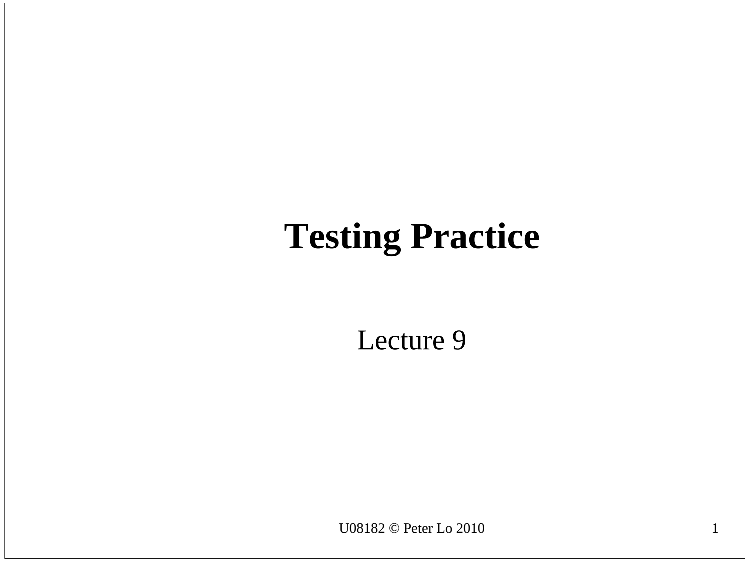
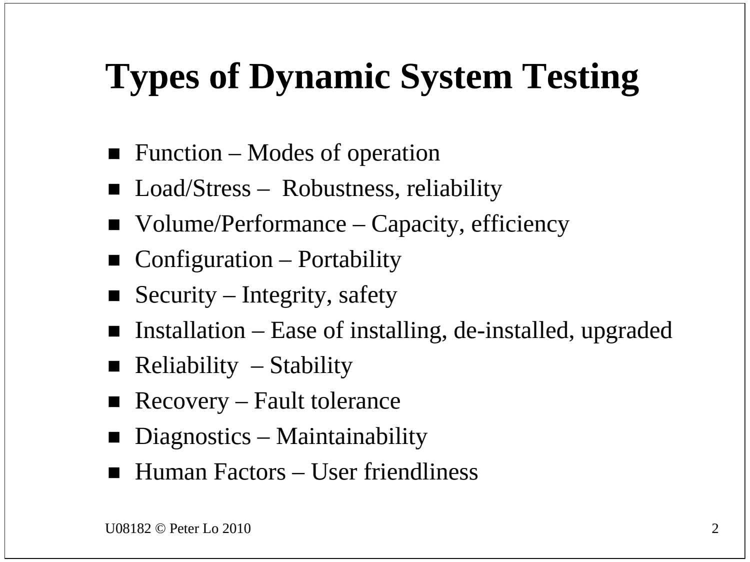


Testing Practice

Lecture 9

In this lecture you will learn:

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



Types of Dynamic System Testing

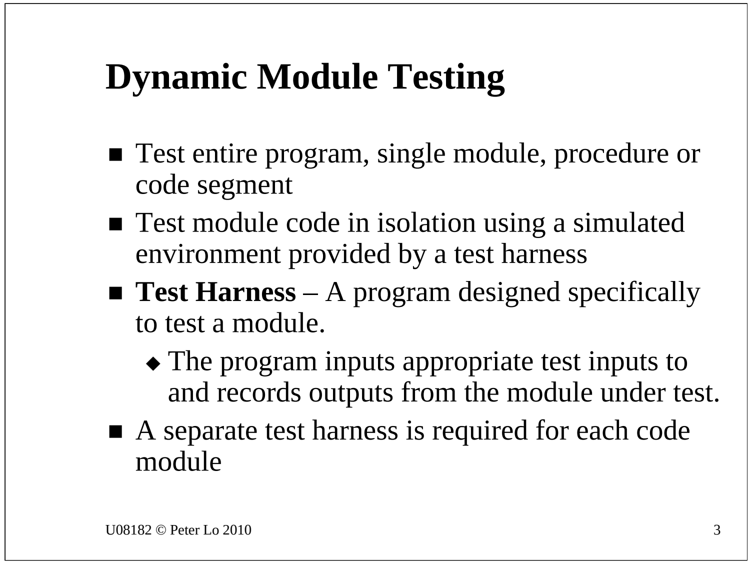
- Function – Modes of operation
- Load/Stress – Robustness, reliability
- Volume/Performance – Capacity, efficiency
- Configuration – Portability
- Security – Integrity, safety
- Installation – Ease of installing, de-installed, upgraded
- Reliability – Stability
- Recovery – Fault tolerance
- Diagnostics – Maintainability
- Human Factors – User friendliness

Dynamic Testing involves the computer operation of the system (or module) under test

System testing can assess many aspects of system performance. From the project manager's point of view system testing may involve considerable extra planning. For example, load or stress testing (increasing the load on the system until its performance begins to degrade) may require additional hardware to be acquired possibly extra staff (who may need initial training) etc.

The main forms of system testing are:

- Function – The system performs the functions required of it
- Load/Stress – The impact on performance of increasing the amount the system is doing or the impact of running other systems concurrently with it. Does the system degrade 'gracefully'?
- Volume/Performance – Measuring the performance of the system at various levels of loading (unlike the last set of tests this is not intended to 'break' the system).
- Configuration – Systems are often developed on different hardware from that which will be used for live running of the system - or in the case of packages it may have to run on several hardware platforms.
- Security – Is it possible to breach the security of the system in anyway?
- Installation – Can the software be installed once it has been packaged? In some cases installation software may have had to be written and tested.
- Reliability – Can the system run for long periods without crashing?
- Recovery – How well does the system recover from failure? This may include difficult to simulate problems such as hardware errors.
- Diagnostics – In the event of failure, is diagnostic information produced? This could also include systems where there is an auditing requirement that, for example, a transaction can be traced through the system.
- Human Factors – Of use, consistency of interfaces etc



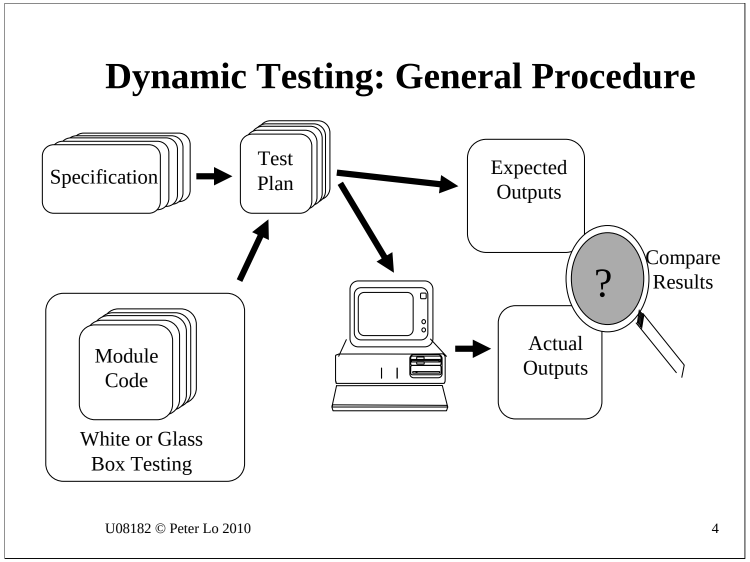
Dynamic Module Testing

- 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

Each module to test requires a self contained test harness to supply the test inputs, call module under test and capture the outputs.

The module outputs are compared with known good output

The test harness can be used at any time to conform the module operation



Dynamic Testing: General Procedure

Specification

Test Plan

Expected Outputs

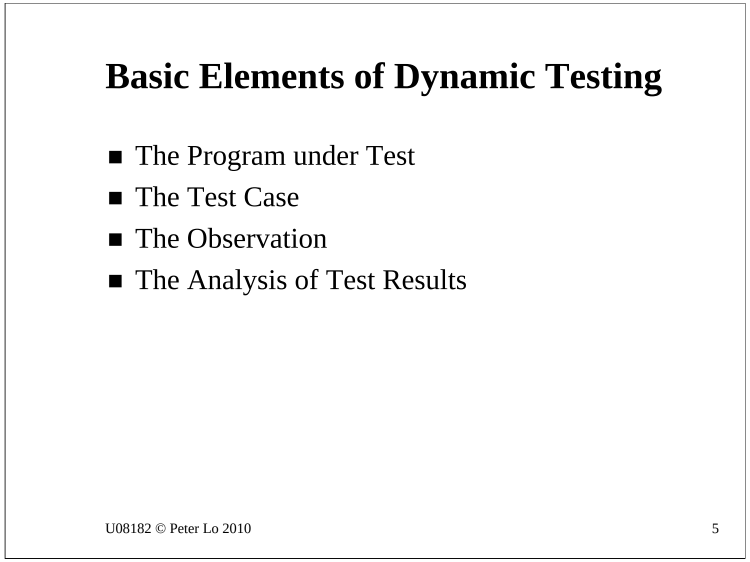
Compare Results

?

