COM2070 – Software Hut

Dr Gerald Luettgen
Dr Marian Gheorghe

Topic: Software Testing & Test Management

Idea: Focus mainly on general principles & widely accepted procedures

Testing is an activity within

Validation & Verification

(Boehm1981)

Validation: Are we building the right product?

Software product must be traceable to customer requirements

Verification: Are we building the product right?

Software product correctly implements specified functions

In addition to testing, V&V encompasses

- formal reviews (documentation, processes, documentation)
- quality audits
- feasibility studies
- algorithmic analysis
- …

TESTING has two directions:

DEFECT TESTING

A test is successful if it reveals an error/fault

VALIDATION TESTING

A test is successful if the system performs the given test set correctly; used to validate a design or implementation against a specification – ex X-machine testing

This lecture will introduce/review

TESTING TECHNIQUES

TESTING STRATEGIES (TEST MANAGEMENT)

OO TESTING

DEFECT TESTING

Demonstrate the presence, not the absence, of faults

TEST CASE:

- Specification of the input to the test
- Specification of the expected output
- Statement of what is being tested

TEST DATA:

- Input data according to test specification
- Generated manually or automatically

TESTING POLICIES

- Exhaustive testing is impractical
- All program statement should be tested
- All system functions accessed through menus should be tested
- All functions relying on user input must be tested with both correct and incorrect input

BLACK-BOX TESTING

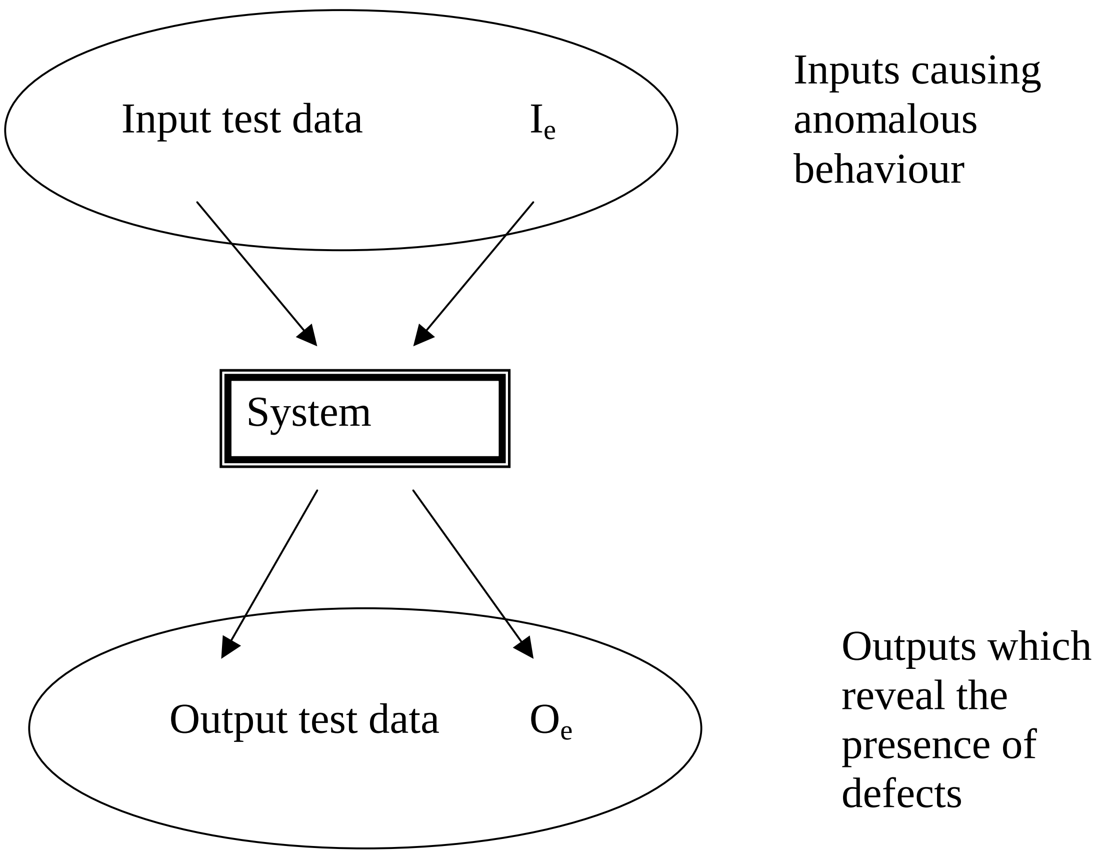
- Tests are derived from the program (components) specifications
- System behaviour can only be determined by studying its inputs and related outputs; the system is a black box

