4B Summary of some Key SW Testing Concepts[1]

Note: **Section 4A** presented a particular, specific method for developing (system or requirements level) tests cases from UML use cases.

Section 4B provides a very brief overview of software testing in general.

Finally, **Section 4A+** provides further material on system level software testing based on use cases. The terminology is not exactly as in **Section 4A** but should nevertheless give additional insight.

4B Summary of some Key SW Testing Concepts 1
4B.1. Basic Concepts 2
What should be in a test case? 2
Behaviour (Function) versus Structure 2
Identifying test cases 3
4B.2. FUNCTIONAL TESTING techniques 5
4B.2.1 Boundary Value and Related Methods 5
4B.2.1.1 Boundary Value Analysis 5
4B.2.1.2 Robustness Tests 5
4B.2.1.3 Worst Case Testing 6
4B.2.1.4 Special Value Testing 6
4B.2.1.5 Random Testing 6
4B.2.1.6 Concluding notes on boundary value analysis etc 6
4B.2.2 Equivalence Class Testing 7
4B.2.3 Decision Table-Based Testing 8
4B.3. STRUCTURAL TESTING 10
4B.4. Some Issues in Object Oriented Testing 11
Class as the choice of unit for object oriented testing 11
Implications of inheritance 11
Implications of polymorphism 12
UML Interaction Diagrams as a basis for integration testing 13
Note: Object-oriented system testing & UML use cases 13

[1] Notes in this section draw heavily on "Software Testing, a craftsman's approach", Jorgensen, P.C., CRC Press, 2002.

4B.1. Basic Concepts

What should be in a test case?

Headings for the information to be provided in a test case are

Heading	
Test Case ID	
Purpose	*e.g. to verify some SW requirement*
Preconditions	
Inputs	*Identified by some testing method*
Expected Outputs	
Postconditions	
Execution History	

Date	Result	Version	Run By
•	•	•	•
•	•	•	•

Behaviour (Function) versus Structure

Structural view focus is on what software is. Behavioural view is on what software does.

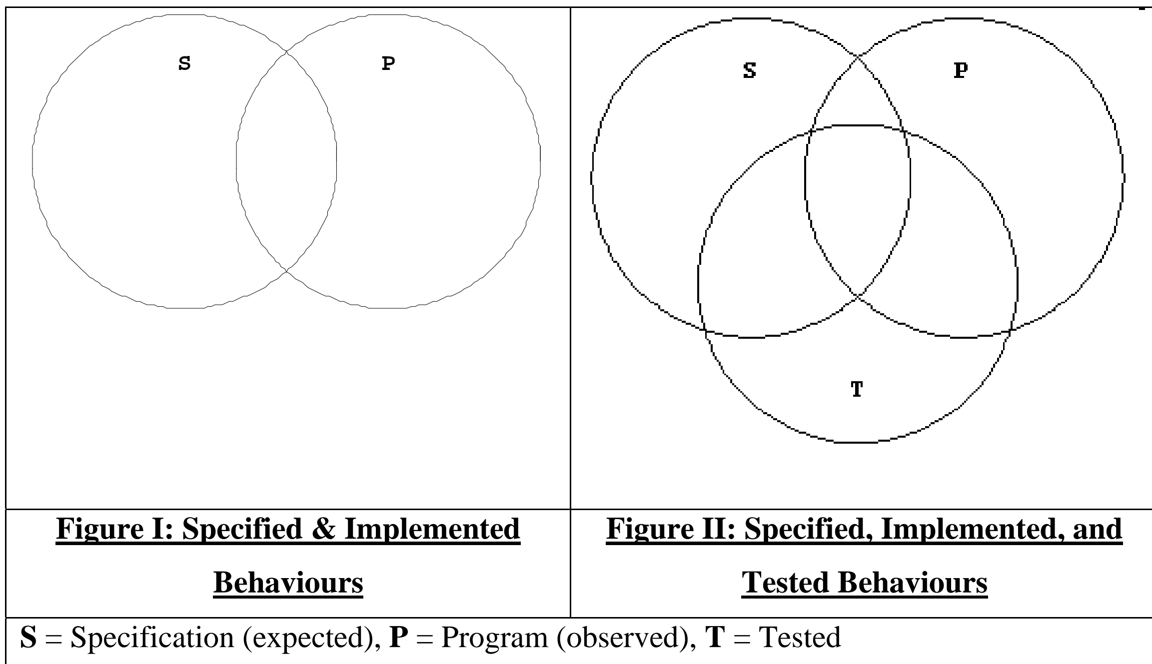


Figure I: Specified & Implemented Behaviours	**Figure II: Specified, Implemented, and Tested Behaviours**

