: sum = 5x+10y

A is the Set of x,y such that x<=30 and 5x+10y<=200

Case 2: If inputs x and y have the property that ($x \leq 30$ and $5x + 10y > 200$), the output should be (5x + 10y)*0.95, i.e. a 5% discount

Sub-function 2:

- Sub-Domain: $B = \{(x, y) \mid x \leq 30, 5x+10y>200\}$
- Relationship: sum = Int(0.95*(5x+10y))

Cotd.

Discount Invoice: Decomposition (cotd.)

Case 3: If inputs x and y have the property that ($x>30$ and $5x+10y\leq1000$), the output should be (5x + 10y) less 10% discount

Sub-function 3:

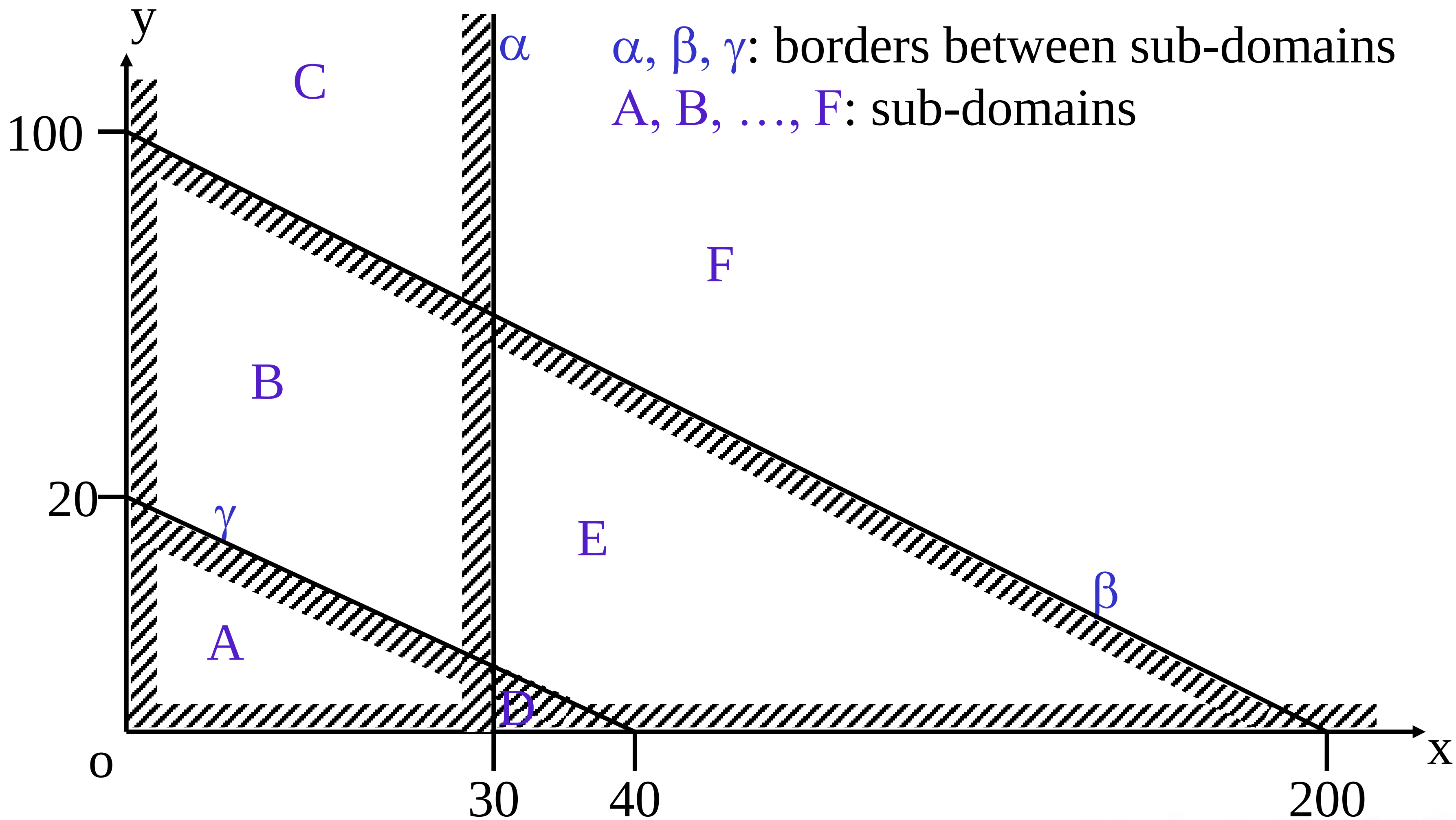
- Sub-Domain: D and E = $\{(x, y) \mid x > 30, 5x+10y\leq1000\}$
- Relationship: sum = Int(0.9*(5x+10y))

Case 4: If inputs x and y have the property that ($5x+10y>1000$), the output should be (5x + 10y) less a 20% discount

Sub-function 4:

- Sub-Domain: C and F = $\{(x, y) \mid 5x+10y>1000\}$
- Relationship: sum = Int(0.8*(5x+10y))