Challenge:

- Selecting inputs that have a high probability of revealing an error

Approaches:

- Apply domain knowledge
- Employ a systematic technique called equivalence partitioning



1. Equivalence partitioning (with **Boundary value analysis**)

- Partition input data into a number of different classes (one might partition integer data into negative integers, 0, positive integers)
- The partition should be done such that all input data from the same equivalence class yields an ‘equivalent’ behaviour and output

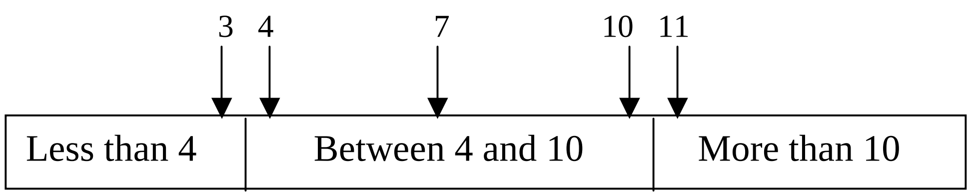
CHOOSING TEST CASES

- An arbitrary value from each class – in general a ‘mi-point’
- Might be enforced with boundary values (when an ordered set of data is an equivalence class then the first, middle and last elements are usually considered)

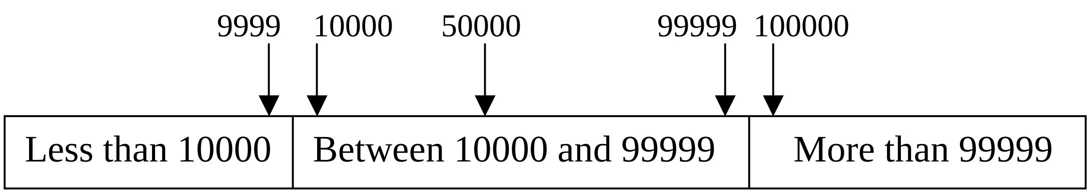
CLASS’ IDENTIFICATION

- Program specification
- User documentation
- Experience …

Example: Program accepts 4 to 10 inputs which are 5-digit integers greater than or equal to 10,000



Number of input values



Input values

Example of derivation of test cases: **`search` routine**

```
public static void search (int key, int [] t,
                           Result r)
```

Where `r` has two components: `found` of type boolean and `ind` of type integer; when `key` is found in `t` then `r.found` is becoming true and `r.ind` returns its position in `t`, otherwise `r.found` is false and `r.ind` is -1

Pre-condition

- the sequence has at least one element

Post-condition

- the element is found and referenced by `r.ind` (`r.found` is true and `t[r.ind]` contains `key`) or
- the element is not in the sequence (`r.found` is false and doesn't exist `k` such that `t[k] == key, then r.ind=-1`)

3 partition criteria:

- inputs where the key element is/is not a member of the sequence
- inputs where the sequence has length 1/greater than 1
- inputs where the key element is included in the front/middle/back of the sequence

{one might also derive test cases where the sequence is ascending/descending ordered or unordered}

Equivalence partitions for `search` routine

TEST CASES:

Array	Element	Input	Output (found, ind)
Single value	In sequence	t, key	true, 0
Single value	Not in sequence	t, key	false, ??
More than 1 value	First element in sequence	t, key	true, 0
More than 1 value	Last element in sequence	t, key	true, last
More than 1 value	Middle element in sequence	t, key	true, position
More than 1 value	Not in sequence	t, key	false, ??

TEST DATA SET:

Input sequence (t)	Input value (key)	Expected output
17	17	true, 0
17	0	false, -1
17, 29, 21, 6, 10	17	true, 0
17, 29, 21, 6, 10	10	true, 4
17, 29, 21, 6, 10	21	true, 2
17, 29, 21, 6, 10	0	false, -1