Module Code

White or Glass Box Testing

Actual Outputs



Basic Elements of Dynamic Testing

- The Program under Test
- The Test Case
- The Observation
- The Analysis of Test Results

The Program under Test

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

The Test Case

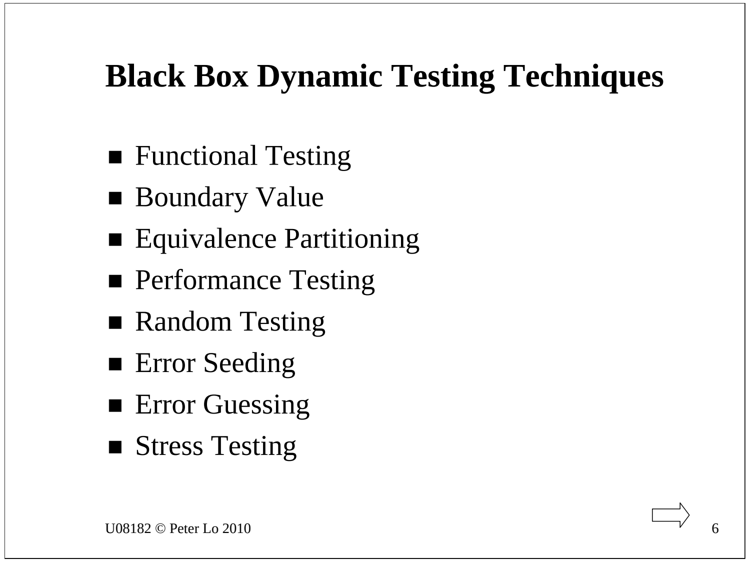
- The input data to run the program
- The expected output / dynamic behaviour (e.g. timing)

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
- Boundary Value
- Equivalence Partitioning
- Performance Testing
- Random Testing
- Error Seeding
- Error Guessing
- Stress Testing

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

Black Box Dynamic Testing Techniques: *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.

Black Box Dynamic Testing Techniques: *Equivalence Partitioning*

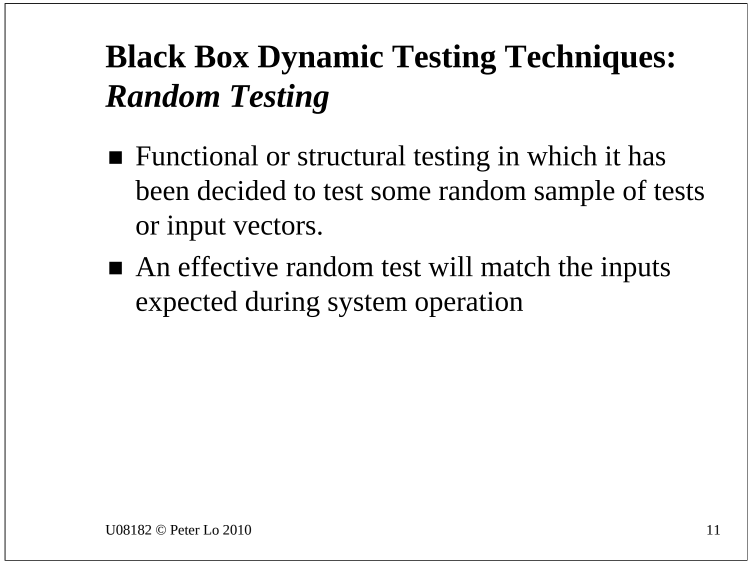
- Group sets of input and output ranges that can be treated in same way.
- Test performed on each set.

Equivalence partitioning – group sets of input and output ranges that can be treated in same way. Test performed on each set.

Aim is to reduce the number of discrete tests. An ideal test case single-handedly uncovers a class of errors (e.g. incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. An equivalence class could be: a single numeric value, a range of values, a set of related values or a Boolean condition. Then, for example, if the input condition specifies a set then one valid and one invalid equivalence class are defined.

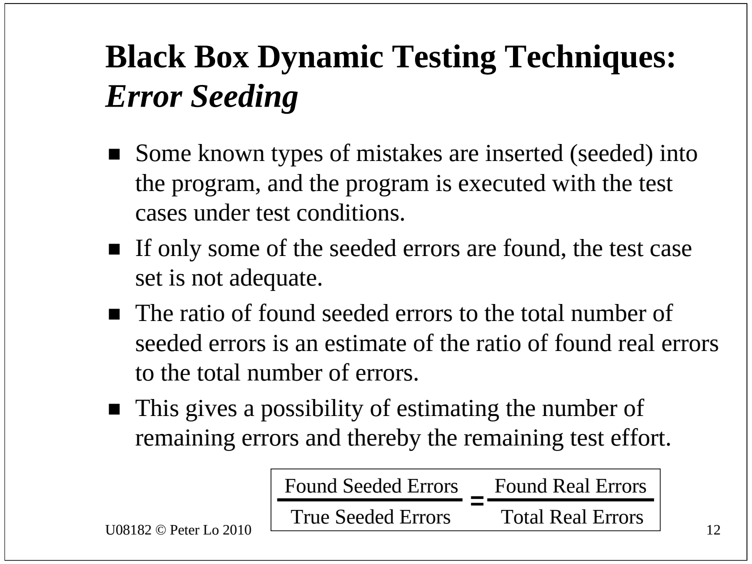
Black Box Dynamic Testing Techniques: *Performance Testing*

- 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



Black Box Dynamic Testing Techniques: *Random Testing*

- 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



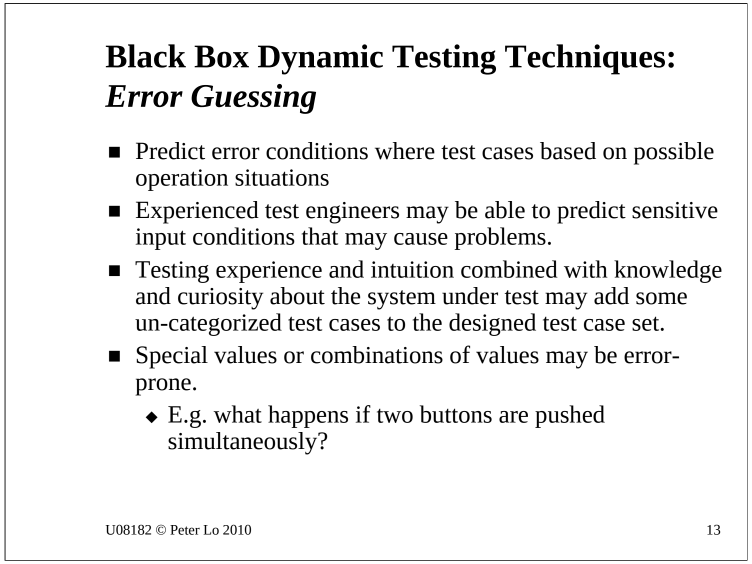
Black Box Dynamic Testing Techniques: *Error Seeding*

- Some known types of mistakes are inserted (seeded) into the program, and the program is executed with the test cases under test conditions.
- If only some of the seeded errors are found, the test case set is not adequate.
- The ratio of found seeded errors to the total number of seeded errors is an estimate of the ratio of found real errors to the total number of errors.
- This gives a possibility of estimating the number of remaining errors and thereby the remaining test effort.

$$\boxed{\frac{\text{Found Seeded Errors}}{\text{True Seeded Errors}} = \frac{\text{Found Real Errors}}{\text{Total Real Errors}}}$$

Error seeding – involves inserting errors into the implementation to check that the testing will find them, and hence check the testing process

Some known types of mistakes are inserted (seeded) into the program, and the program is executed with the test cases under test conditions. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is an estimate of the ratio of found real errors to the total number of errors. This gives a possibility of estimating the number of remaining errors and thereby the remaining test effort. **Found seeded errors/true seeded errors = found real errors/total real errors**

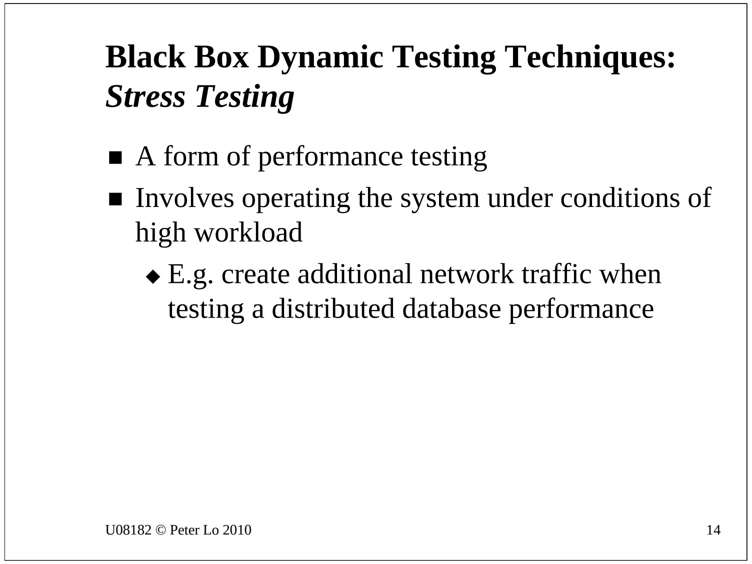


Black Box Dynamic Testing Techniques: *Error Guessing*

- Predict error conditions where test cases based on possible operation situations
- Experienced test engineers may be able to predict sensitive input conditions that may cause problems.
- Testing experience and intuition combined with knowledge and curiosity about the system under test may add some un-categorized test cases to the designed test case set.
- Special values or combinations of values may be error-prone.
  - E.g. what happens if two buttons are pushed simultaneously?