Sub-Domains



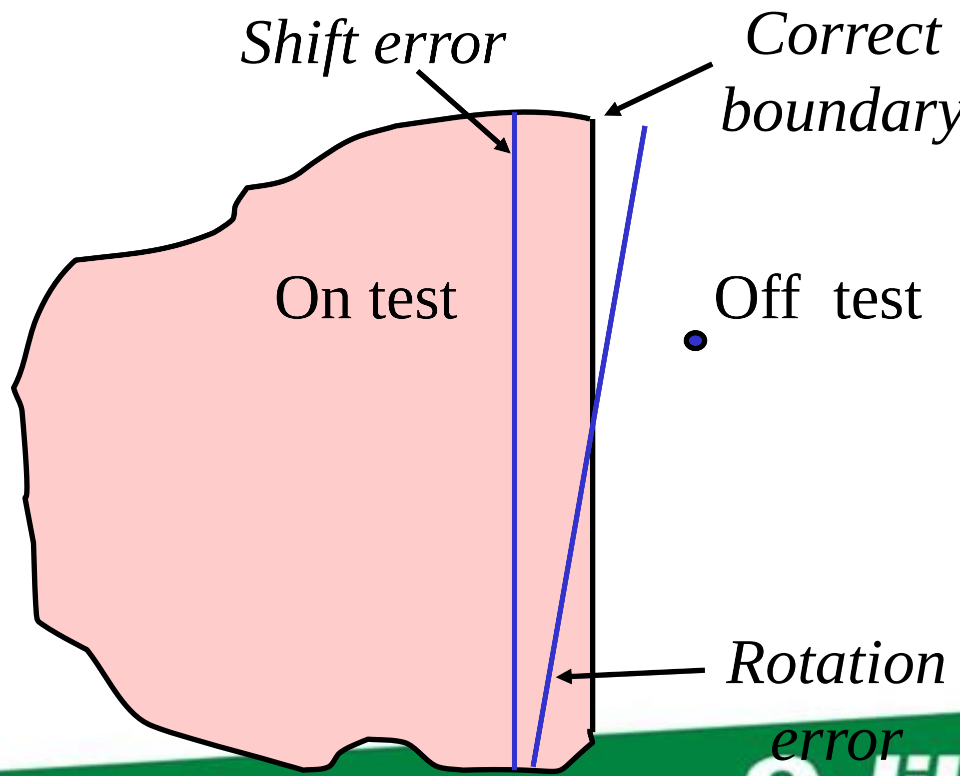
α, β, γ: borders between sub-domains

A, B, …, F: sub-domains

Definition of Terminology

- **On test** - The test case whose input is inside the sub-domain
- **Off test** - The test case whose input is outside the sub-domain

Shift error
The implemented boundary is a parallel shift from the correct boundary
Rotation error
The implemented boundary is a rotation of the correct boundary