2. Graph-based Testing Method

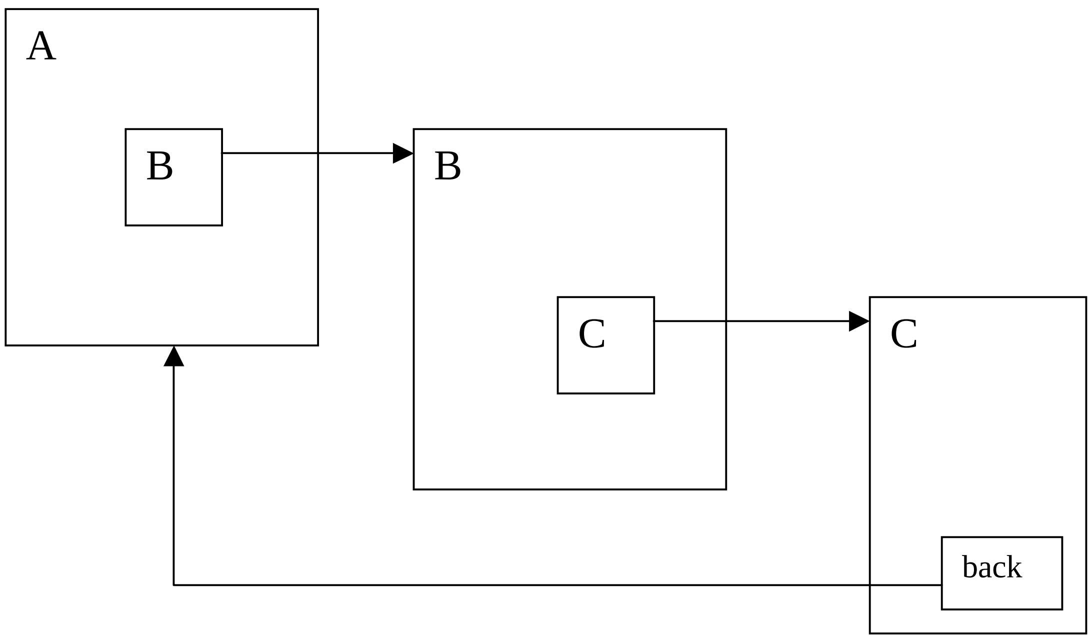
Identify (objects') states and links (transitions) between them associated to some actions; useful when testing object based systems or HCI

STATES and LINKS identification

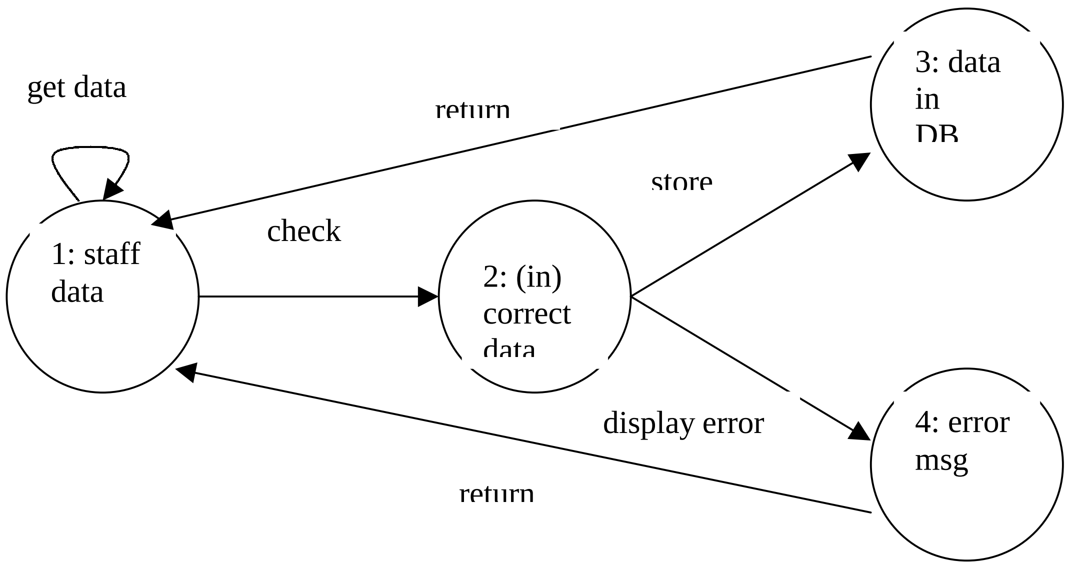
- States: (relevant) attribute values, page content, states of previous models (state models, DFD models)
- Links: operations, links between pages, transitions of previous models (state models, DFD models)

TEST CASES:

- individual links with the associated states
- paths through the state machine



Example: In a Library system data to identify a staff member (name and library number – 9 digits -) are introduced and validated; valid data are stored in a data base; if invalid data are introduced an error message is displayed and data are reintroduced



Paths like:

((get data)* check (store|display error) return)*

with the associated states/values may tested

Input state	Link(s)	Output state
1: correct name & lib no	get data; get data	1: correct name & lib no
1: correct name & lib no	check	2: correct name & lib no
2: correct name & lib no	store	3: correct name & lib no
3: correct name & lib no	return	1: no data
1: incorrect name, and/or lib no	get data; get data	1: incorrect name, and/or lib no
1: incorrect name, and/or lib no	check	2: incorrect name, and/or lib no
2: incorrect name, and/or lib no	display error	4: error message

TEST DATA SET

Input state	Link(s)	Output state
1:Marian Gheorghe, 001178514	get data; get data	1:Marian Gheorghe, 001178514
1:Marian Gheorghe, 001178514	check	2:Marian Gheorghe, 001178514
2:Marian Gheorghe, 001178514	store	3:Marian Gheorghe, 001178514
3:Marian Gheorghe, 001178514	return	1: no data
1:Marian Gheorghe, 1178514	get data; get data	1:Marian Gheorghe, 1178514
1:Marian Gheorghe, 1178514	check	2:Marian Gheorghe, 1178514
2:Marian Gheorghe, 1178514	display error	4: error 'wrong lib no'
4: error 'wrong lib no'	return	1: no data
1: ,001178514	get data; get data …	1: ,001178514
2: ,001178514	display error	4: error 'missing name'
4: error 'missing name'	return	1: no data
1:, 1178514	get data; get data …	1:, 1178514
2:, 1178514	display error	4: error 'missing name & wrong lib no'

Comments. A more elaborated table may be produced by distinguishing between input/output on the one hand and memory values used and yielded on the other hand (X-machine)

WHITE-BOX TESTING

- The tests are derived from knowledge of the software’s structure and implementation

- This testing method is suitable for small program units

- Analyse the code to find out how many test cases are needed to execute all program statements at least once