Error guessing – experienced test engineers may be able to predict sensitive input conditions that may cause problems.

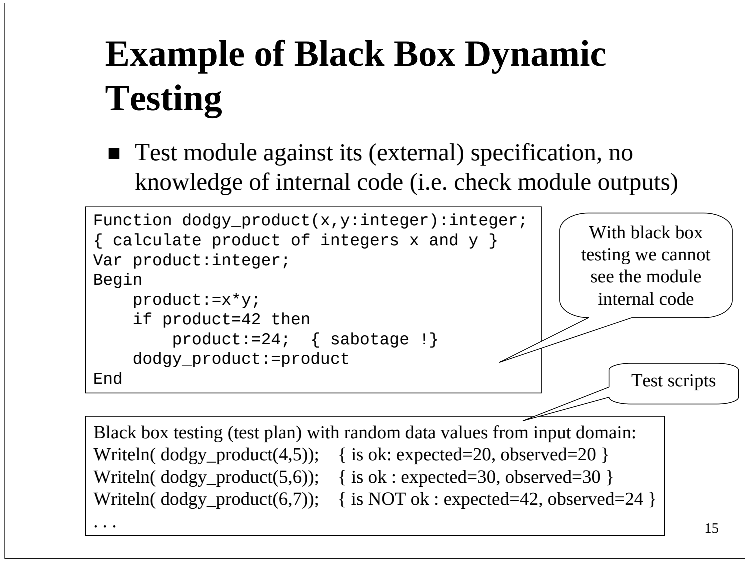
Testing experience and intuition combined with knowledge and curiosity about the system under test may add some un-categorized test cases to the designed test case set. Special values or combinations of values may be error-prone. For example, can the buttons be pushed on a front-panel too fast or too often? and what happens if two buttons are pushed simultaneously?



Black Box Dynamic Testing Techniques: *Stress Testing*

- A form of performance testing
- Involves operating the system under conditions of high workload
  - E.g. create additional network traffic when testing a distributed database performance

Stress testing (a form of performance testing) – involves operating the system under conditions of high workload (e.g. create additional network traffic when testing a distributed database performance)



Example of Black Box Dynamic Testing

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

```
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

Test scripts

```
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 }
...
```

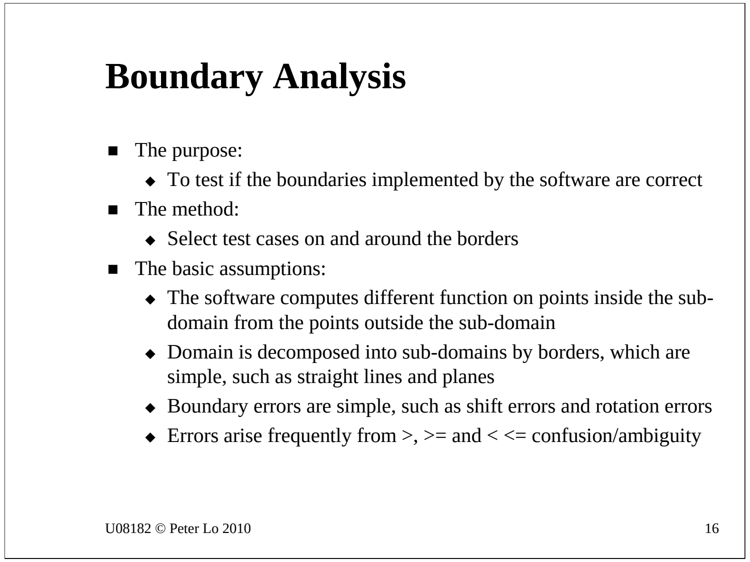
Example black box test plan

Choose values from input domain

Check for correct operation/output

Check against external spec only

Obvious limitation to black box testing



Boundary Analysis

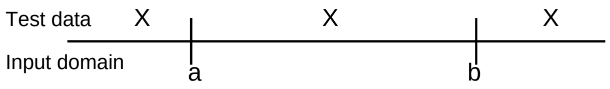
- The purpose:
  - To test if the boundaries implemented by the software are correct
- The method:
  - Select test cases on and around the borders
- The basic assumptions:
  - The software computes different function on points inside the sub-domain from the points outside the sub-domain
  - Domain is 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

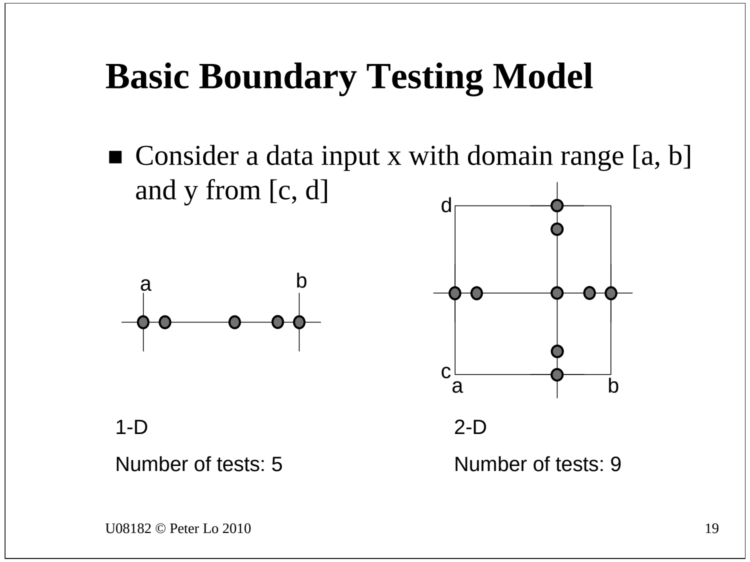
Boundary Value Testing

- 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

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

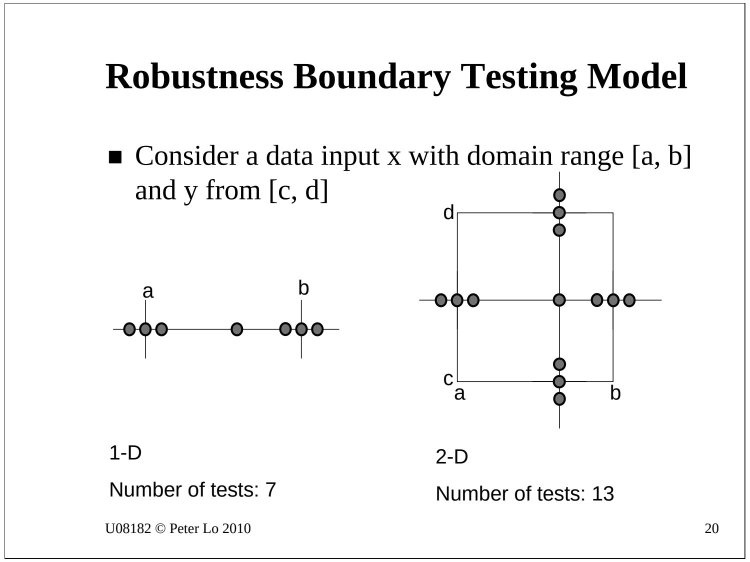
1-D

Number of tests: 5

2-D

Number of tests: 9

U08182 © Peter Lo 2010 19



Robustness Boundary Testing Model

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

1-D

Number of tests: 7

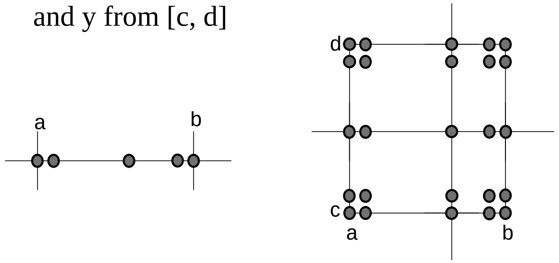
2-D

Number of tests: 13

U08182 © Peter Lo 2010 20

Worst Case Boundary Testing Model

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



1-D

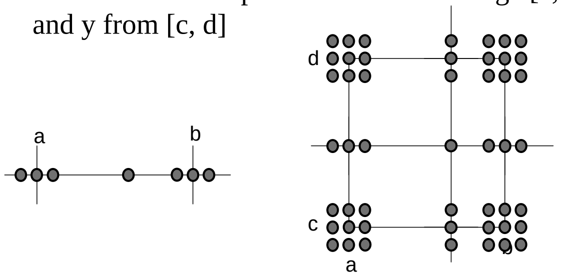
Number of tests: 5 (5^1)

2-D

Number of tests: 25 (5^2)

Worst Case Robust Boundary Testing Model

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



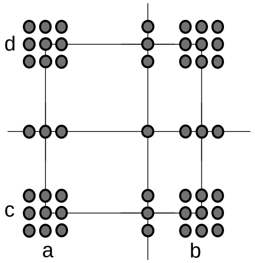
1-D

Number of tests: 7 (7^1)

2-D

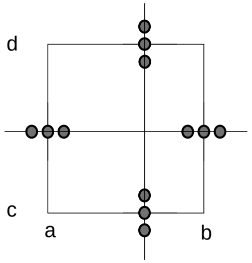
Number of tests: 49 (7^2)

Adequacy Criteria (using White Box Static Testing)



Worst Case Robust Boundary Testing

No. of tests: 49



Adequacy Criteria - no of tests: 12

Where x and y testing is independent

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

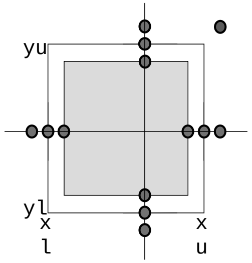
So in the case of the previous example

E.g. for testing for point inside a rectangle

Adequacy Criteria for Point inside Rectangle

- Example