S = Specification (expected), **P** = Program (observed), **T** = Tested

Figure I is intended to highlight issues that a program tester should be aware of in devising tests. The correct portion is where sets **S** and **P** intersect, but
What if certain specified behaviours have not been programmed?
What if some implemented (i.e. programmed) behaviours have not been specified?

Figure II has an additional set **T**, representing the set of test cases. There are several possibilities:

a.Specified behaviours that **are not** tested	*Part of **S** not in **T***
b.Specified behaviours that **are** tested	*Part of **S** that is also in **T***
c.Test cases that correspond to unspecified behaviours	*Part of **T** not in **S***
d.Programmed behaviours that **are not** tested	*Part of **P** not in **T***
e.Programmed behaviours that **are** tested	*Part of **P** that is also in **T***
f.Test cases that correspond to unprogrammed behaviours	*Part of **T** not in **P***

Some questions that arise:
What can a tester do to maximise the region where **S**, **P** and **T** all intersect? [Not just up to the tester?]
How are the test cases in **T** identified?

Identifying test cases

Functional testing is based on the view that any program can be considered to be a function that maps values from its **input domain** to values in its **output range**. This leads to the term **black box testing**, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs. In this functional approach to test case identification, the only information used is the software specification. In Figure II, set **T** would be contained within set **S**.
In **Structural testing**, the implementation is known and used to identify test cases. Hence, it is sometimes called **white box testing**. In Figure II, set **T** would be contained within **P**.
In fact, a combination of functional and structural testing is usually the best approach. The following is a summary some of the issues:

Advantages of functional approach	(i) test cases are not affected by changes of implementation (ii) test case development can proceed in parallel with implementation.
Common difficulties of functional approach	(i) Significant redundancies may exist between test cases (ii) May be gaps in that some of the software may be untested (iii) Will never uncover unspecified behaviour (*e.g. a virus!*)
Two different positions on structural testing!	(i) "This tool has been wasting tester's time since the 1970s …" (ii) "Branch coverage [arising in structural testing], if attained at the 85% or better level, tends to identify twice the number of defects that would have been found by 'intuitive' [functional] testing"
One disadvantage of structural approach	Hard to see how unprogrammed behaviours could be identified.
Characterisation of functional approach	Establishes confidence
Characterisation of structural approach	Seeks faults

4B.2. FUNCTIONAL TESTING techniques

It was indicated above that "**Functional testing** is based on the view that any program can be considered to be a function that maps values from its **input domain** to values in its **output range**". Very often, because it is more straightforward, this form of testing has focused on the input domain but, if possible, tests based on the output range should also be developed.

4B.2.1 Boundary Value and Related Methods

4B.2.1.1 Boundary Value Analysis

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extremes of an input variable. Loop conditions, for example, may test for < when they should test for ≤, and counters are often "off by one".

Normally, for each input variable, there are 5 tests cases corresponding to

minimum value (min)
value just above minimum (min+)
a nominal or typical value
value just below maximum (max-)
maximum value (max)

If there are 2 input variables there will be a total of 4*2+1=9 test cases (nominal is common). If there are n input variables there will be a total of 4*n+1 test cases.

Strongly typed languages permit explicit definition of variable ranges but other languages, such as C, do not; boundary value testing is more appropriate for the latter.

4B.2.1.2 Robustness Tests

At its simplest, robustness testing is an extension of boundary value analysis where we investigate what happens when the extrema are exceeded slightly.

More generally, the objective of robustness testing is to demonstrate the ability of software to respond to abnormal inputs and conditions. For example, for state transitions, test cases should be developed to provoke transitions that are not allowed by the software requirements.

4B.2.1.3 Worst Case Testing

A limitation or criticism of boundary value analysis is that it only works well when the program to be tested is a function of several ***independent*** variables. However, there are often interesting dependencies between variables.

"Worst case analysis" focuses on what happens when more than one variable has an extreme value.