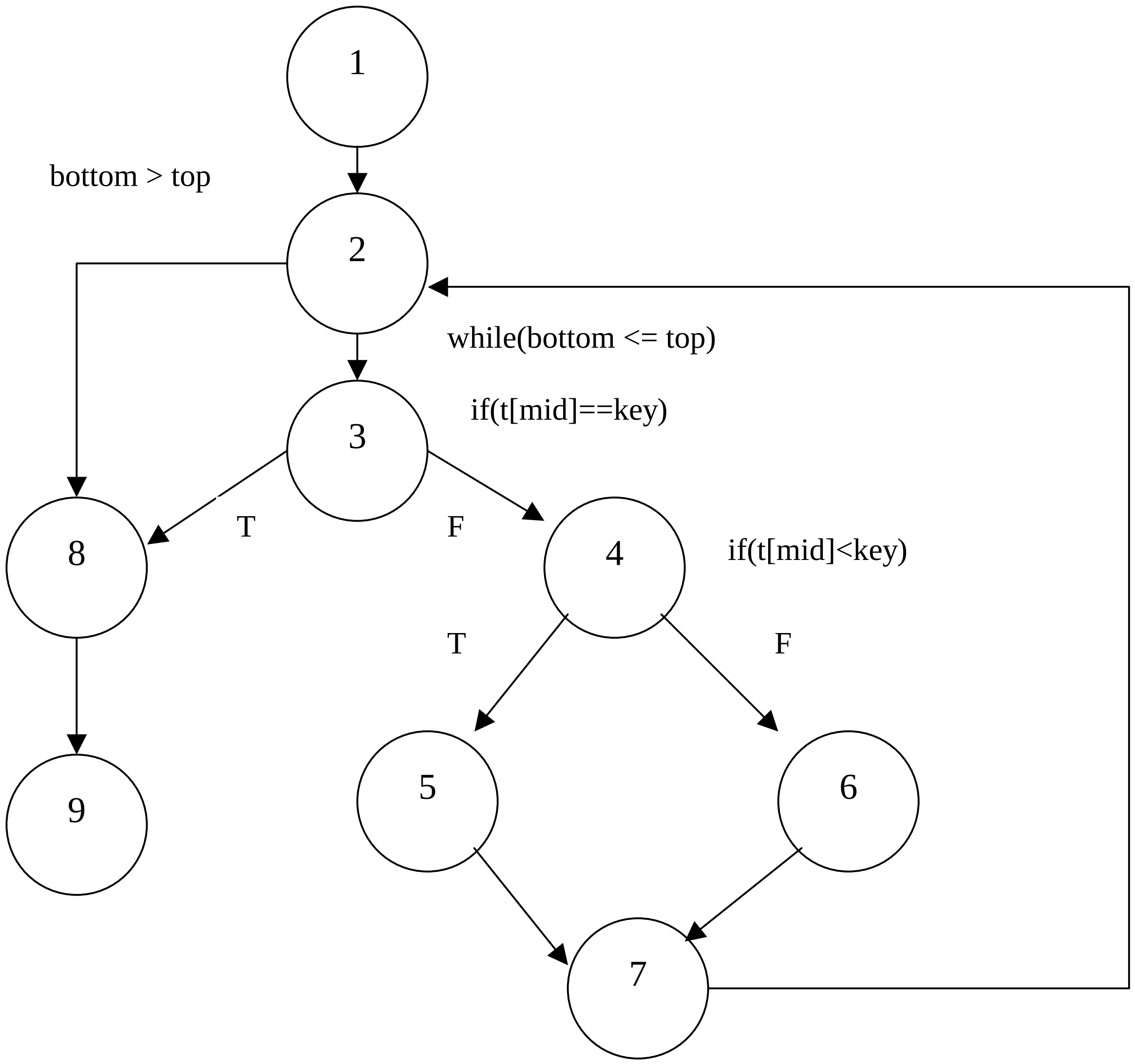
Example: binary search

```
Class BinSearch{
/* This takes an array of ordered objects and  a
key and returns an object r with 2 components
ind – the value of the array index
found – a Boolean indicating whether or not the
key is in the array
r.ind = -1 when the element is not found */

public static void search (int key, int [] t,
                           Result r)
{
  int bottom = 0;
  int top = t.length – 1;
  int mid;
  r.found = false;
  r.ind = -1;
  while (bottom <= top)
  {
    mid = (top + bottom)/2;
    if (t[mid] == key)
    {
      r.ind = mid;
      r.found = true;
      return;
    } // end if
    else
    {
      if (t[mid] < key)
        bottom = mid + 1;
      else
        top = mid – 1;
    } // end else
  } //end while
} //end search

} //end BinSearch
```

Path testing

Exercise every independent path through a component (this implies that every statement is executed at least once; in particular all conditional statements are tested for both true and false cases). These are derived from the **flow graph** (every statement type has a node type associated with)

Flow graph of `search` method of class `BinSearch` is



Independent paths	**Data to exercise**
- 1 2 3 8 9	empty sequence
- 1 2 3 4 6 7 2 8 9	t = {5,6}, key = 4
- 1 2 3 4 5 7 2 8 9	t = {5}, key = 6

These are simple paths; 1 $(2\ 3\ 4\ 5\ 7)^n$ 2 8 9 may be used
TEST MANAGEMENT

Conflict of interest

- software development is constructive
- software testing is destructive

Conclusion. Software developers should not be the same people testing the software they produced, although they know their programs best

Important principles

- have an independent test group working together with the software developers
- make the software developers responsible for testing individual program units/modules
- think about and conduct testing from the very early stages of the product cycle on; testing is associated with all SE process stages:
    - unit testing during codification
    - integration testing associated with (architectural) design
    - validation testing associated with requirements
    - system testing associated with the system as a whole

TESTING DOCUMENTATION

BE SYSTEMATIC AND RECORD WHAT YOU DO

Sample test script

Test Objective: …

Test No	Input	Expected	Actual	Analysis	Action

Part of the test strategy: assign priorities

- mandatory: must test this aspect
- desirable: should test this aspect
- beneficial: may test this aspect

This leads to the following question: when is testing complete?