```
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

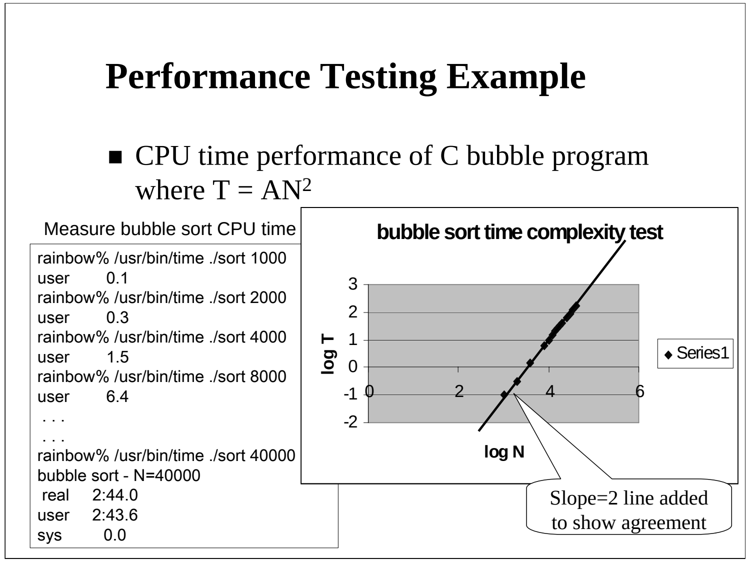
Other regions are either on the edge, or outside so return FALSE

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 $T = AN^2$

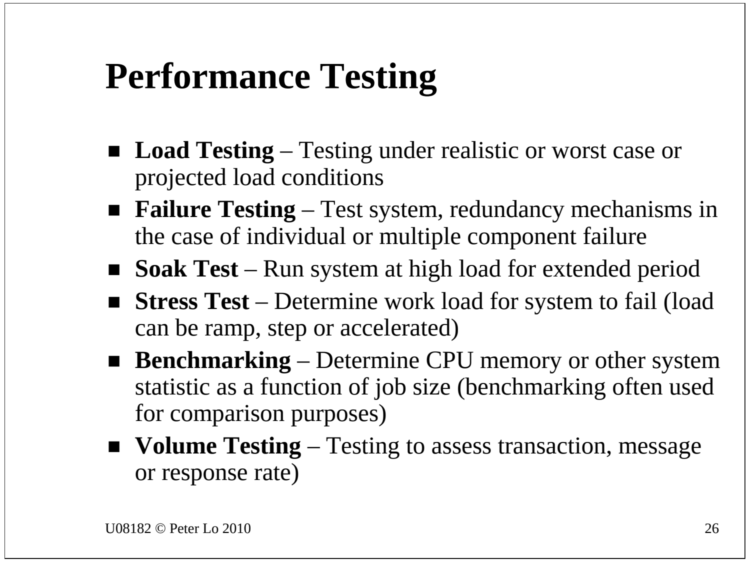
Measure bubble sort CPU time

```
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
```

Slope=2 line added to show agreement

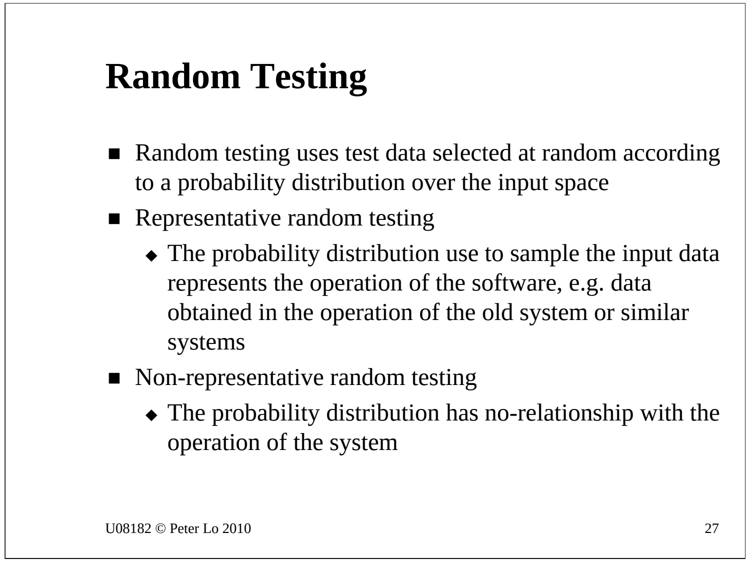
Performance testing to confirm complexity behaviour

Douby nested loop (in bubble sort) leads to (Nsquared) behaviour



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

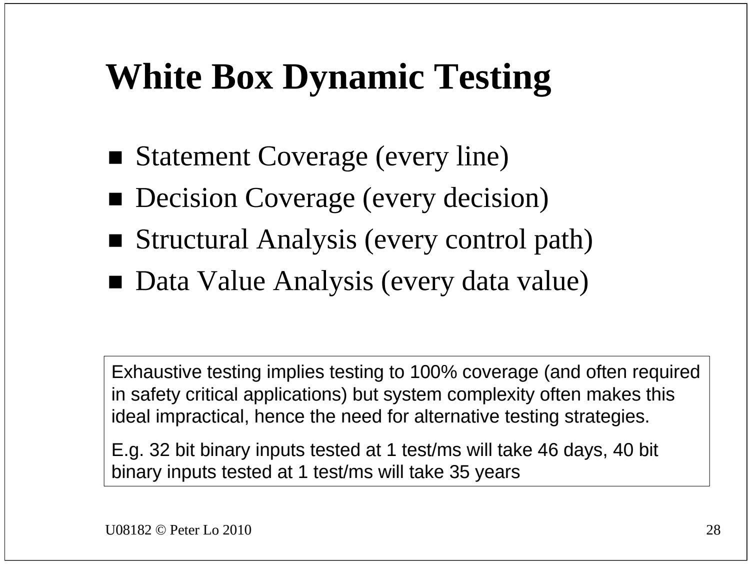
- Random testing uses test data selected at random according to a probability distribution over the input space
- Representative random testing
  - The probability distribution use to sample the input data represents the operation of the software, e.g. data obtained in the operation of the old system 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)
- Decision Coverage (every decision)
- Structural Analysis (every control path)
- Data Value Analysis (every data value)

Exhaustive testing implies testing to 100% coverage (and often required in safety critical applications) but system complexity often makes this ideal impractical, hence the need for alternative testing strategies.

E.g. 32 bit binary inputs tested at 1 test/ms will take 46 days, 40 bit binary inputs tested at 1 test/ms will take 35 years

Statement coverage (every line) - aim is to create enough tests to ensure every statement is executed at least once

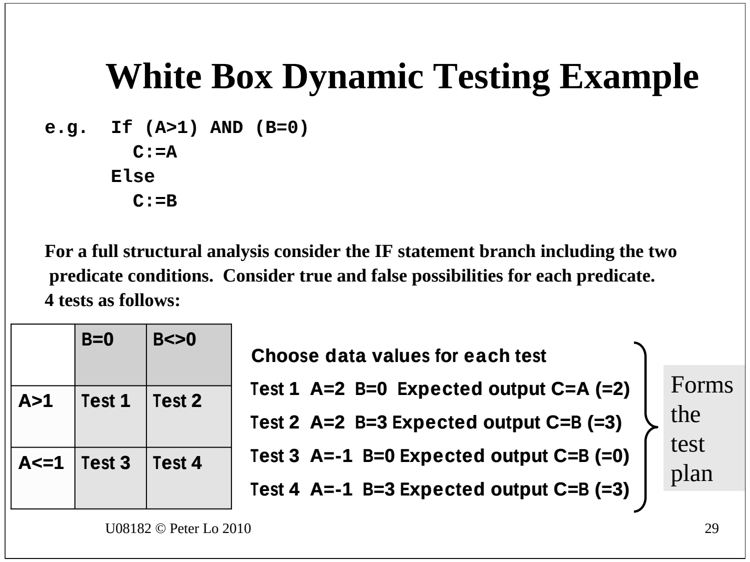
Decision coverage (every decision) - aim is to generate tests to execute each decision statement branch and module exit path. For example provide tests to exercise both **true** and **false** branches in IF statements. More rigorous than statement coverage

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.