The Generic Testing Process

Test Planning

Test Specification

Test Execution

Test Results Recording

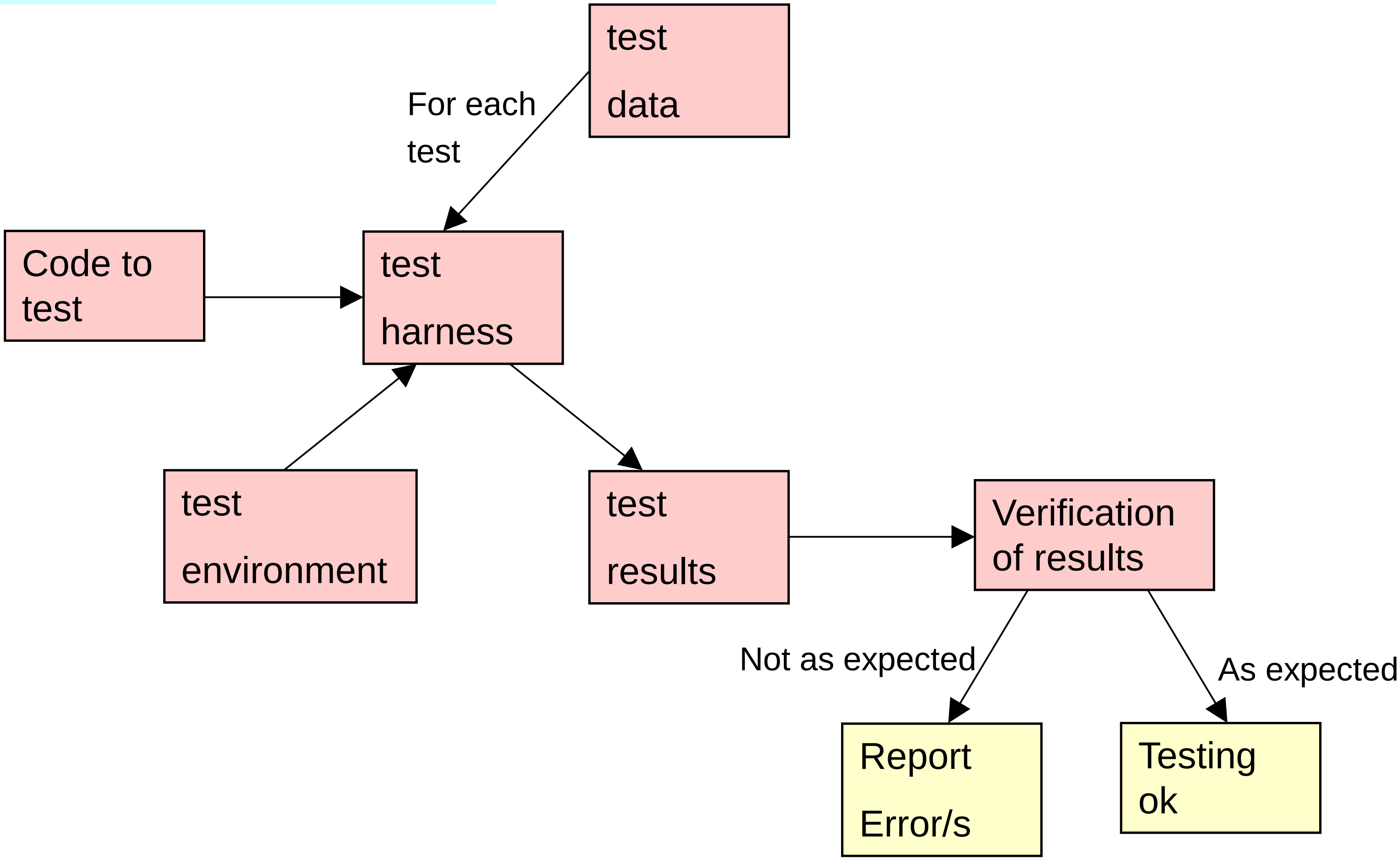
Test Evaluation

1. What to test
2. How to test
3. Record of tests
4. Test evaluation

process

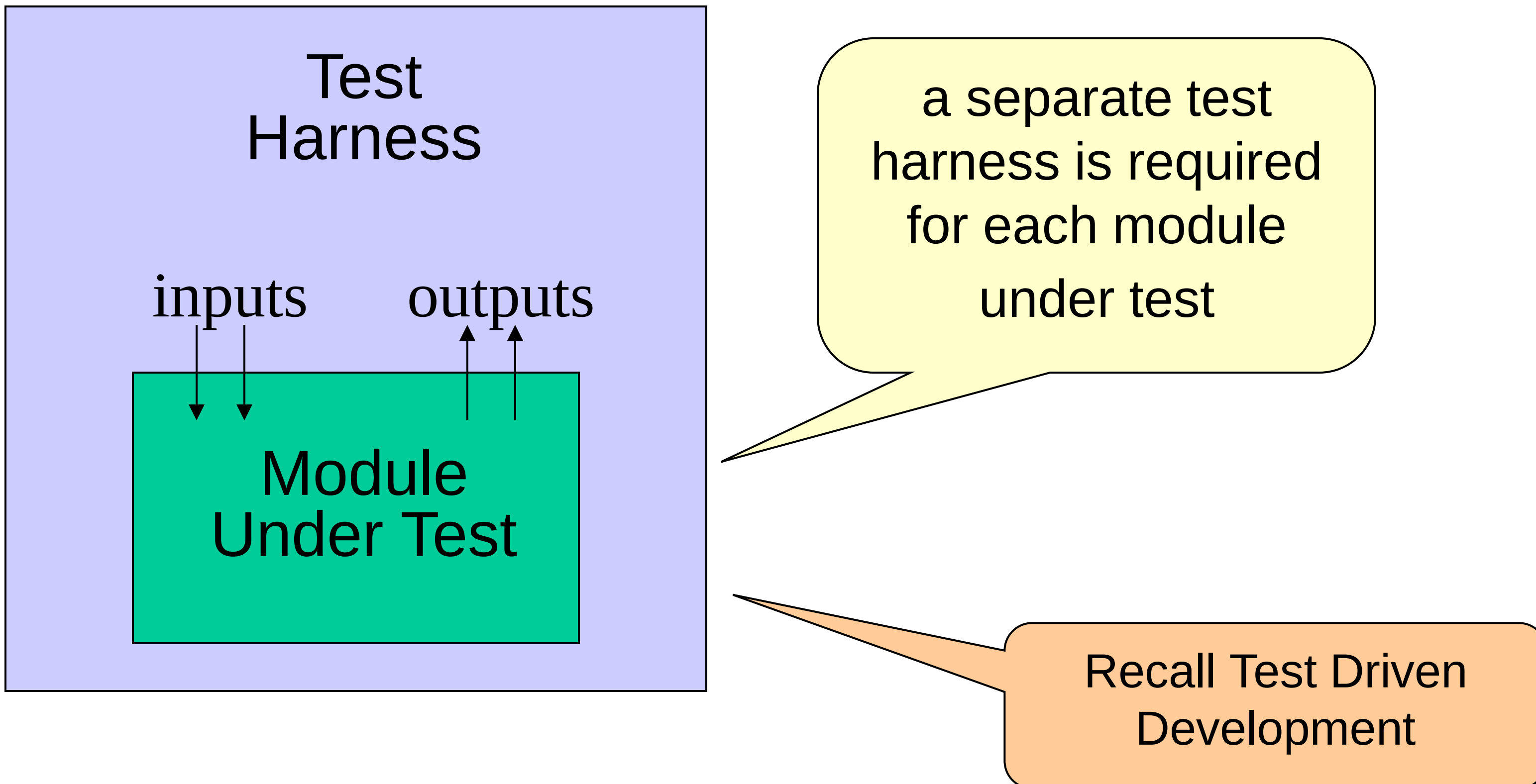
tasks

Generic Testing



The Test Harness (program)

- provides simulated test environment to apply the required inputs and capture outputs for a module under test.