NEVER! … Each time the software is run, it is tested

You end it up when

- finish up applying an employed testing strategy
- run out of time/money

UNIT TESTING

Focuses on smallest units of software design

Consider a single module and test it wrt

- interface: data flow in and out of the program unit
- integrity of local data structures
- boundary conditions
- independent path exercising each statement at least once
- error handling paths

Employ WHITE-BOX TESTING here

Tests at this level are usually conducted by the unit designer/programmer

Potential erroneous computations:

- incorrect/lack of initialisation
- wrongly assumed operators‘ precedence order
- improper use of Boolean operators
- improper or non-existent loop termination
- improper modified loop variables
- comparison of different data types, …

Potential errors in error handling

- exception-condition processing is incorrect
- error description is unintelligible or vague

Many errors are revealed when testing boundaries

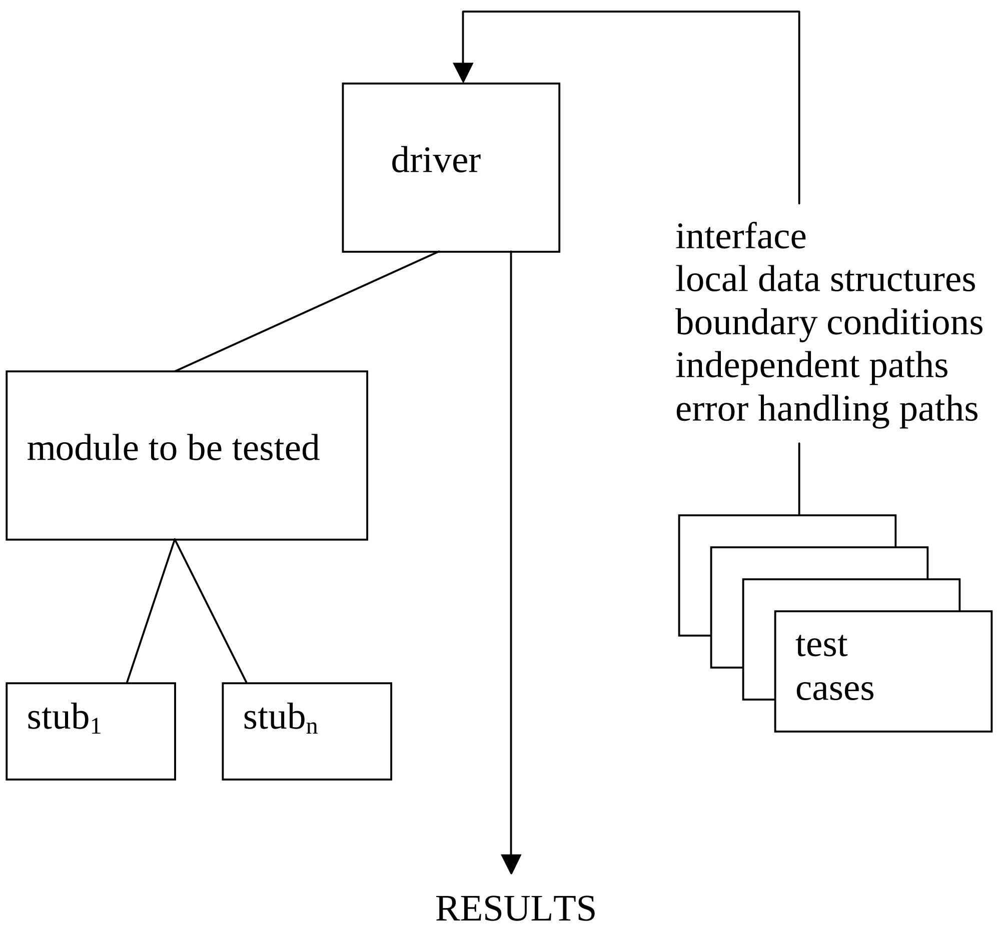
- when the first/last element of an array is processed
- when an array has one element or nothing at all
- when a loop body is evaluated for the last time
- exercise data structure, control flow just below/above maxima/minima

UNIT TESTING procedures

To test a unit, one first needs to build a stand-alone program around it by providing

- drives which accept data, pass data to the unit under testing and print out relevant results
- stubs which replace subordinate modules, partially implement some functionalities

Problem: writing drivers and stubs induces overhead and sometimes this is too expensive and testing is postponed until more units are available



INTEGRATION TESTING

Modules which work individually correct, might not behave correctly when composed/integrated with other modules due to

- error regarding interfacing
- combination of sub-functions does not yield desired function
- individual arithmetic imprecisions add up to an unacceptable amount
- global data structures …

INTEGRATION TESTING

- construct the program structure & at the same time conduct tests to uncover interfacing errors