Exhaustive testing implies testing to 100% coverage (and often required in safety critical applications) but system complexity often makes this ideal impractical, hence the need for alternative testing strategies.

e.g. 40 bit binary inputs tested at 1 test/ms will take 35 years



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

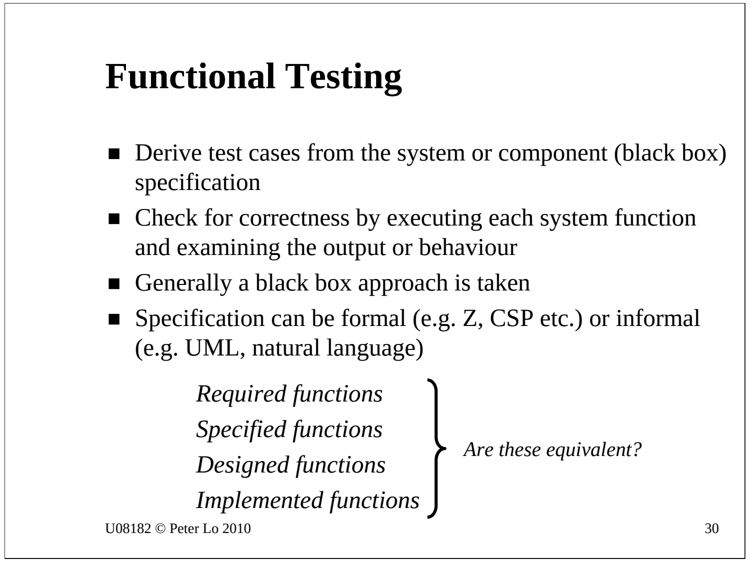
Test 1 A=2 B=0 Expected output C=A (=2)

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)

Required functions
Specified functions
Designed functions
Implemented functions

Are these equivalent?

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.

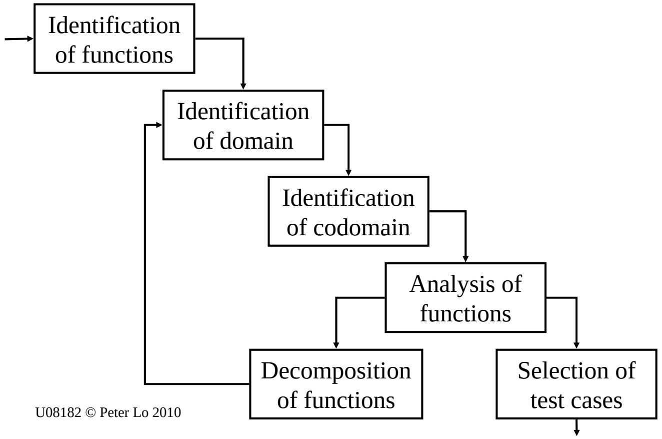
Dimension of domain

- 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


Identification of functions

Identification of domain

Identification of codomain

Analysis of functions

Decomposition of functions

Selection of test cases

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?

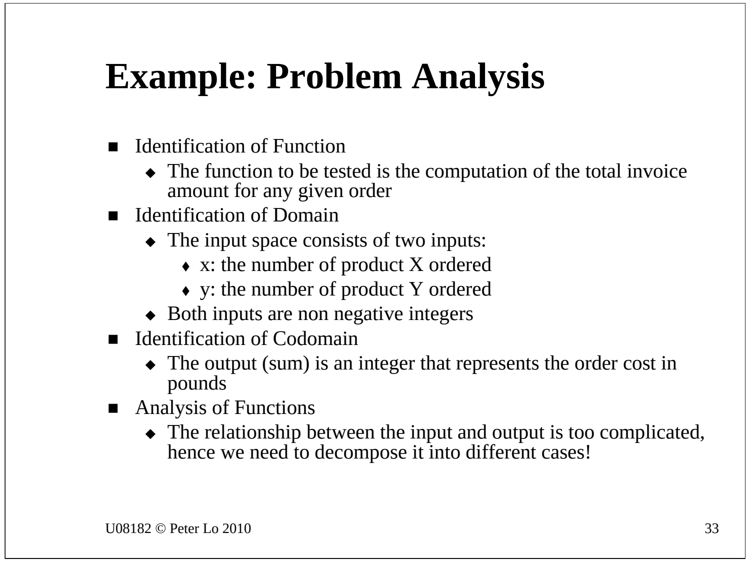
Analysis of the Function
- What is the relationship between the domain and codomain?
- Can it be decomposed into simpler functions?

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?

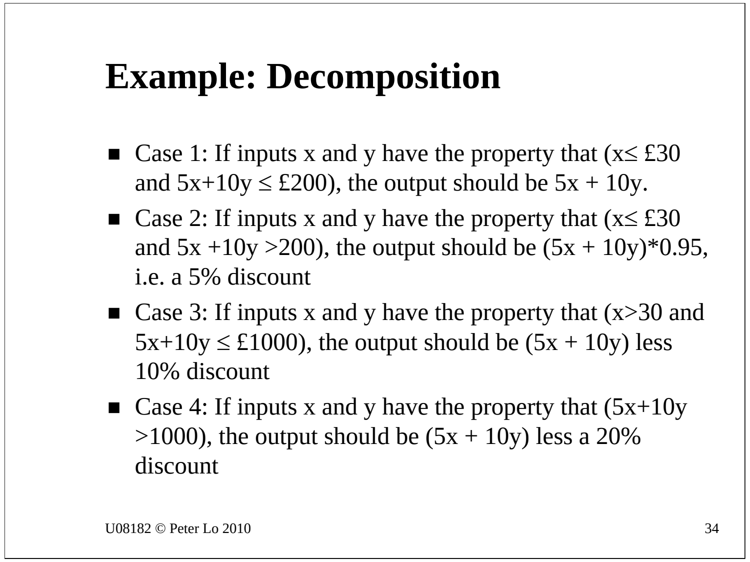
Example: Discount Invoice

- 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.



Example: 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 into different cases!



Example: Decomposition

- Case 1: If inputs x and y have the property that (x≤ £30 and 5x+10y ≤ £200), the output should be 5x + 10y.
- Case 2: If inputs x and y have the property that (x≤ £30 and 5x +10y >200), the output should be (5x + 10y)*0.95, i.e. a 5% discount
- Case 3: If inputs x and y have the property that (x>30 and 5x+10y ≤ £1000), the output should be (5x + 10y) less 10% discount
- Case 4: If inputs x and y have the property that (5x+10y >1000), the output should be (5x + 10y) less a 20% discount

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

- Sub-function 1:
  - Sub-Domain: A = {(x, y) | x ≤ £30, 5x+10y≤£200}
  - Relationship: sum = 5x+10y

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

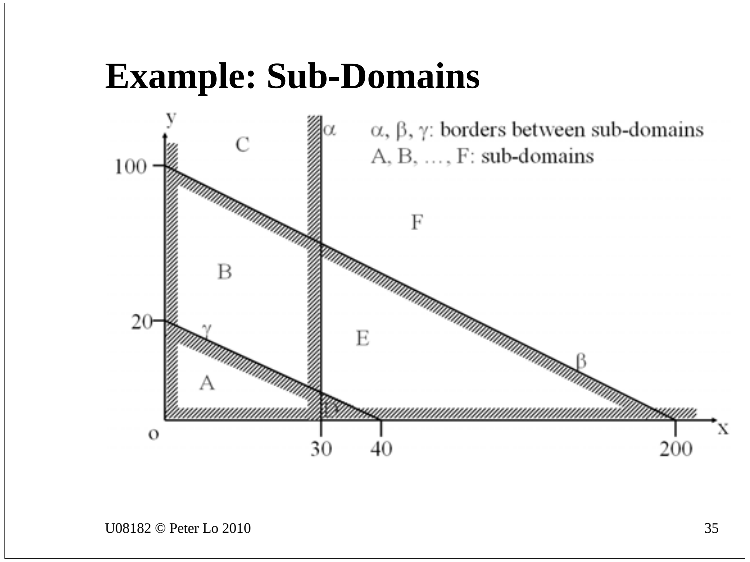
- Sub-function 2:
  - Sub-Domain: B = {(x, y) | x ≤ £30, 5x+10y>200}
  - Relationship: sum = Int(0.95*(5x+10y))

Case 3: If inputs x and y have the property that (x>30 and 5x+10y ≤ £1000), the output should be (5x + 10y) less 10% discount

- Sub-function 3:
  - Sub-Domain: D and E = {(x, y) | x >30, 5x+10y ≤ £1000}
  - Relationship: sum = Int(0.9*(5x+10y))