For example, if a program depends on 2 variables x and y then we have

(a) Boundary value analysis: 4*2+1=9 test cases (see above)

(b) Worst case analysis: 5*5 = 25 test cases

[{(min_x, min_y), (min_x, $min+_y$}, (min_x, $nominal_y$), (min_x, $max-_y$), (min_x, max_y), etc]

If there were n variables then a total of 5^n tests cases would be needed for the worst case analysis, compared to 4n+1 for the boundary value analysis.

4B.2.1.4 Special Value Testing

This "ad hoc" approach occurs when testers use their domain knowledge or experience with similar programs or information about "soft spots" to devise test cases. While subjective, can be fruitful.

4B.2.1.5 Random Testing

The basic idea is to use a random number generator to pick out test cases rather than using a deterministic approach, particularly the boundary value approach. While this avoids a bias in testing it raises the question of "how many random cases are sufficient?".

4B.2.1.6 Concluding notes on boundary value analysis etc

Each of the foregoing can, in principle, be applied to the output range of a program.

Another form of output-based testing is to devise test cases to check that error messages are generated when appropriate and are not falsely generated.

Domain analysis can also be used for internal variables (e.g. loop control variables, indices, etc). Of course, this brings up the issue of "visibility" of such variables to the testing software.

4B.2.2 Equivalence Class Testing

Formally, in set theory, a ***partition*** of a set means a collection of mutually disjoint subsets whose union is the entire set. Equivalence classes form such a partition.

From a testing point of view there are 2 points to note:

a) “Entire set is represented” provides a form of completeness in the test case.

b) “Equivalence classes are disjoint” ensures a form of non-redundancy in test cases.

Example: Suppose a function F depends on two variables x and y and is implemented as a program. Suppose that input variables x and y have the following boundaries, and (meaningful) intervals within the boundaries:

$a \le x \le d$, with intervals $[a, b)$, $[b, c)$, $[c, d]$

$e \le y \le g$, with intervals $[e, f)$, $[f, g]$.

Note: “[“ denotes a closed (i.e. included) end-point, “(“ denotes an open (i.e. excluded) endpoint.

For completeness, we note the invalid intervals:

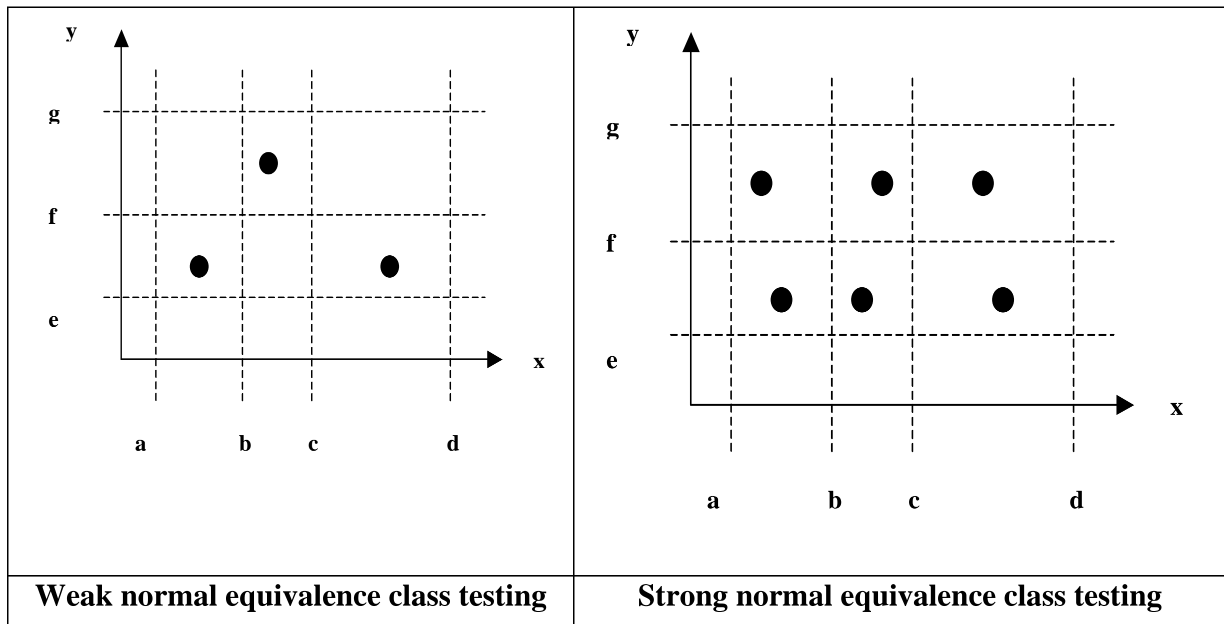
$x < a$ or (∞, a), $x > d$ or (d, ∞)