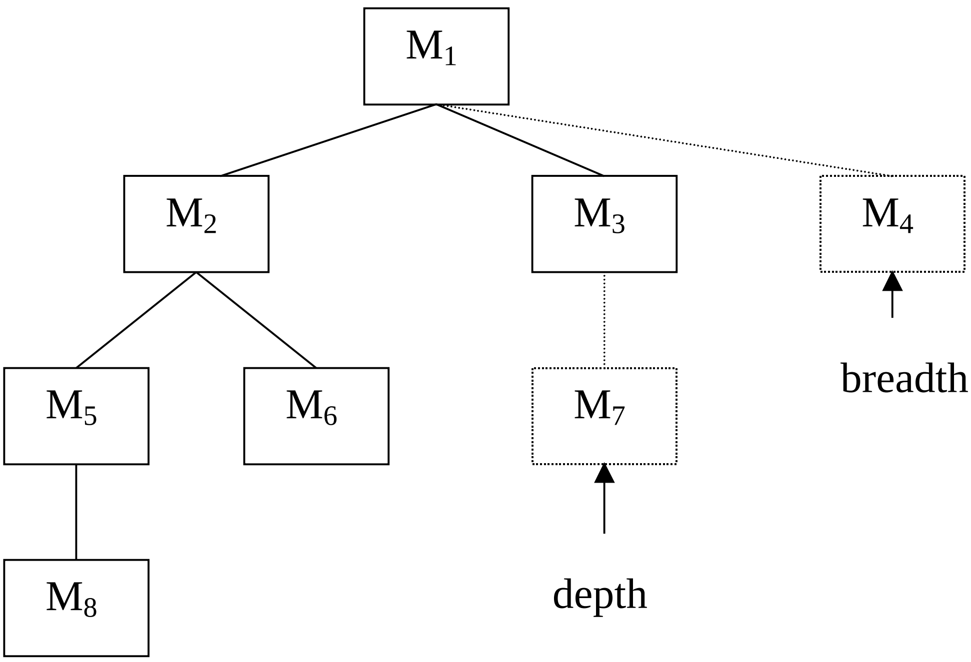
IMPORTANT

- employ a combination of black-box (units) and white-box testing (paths between units)
- integrate incrementally (big bang = big surprise!)

TOP-DOWN INTEGRATION

Means

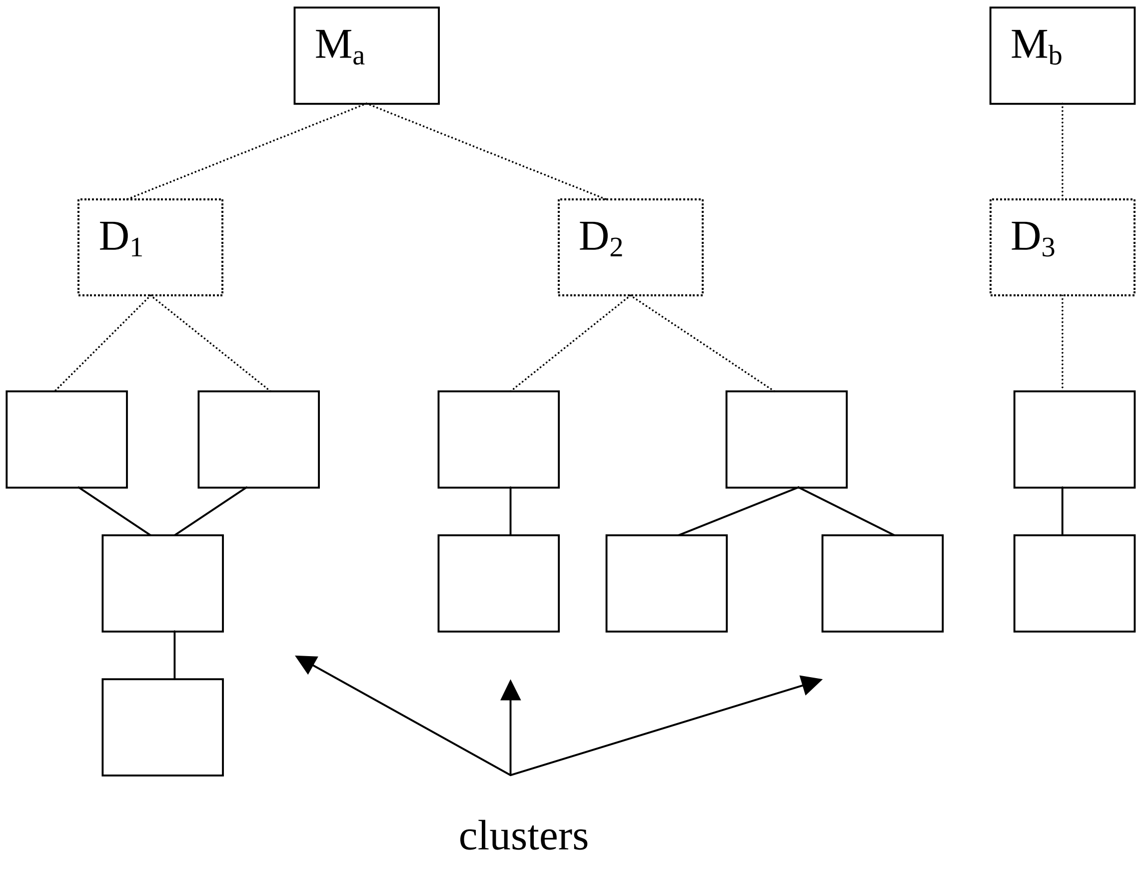
- moving downward through the control hierarchy, beginning with the main module, which also acts as a test driver
- proceeding either in a
  - depth first fashion or
  - breadth first fashion
- employing stubs which are successively replaced by real components