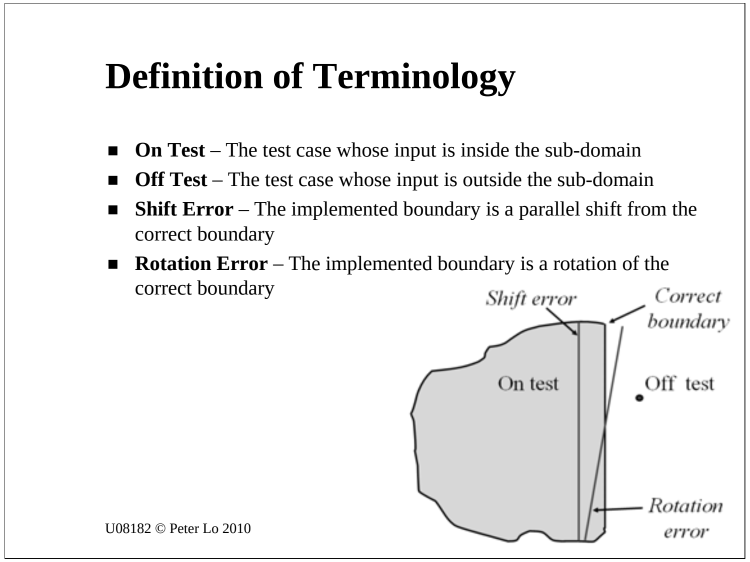
Case 4: If inputs x and y have the property that (5x+10y >1000), the output should be (5x + 10y) less a 20% discount

- Sub-function 4:
  - Sub-Domain: C and F = {(x, y) | 5x+10y>1000}
  - Relationship: sum = Int(0.8*(5x+10y))



Example: 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

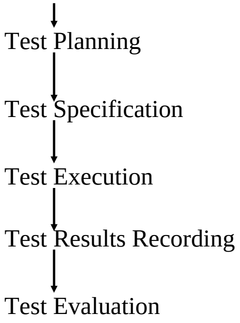
On Test – The test case whose input is inside the sub-domain

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

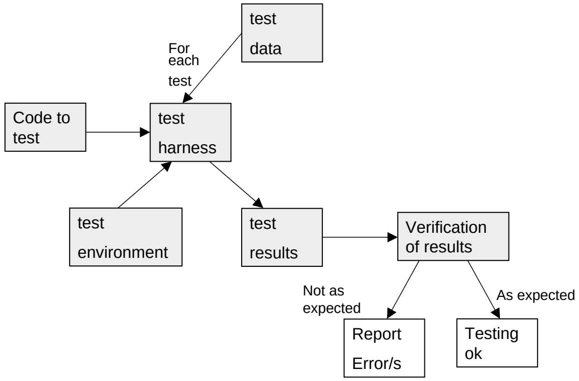


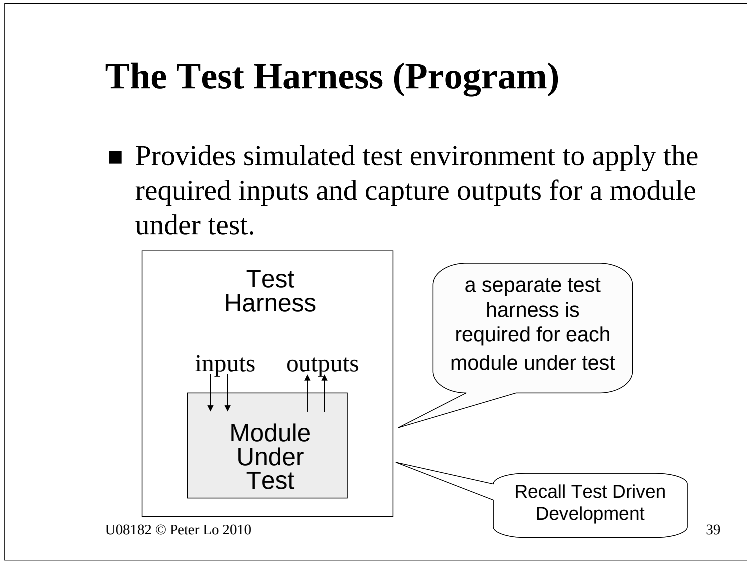
1. What to test
2. How to test
3. Record of tests
4. Test evaluation

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.

Test Harness
inputs
outputs
Module Under Test
a separate test harness is required for each module under test
Recall Test Driven Development

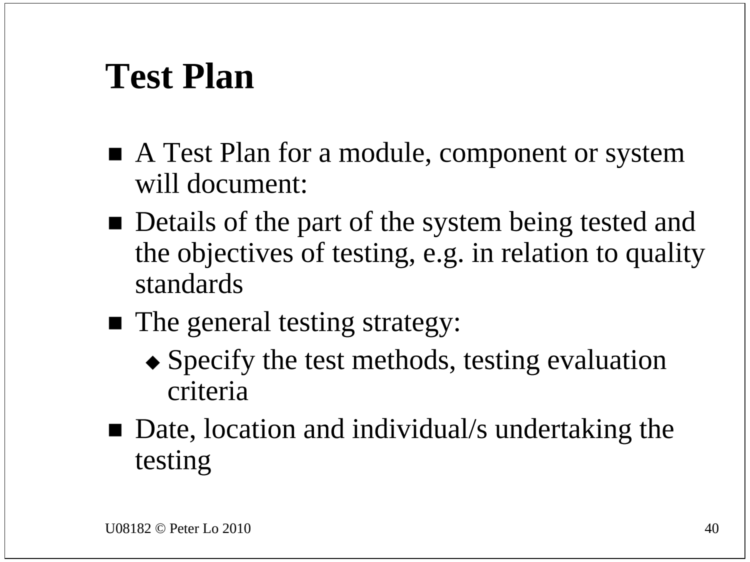
The use of the test harness can be extended to the use of Simulators to provide dynamic testing applied outside the operational environment (e.g. nuclear shutdown system) , and particularly applied to safety critical systems. Note accuracy of simulation will be limited by:

Comprehensiveness of environmental variables

Accuracy of system model and environmental factors

Accuracy of the dynamic behaviour

Therefore note that results will only be as good as the system model



Test Plan

- 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
- Date, location and individual/s undertaking the testing

For each test, include:

•Details and purpose of test **(what program/module, what level of testing)**

•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

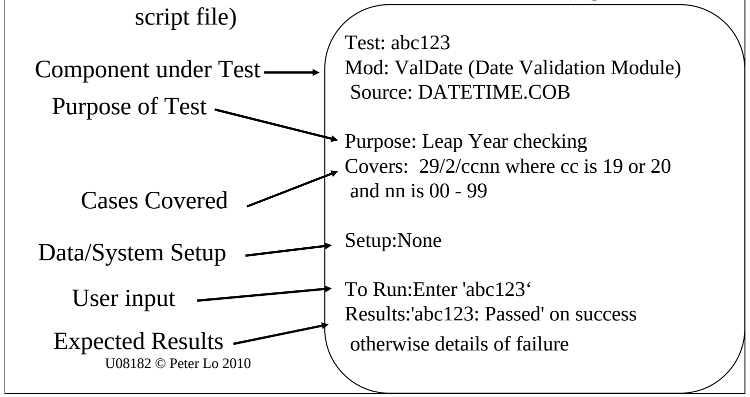
The test plan forms an integral part of the software life cycle design process

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

Purpose of Test

Cases Covered

Data/System Setup

User input

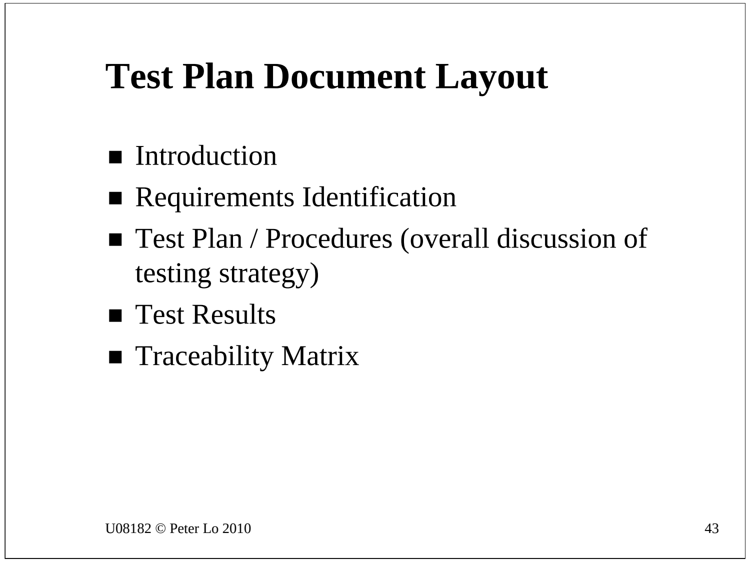
Expected Results

Test: abc123
Mod: ValDate (Date Validation Module)
Source: DATETIME.COB

Purpose: Leap Year checking
Covers: 29/2/ccnn where cc is 19 or 20
and nn is 00 - 99