Testing: the basis of testing is the <u>Test Plan</u>

A **Test Plan** for a module, component or system will document:

- Details of the part of the system being tested and the objectives of testing, e.g. in relation to quality standards
- The general testing strategy: specify the test methods, testing evaluation criteria
- Hardware and software dependencies
- Date, location and individual/s undertaking the testing
- For each test, include:
  - Details and purpose of test **(what program/module, what level of testing, scenario or test case)**
  - Test data input and expected output **(how to test)**
  - How the test data is to be prepared and submitted too the system
  - How the outputs are to be captured **(record raw test output)**
  - How the results will be analysed **(test results**)
  - Any other operational procedures
- The test plan forms an integral part of the software life cycle design process

Test Harness, Stub and Test Script

Test Harness

A program written to call the code to be tested, such as a procedure, function or a module of program. The objective is to test the code in isolation.

Stub

A piece of program code written to replace the modules or procedures that the program under test depends on and calls so that it can be executed.

Test Script

Some test tools can support the generation of such code, but the tester may need to describe the environment. Such description is usually called test script.

Dynamic Testing: Record of Testing

– full details of each test should be recorded (e.g. as a UNIX script file)

Component under Test	Test: abc123 Mod: ValDate (Date Validation Module) Source: DATETIME.COB
Purpose of Test	Purpose: Leap Year checking
Cases Covered	Covers: 29/2/ccnn where cc is 19 or 20 and nn is 00 - 99
Data/System Setup	Setup:None
User input	To Run:Enter 'abc123‘
Expected Results	Results:'abc123: Passed' on success otherwise details of failure

Test Plan Document Layout

Introduction

- summary from requirements specification

Requirements Identification

- taken from requirements specification verification section
- identifies what aspects are to be tested

Test Plan / Procedures (overall discussion of testing strategy)

- develop a minimum number of tests which cover all the requirements
- each case (scenario) requires details of the hardware and software setup, input required together with the expected behaviour/output and how this can be observed. The use case documentation can form the basis for this

Test Results

- table listing test scenarios, software version, results observed, signature of observer

Traceability Matrix

- relate each requirement scenario to the test result evidence

Design Test Cases

List test cases

- For each function (use case) check the scenarios
- Each generic scenario forms a test case

Give priority to each test case

- Consider the priority of the function
- Further take into account the following aspects:
  - frequency of the occurrences of the scenario in the use case
  - possible errors in the scenario
  - consequences of the errors

Identify input, output and environments for each test case

- Check the scenario description
- Find the input/output variables and the data needed to be stored in the system

Design Test Cases

The Clinic Example: Make Appointment Function

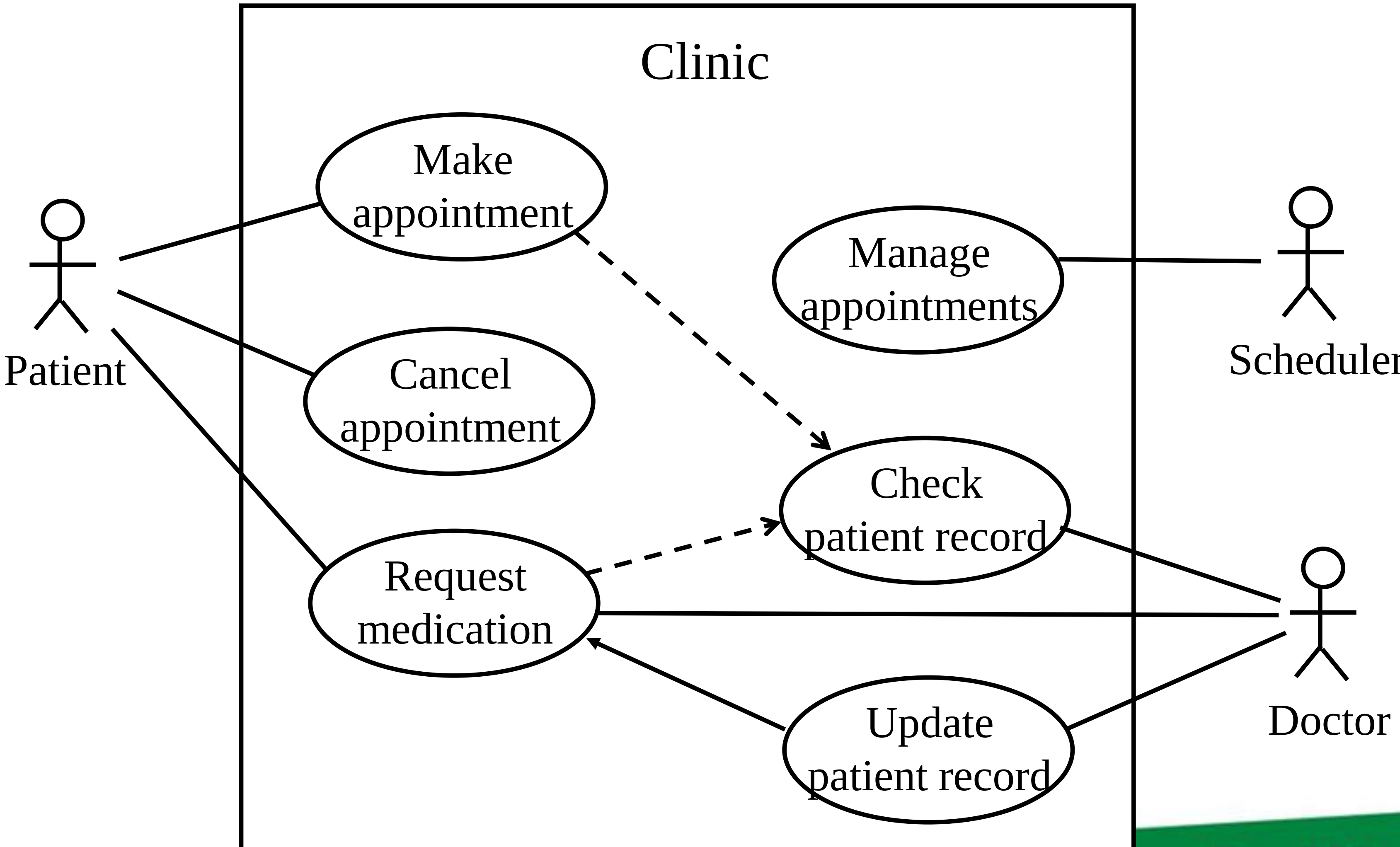
Actor	System
1) Patient enters their own name and their preferred doctor's name	2) After confirming this is a registered patient, display days the doctor is in the clinic
3) Selects date	4) Displays available appointments
5) Selects preferred time	6) Confirms appointment and displays information about parking etc
7) Confirms acceptance of appointment	8) Add to appointments list.