BOTTOM-UP INTEGRATION

Means

- starting construction and testing with atomic units; moving upwards
- combining several units into clusters before testing; this keeps the necessary drivers simple
- replacing drivers successively by real modules/clusters upwards



TOP-DOWN VS BOTTOM-UP INTEGRATION

Major disadvantages

Top-Down
- need for stubs

Bottom-Up
- program as an entity doesn't exist until the last module is added

Major advantages

Top-Down
- tests major control functions early

Bottom-Up
- lack of stubs

Combined Top-Down/Bottom-Up approach
- Top-Down for upper levels
- Bottom-Up for lower levels

Identify 'critical' modules and integrate them early

REGRESSION TESTING

- Because software continuously changes during integration then re-execute subsets of tests that have already been conducted to ensure that changes have not propagated unintended side-effects

How to decide on subsets of test cases?

Include

- representative sample of tests that exercise all software functions
- additional tests regarding software functions which are likely to be affected by the change
- tests regarding the software components that have been changed

VALIDATION TESTING

Validation testing succeeds when

Software functions are implemented in a manner that can be reasonably expected by the customer; as agreed in the requirements document – refer mostly to use cases

Validation testing employs black-box testing

Test

- functional requirements
- behavioural characteristics
- performance requirements
- documentation
- …

Large software projects, where products are developed for multiple customers, often employ alpha & beta testing

- with customers as testers
- alpha testing at software developers’ site
- beta testing at customers’ site

SYSTEM TESTING

The main purpose is to fully exercise the computer-based system in the **client's environment**

Recovery testing

Provoke different kinds of failure and check the consequences

Security testing

Provoke similar effects as 'intruders' can cause

Stress testing

Confront programs with abnormal situations regarding quantity, frequency, volume (of data/transactions)

Performance testing

Test performance issues: speed, use of resources, time spent (to perform some tasks)

OBJECT-ORIENTED TESTING STRATEGIES

In the classical approach testing computer software starts with 'testing in the small' – **unit testing** – and works outward toward 'testing in the large' - progresses toward **integration testing**, ending with **validation** and **system testing** -.

Testing in OO context addresses UNIT and INTEGRATION testing

UNIT testing in OO context

The smallest testable unit is the class; a method cann't be tested in isolation – a method defined in a superclass may be used by its subclasses in different contexts

INTEGRATION testing

The classical top-down and bottom-up approaches are not appropriate as OO doesn't provide a hierarchical structure of the software product

Alternative solutions are:

- thread-based testing: integrates those classes responsible to respond to the same inputs/events
- use-based testing: first integrate those independent classes (do not use other classes or only some server classes); at the next are integrated those using independent classes
- cluster testing: a set of classes collaborating are integrated in one step

Three types of faults are encountered during integration testing:

- unexpected results
- wrong operation used
- incorrect invocation

CONCLUSIONS

- Testing is a ‘destructive’ activity, time- and resource-consuming, but must be done

- Testing can be done systematically employing some strategies

- Testing requires some creativity in identifying the most likely used components and setting adequate tests for them

- Testing never ends; there will always be some unintended bugs in the code; make sure they are inconsequential