Setup:None

To Run:Enter 'abc123‘
Results:'abc123: Passed' on success
otherwise details of failure



Test Plan Document Layout

- Introduction
- Requirements Identification
- Test Plan / Procedures (overall discussion of testing strategy)
- Test Results
- Traceability Matrix

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)

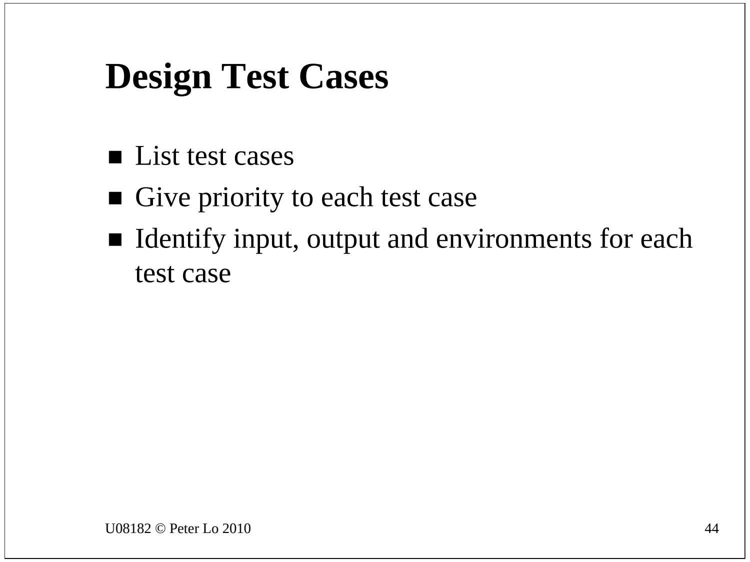
- Develop a minimum number of tests which cover all the requirements
- Each case (scenario) requires details of the hardware and software setup, input required together with the expected behavior/output and how this can be observed.
- The use case documentation can form the basis for this

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
- Give priority to each test case
- Identify input, output and environments for each test case

List test cases

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

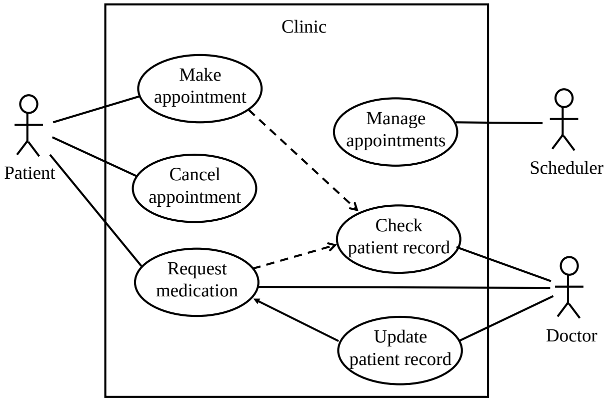
Give priority to each test case

- 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

The Clinic Example: Developing Test Plan from Use Cases

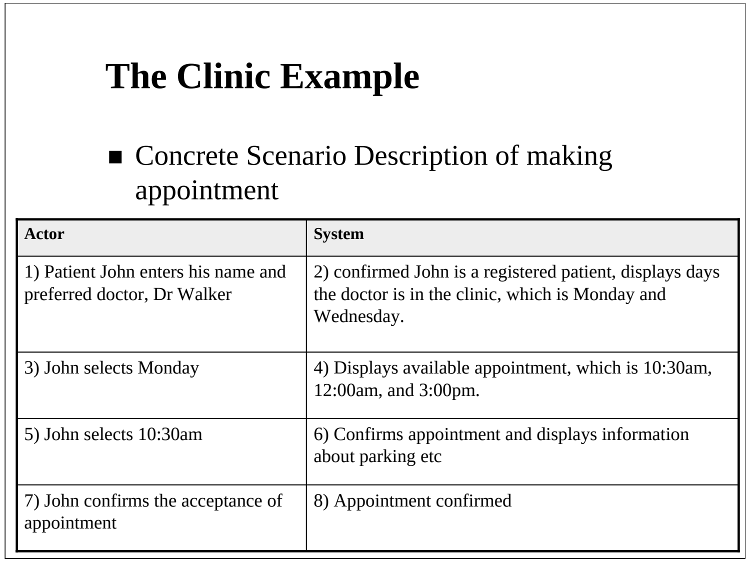


The Clinic Example: Make Appointment Function

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.

The Clinic Example: Make Appointment Function

- 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



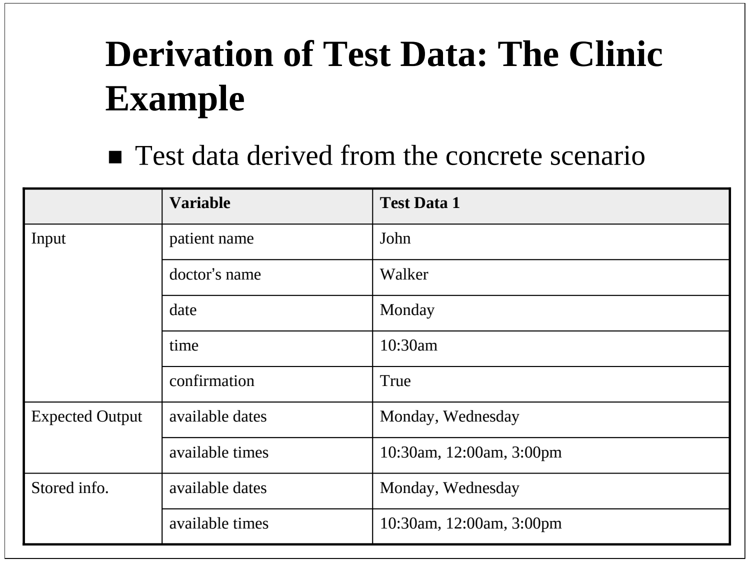
The Clinic Example

- 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

Selection of Test Data

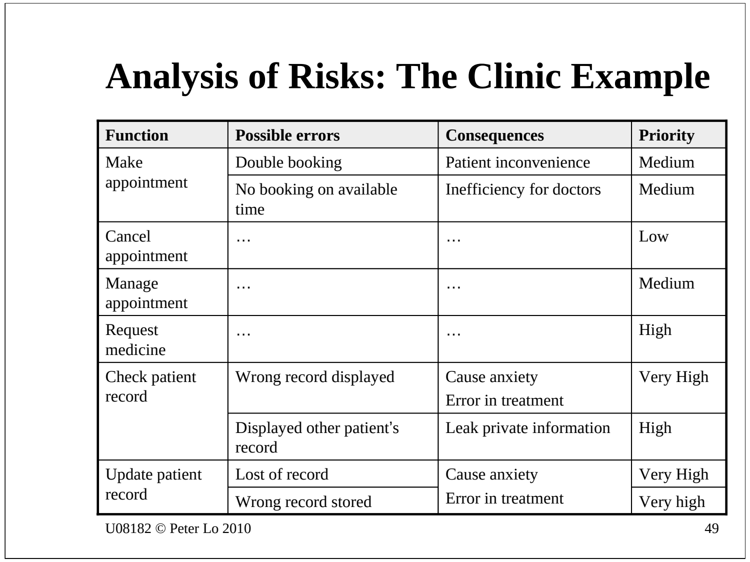
- Generate test data from each concrete scenario
- Identify the values in the concrete scenario for each variable of the corresponding generic scenario
- Set the environment variables as the values of the concrete scenario
- List values for input variables
- List expected values for output variables



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

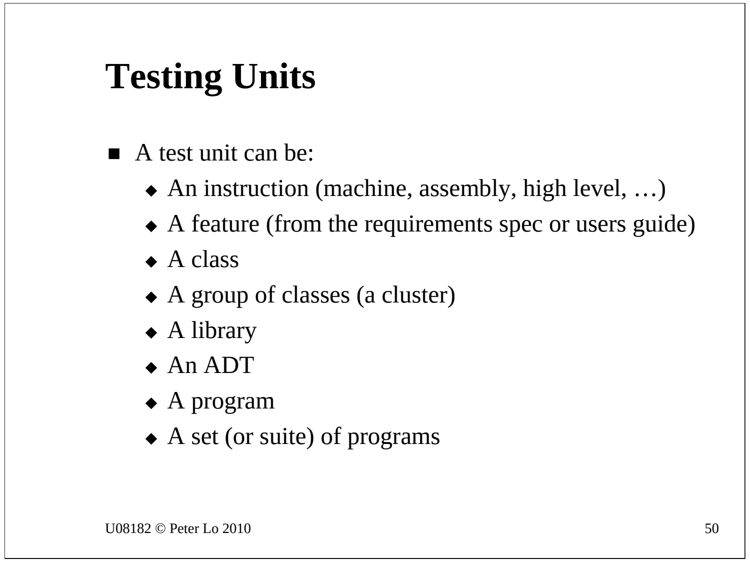


Analysis of Risks: The Clinic Example

Function	Possible errors	Consequences	Priority
Make appointment	Double booking	Patient inconvenience	Medium
	No booking on available time	Inefficiency for doctors	Medium
Cancel appointment	…	…	Low
Manage appointment	…	…	Medium
Request medicine	…	…	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