- Input:
  - Patient name, preferred doctor's name, date, time, confirmation
- Output:
  - available dates of a doctor, available times
- Stored information:
  - available dates of a doctor, available times
  - Appointment detail

Selection of Test Data

- Generate test data from each concrete scenario
- Identify the values in the concrete scenario for each variable of the corresponding generic scenario
- Set the environment variables as the values of the concrete scenario
- List values for input variables
- List expected values for output variables

Developing Test Plan from Use Cases



The Clinic Example

Concrete Scenario Description of making appointment

Actor	System
1) Patient John enters his name and preferred doctor, Dr Walker	2) confirmed John is a registered patient, displays days the doctor is in the clinic, which is Monday and Wednesday.
3) John selects Monday	4) Displays available appointment, which is 10:30am, 12:00am, and 3:00pm.
5) John selects 10:30am	6) Confirms appointment and displays information about parking etc
7) John confirms the acceptance of appointment	8) Appointment confirmed

Derivation of Test Data: The Clinic Example

Test data derived from the concrete scenario

	Variable	Test Data 1
Input	patient name	John
	doctor’s name	Walker
	date	Monday
	time	10:30am
	confirmation	True
Expected Output	available dates	Monday, Wednesday
	available times	10:30am, 12:00am, 3:00pm
Stored info.	available dates	Monday, Wednesday
	available times	10:30am, 12:00am, 3:00pm

Test Planning: Risk Analysis

List all functions to be tested

- Check the use case diagram
- Each use case is a function to be tested

Analyse the risk of each function

- Possible errors
- The frequency of the use case to be used
- Consequences of any occurrence of an error

Give priorities to the functions

- The server the consequence or the heavier loss the higher priority
- The more frequently used use cases the higher priority

Analysis of Risks: The Clinic Example

Function	Possible errors	Consequences	Priority
Make appointment	Double booking	Patient inconvenience	Medium
	No booking on available time	Inefficiency for doctors	Medium
Cancel app.	…	…	Low
Manage app.	…	…	Medium
Request med.	…	…	High
Check patient record	Wrong record displayed	Cause anxiety Error in treatment	Very High
	Displayed other patient's record	Leak private information	High
Update patient record	Lost of record	Cause anxiety Error in treatment	Very High
	Wrong record stored		Very high