$y < e$ or (∞, e), $y > g$ or (g, ∞)

It is clear that the intervals are in fact equivalence classes – for example,

(∞, a), $[a, b)$, $[b, c)$, $[c, d]$, (d, ∞)

divides the whole x axis into 5 disjoint intervals.



Weak normal equivalence class testing	**Strong normal equivalence class testing**

4B.2.3 Decision Table-Based Testing

Decision tables have the general form,

CASE	ID1	ID2	- - -
Condition 1			
Condition 2			
:			
Condition M			
Action 1			
Action 2			
:			
Action N			

The top row provides a reference for each column of the table. The "conditions" rows specify the variables or conditions to be evaluated while the lower rows specify the corresponding actions to be taken. A table entry of "x" represents "don't care", while “u” represents “unchanged” (w.r.t. value before the table is executed). For example, the pseudocode

```
if A then
    Compute x
elseif B then
    Compute y
else
    Compute x
    Compute y
end if
```

would be represented by the decision table,

CASE	1	2	3
A	T	F	F
B	x	T	F
Compute x	T	F	T
Compute y	F	T	T

It is believed that, in the case of complex logic at least, decision tables provide a clearer less error prone representation than pseudocode or activity diagrams (or equivalent). Moreover, decision tables do ***not*** require that arbitrary and unnecessary sequential constraints be inferred.

Decision tables can be formed by analysis of a requirement specification, a use case description, pseudocode (as above) or actual source code. In fact, it could be decided at

the outset to express requirements with the help of decision tables as illustrated by the following example from an actual project:

S-1.6.1-2
Eclipse_Imminent_Logic_T[IN:**Eclipse_Imminent**, INOUT: **IMU_On**[roll]] and **Eclipse_Imminent_Logic_F**[IN:**Eclipse_Imminent**, INOUT: **IMU_On**[roll]] are common functions that shall be used to switch the roll gyro on and off depending on whether or not an eclipse is imminent. They are, respectively, defined by the decision tables,

Eclipse_Imminent_Logic_T:

Case	1	2	3
Eclipse_Imminent	T	T	F
IMU_On[roll] (in)	T	F	x
IMU_On[roll] (out)	T	T	u
IMUturn on[roll]	F	T	F

(Remark: The value of "**IMU_On**[roll] (out)" in case 2 is the nominally expected output but whether it is achieved depends on the result of executing **IMUturn on**[roll])

and **Eclipse_Imminent_Logic_F**:

Case	1	2	3
Eclipse_Imminent	F	F	T
IMU_On[roll] (in)	T	F	x
IMU_On[roll] (out)	F	F	u
IMUturn off[roll]	T	F	F

TRACE: F3.1.2.1.2.3.4-12(part of sunlit), F3.1.2.1.2.3.5-1(part of sunlit),#

In terms of testing, the basic approach is that there will be a test case for each decision table column. Thus, there would be three test cases for the pseudocode example above.

4B.3. STRUCTURAL TESTING

This is presented very briefly.

According to Jorgensen (see above) “the distinguishing characteristic of structural testing methods is that they are all based on the source code of the program tested, and not on the definition”.

In fact, it is probably useful to bear in mind two aspects:

a) Derivation of test cases based on analysis of the software design, whether at high level (e.g. object-object interactions described by UML sequence diagrams) or detailed level (e.g. the pseudocode for an individual method)

b) Determination of the level of structural test coverage achieved (which can in principle be found automatically via automatic tools).

There are various “structural test coverage” metrics including

Metric	**Description of Coverage**
C_0	Execution of every statement
C_1	Execution of every decision-to-decision (DD) path
C_{MCC}	Execution of every condition in every decision
C_{ik}	Execution of every program path that contains up to k repetitions of a loop
C_{stat}	Execution of a “statistically significant” fraction of paths

Particularly in terms of integration, it should be an objective to achieve adequate test coverage of data and control coupling between code components (e.g. classes).