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



Testing Units

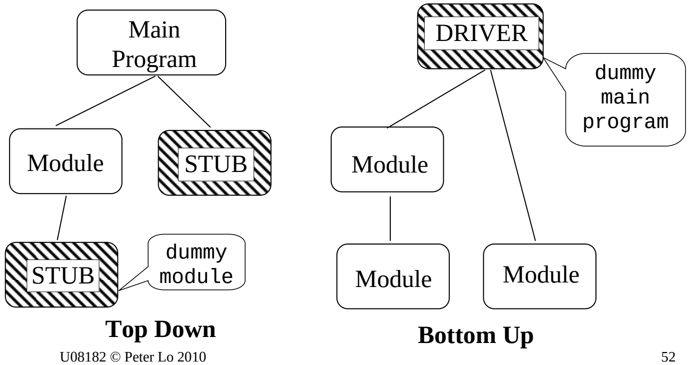
- A test unit can be:
  - An instruction (machine, assembly, high level, …)
  - A feature (from the requirements spec or users guide)
  - A class
  - A group of classes (a cluster)
  - A library
  - An ADT
  - A program
  - A set (or suite) of programs

Traditional & Object Oriented Unit

- Traditional (Structured) units are module, function, procedure etc
- Object oriented (encapsulated data + operations) 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

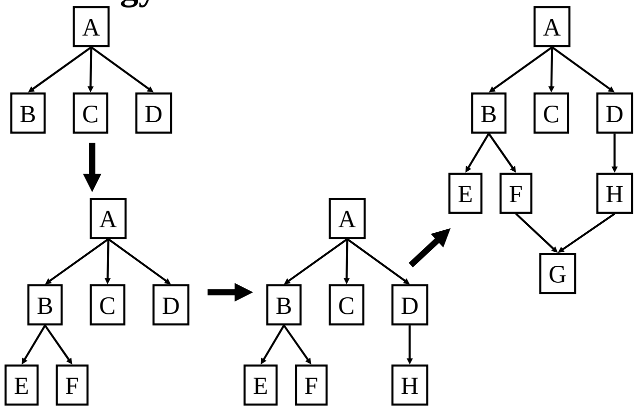
- Systematic approach to testing modular systems



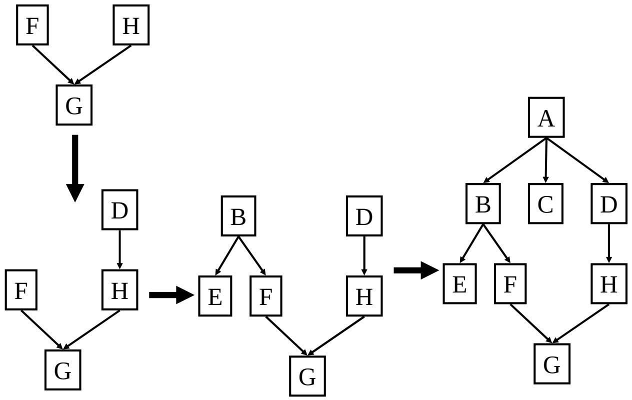
Integration Testing – In Practice

- 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

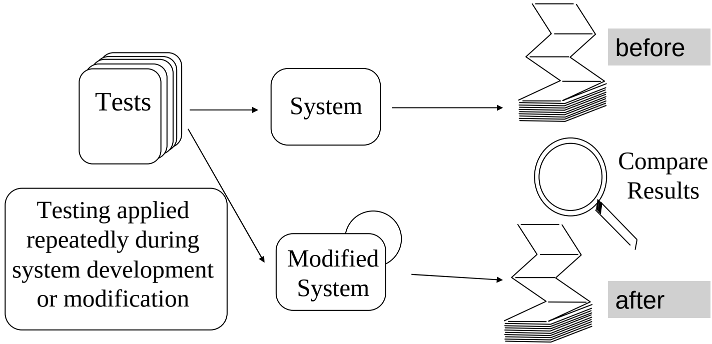


Integration Strategies: Bottom-Up Strategy



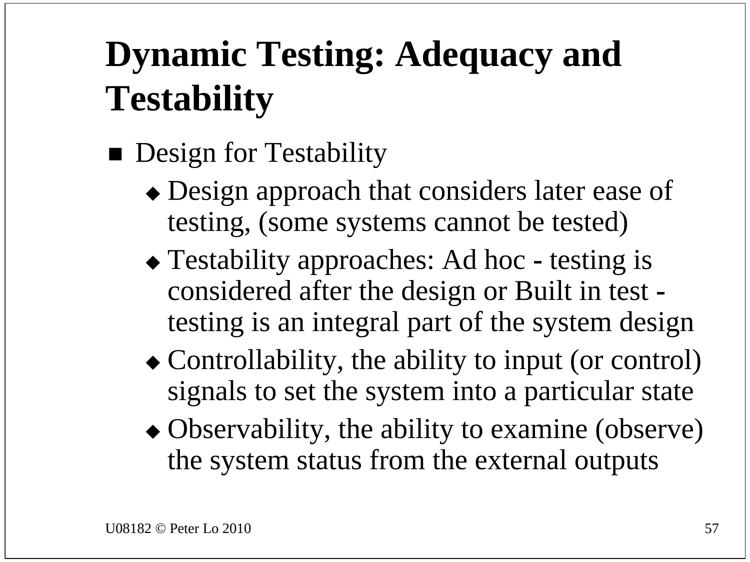
Regression Testing

- Does it still work, after the modification?


Tests
System
before
Compare Results
Testing applied repeatedly during system development or modification
Modified System
after

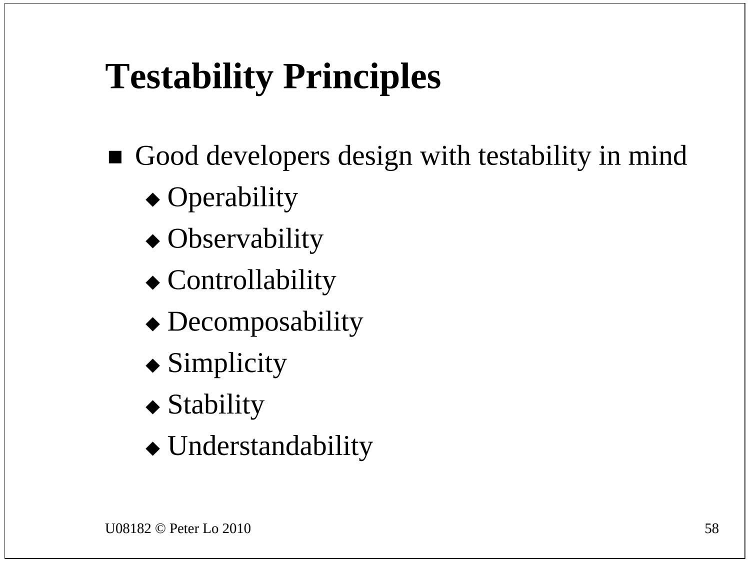
Dynamic Testing: Adequacy and Testability

- 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



Dynamic Testing: Adequacy and Testability

- 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

- Good developers design with testability in mind
  - Operability
  - Observability
  - Controllability
  - Decomposability
  - Simplicity
  - Stability
  - Understandability

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

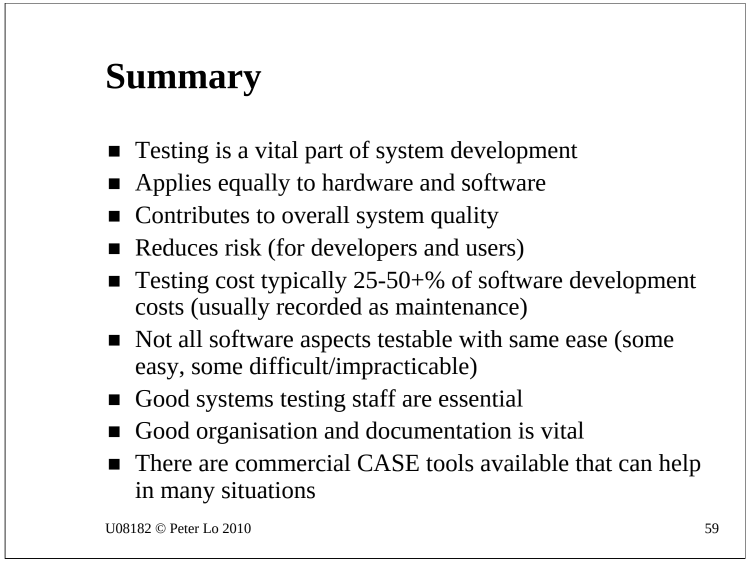
- The less there is to test, the more quickly we can test it

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