Testing Units

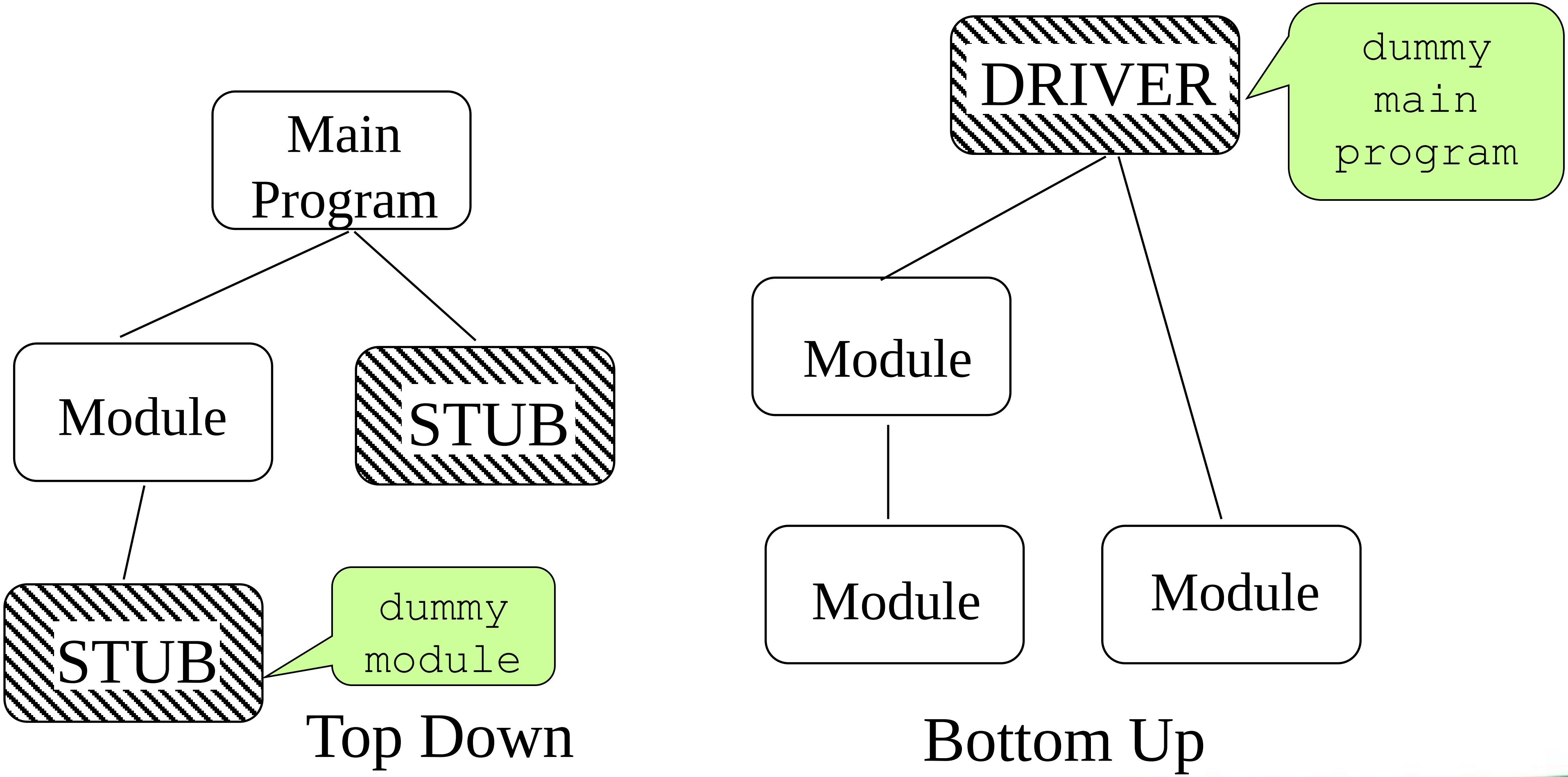
A test unit can be:

- an instruction (machine, assembly, high level, …)
- A feature (from the requirements spec or users guide)
- A class
- A group of classes (a cluster)
- A library
- An ADT
- A program
- A set (or suite) of programs

Traditional (Structured) units are module, function, procedure etc

Object oriented (encapsulated data+operatons) units are Classes. Note with inheritance operations must be tested for each instance, e.g. shape cannot be tested unless circle, rectangle, triangle etc are also tested.

Integration Testing – systematic approach to testing modular systems

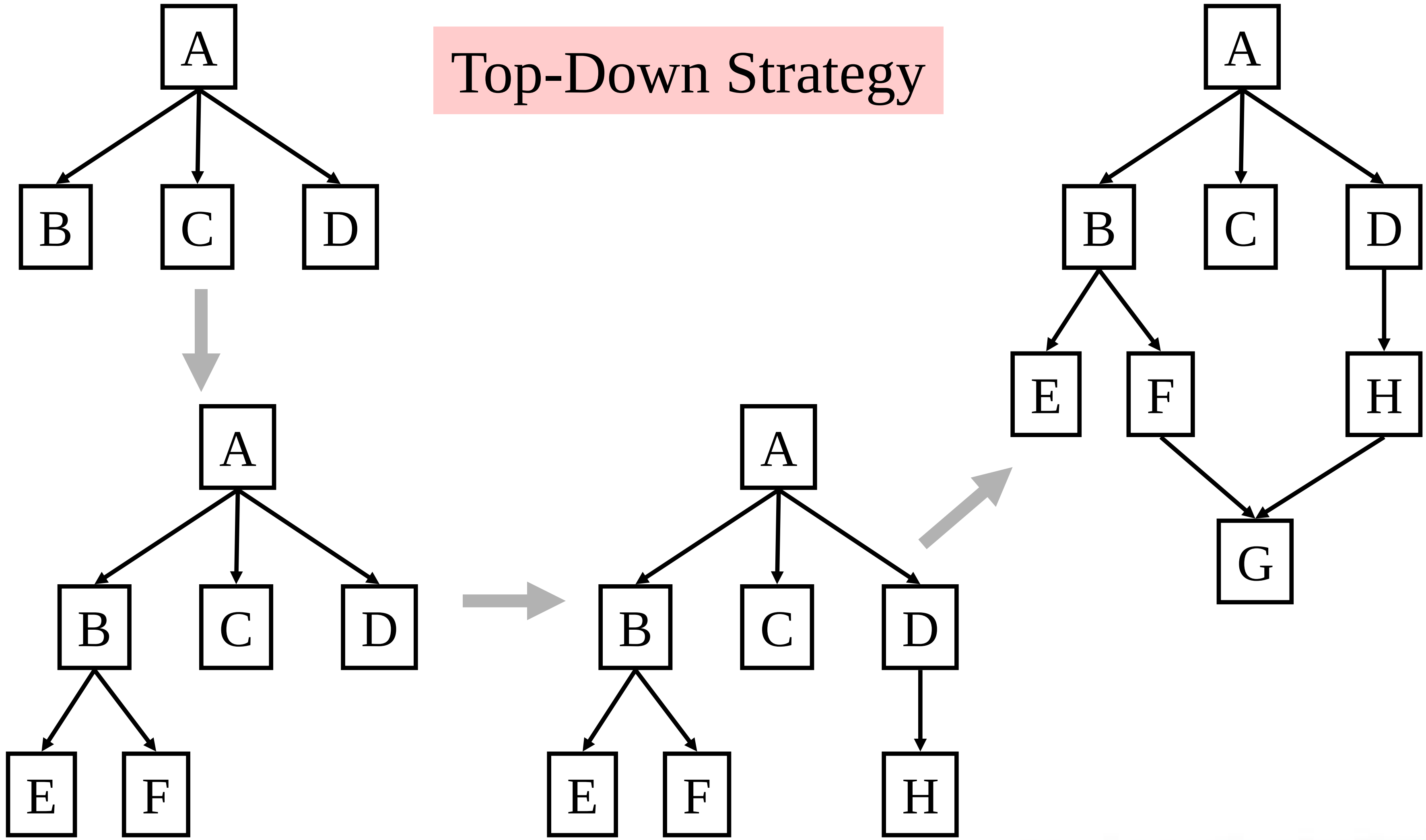


Integration Testing – in practice

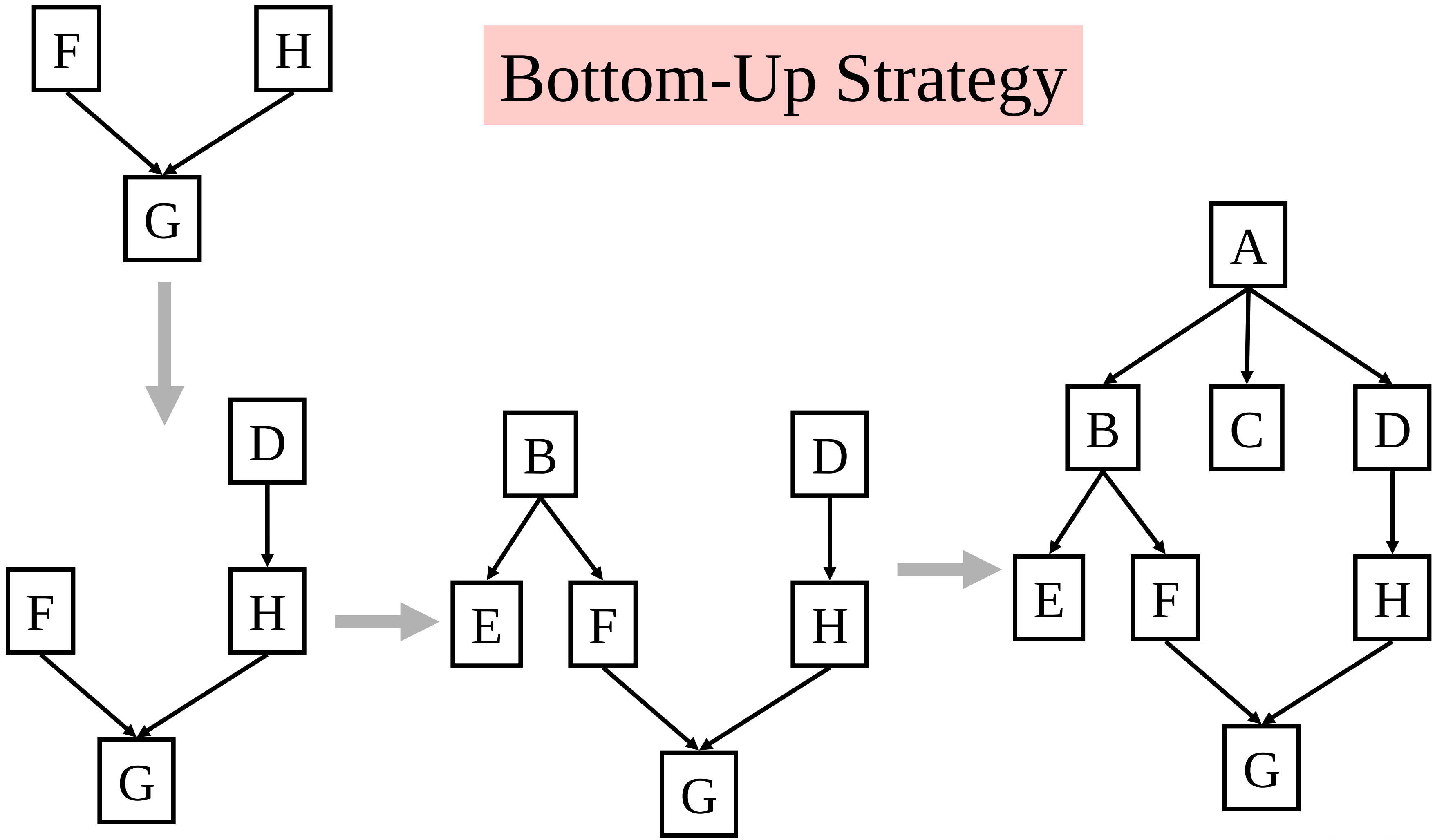
- Test several units as a system or sub-system
- May be several code authors, teams or organisations involved
- Hierarchical test approach top down or bottom up

Integration Strategies:

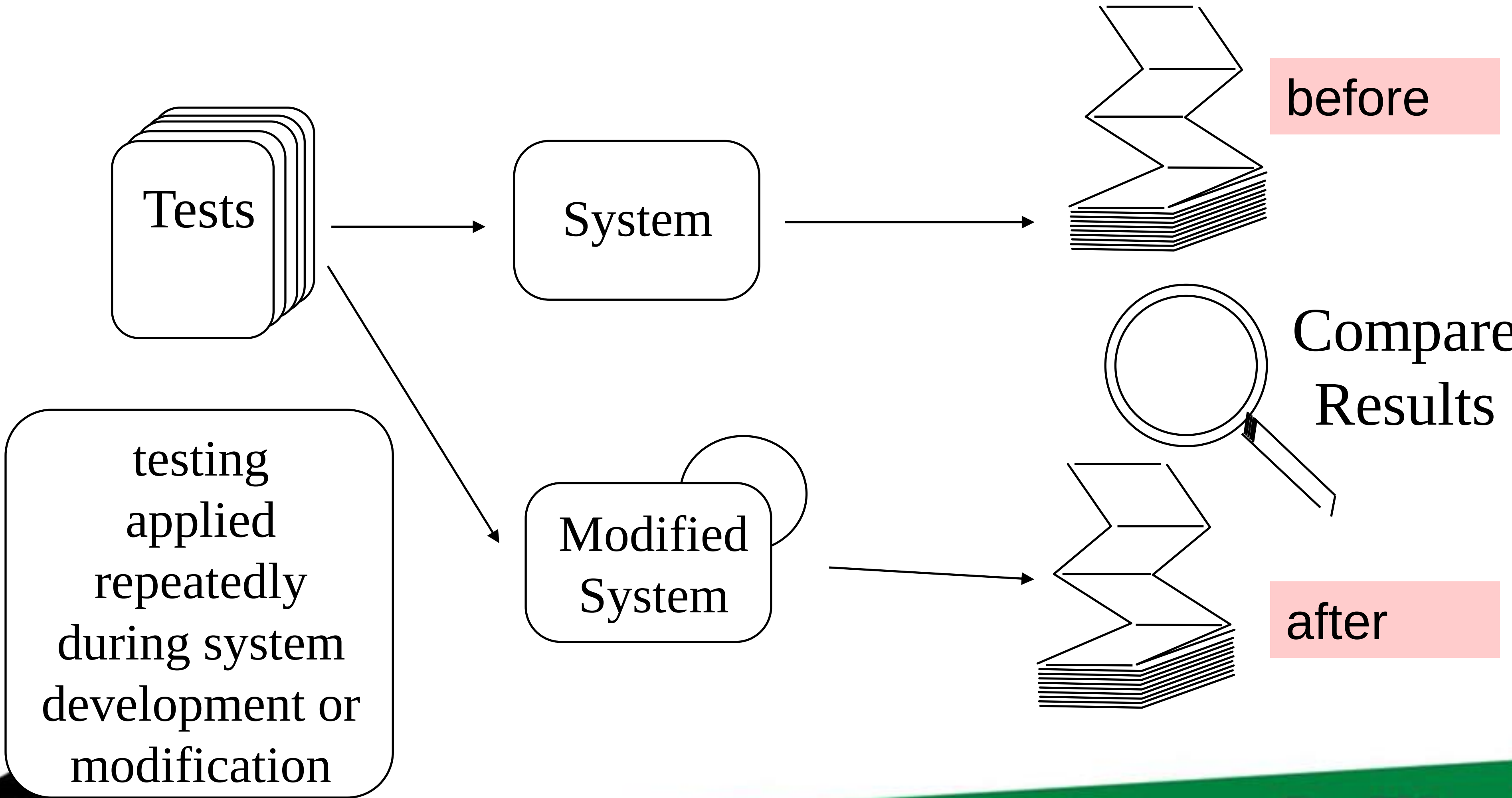
Top-Down Strategy



Bottom-Up Strategy



Regression Testing (does it still work, after the modification?)



Dynamic Testing: Adequacy and Testability

- **Adequacy** – level of confidence in the testing applied to a system.
  - The adequacy criteria can be requirements based (i.e. black box tested) or structure based (i.e. white box tested)
  - Different adequacy criteria for systems of different degrees of criticality
  - Typically these specify 100% coverage for testing related to the system safety requirements
- **Design for Testability**
  - Design approach that considers later ease of testing, (some systems cannot be tested)
  - Testability approaches: ***Ad hoc*** - testing is considered after the design or ***Built in test*** - testing is an integral part of the system design
  - **Controllability**, the ability to input (or control) signals to set the system into a particular state
  - **Observability**, the ability to examine (observe) the system status from the external outputs

Testability Principles

- how easily can a program be tested

- Operability: - The better it works, the more efficiently it can be tested
- Observability: - What you see is what you test
- Controllability: - The better we can control the software, the more the testing can be automated and optimized
- Decomposability: - By controlling the scope of testing, we can more quickly isolate problems and perform smarter re-testing
- Simplicity: - The less there is to test, the more quickly we can test it
- Stability: - The fewer the changes, the fewer the disruptions to testing
- Understandability: - The more information we have, the smarter we will test

Summary

- Testing is a vital part of system development
- Applies equally to hardware and software
- Contributes to overall system quality
- Reduces risk (for developers and users)
- Testing cost typically 25-50+% of software development costs (usually recorded as maintenance)
- Not all software aspects testable with same ease (some easy, some difficult/impracticable)
- Good systems testing staff are essential
- Good organisation and documentation is vital
- There are commercial CASE tools available that can help in many situations