4B.4. Some Issues in Object Oriented Testing

Class as the choice of unit for object oriented testing

In traditional, non-object oriented programming it was common to take individual functions or procedures as units for testing. However, in OO programming, it is usual to take classes as the test units. Some advantages of this are:

- A class often has an associated state (or statechart) diagram that can be very helpful in testing
- The goal of integration testing is clearer, being to check the cooperation of separately tested classes.
- A class can be compiled as a separate entity (for the purpose of unit testing).

In this scenario, testing of individual methods is a sub-element within unit testing.

Implications of inheritance

If a given class inherits attributes and/or operations from super class(-es), the standalone compilation criterion of a unit is sacrificed.

One suggestion to get around this is to use “flattened classes”, where classes are expanded to include all the operations and attributes they inherit. However, this leads to complications including that a flattened class will not be part of the final system. Also, it may lead to duplication of testing effort such as testing the same method in a super class and then in a flattened class. For example,

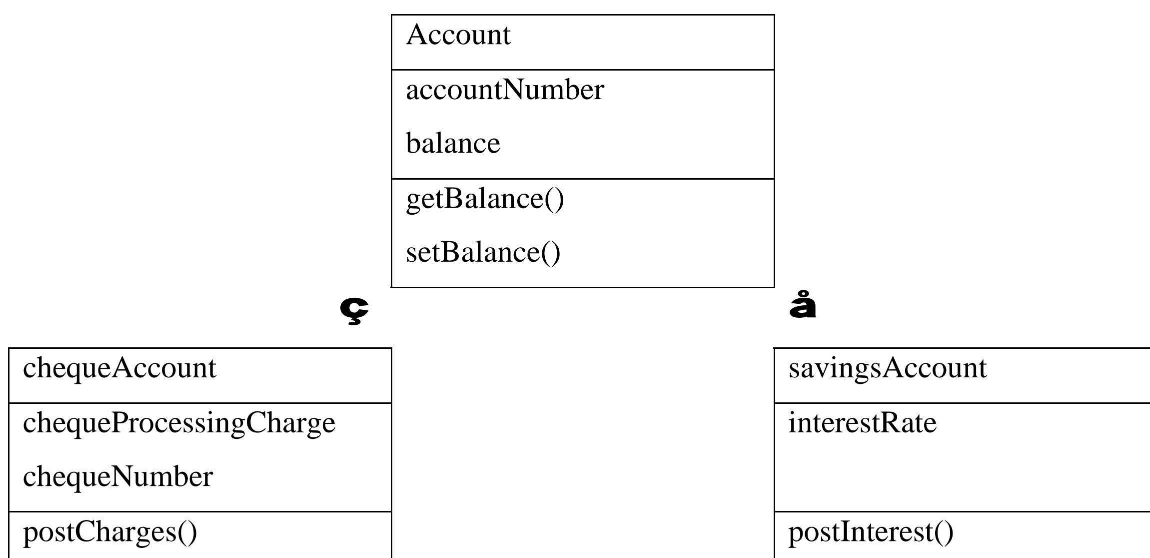


Figure 4B.4-1: Class inheritance example

We wish to test the subclasses “chequeAccount” and “savingsAccount” but we find that if we do not “flatten” (i.e. if we tried to test only the non-inherited parts) these classes we would not be able to access or change the balances. The “flattened” classes are

chequeAccount
accountNumber balance chequeProcessingCharge chequeNumber
getBalance() setBalance() postCharges()

savingsAccount
accountNumber balance interestRate
getBalance() setBalance() postInterest()

Figure 4B.4-2: Flattened subclasses

These are stand-alone units that are sensible to test but we do test some operations twice thereby losing some of the hoped-for economies.

Implications of polymorphism

The essence of polymorphism is that the same method applies to different objects. Considering classes as units implies that any issues of polymorphism will be covered by the class/unit testing. Again, the redundancy of testing polymorphic operations sacrifices hoped-for economies.

UML Interaction Diagrams as a basis for integration testing

We just note that both UML collaboration and particularly sequence diagrams form a very good basis for object oriented integration testing. For example, consider the sequence diagram:

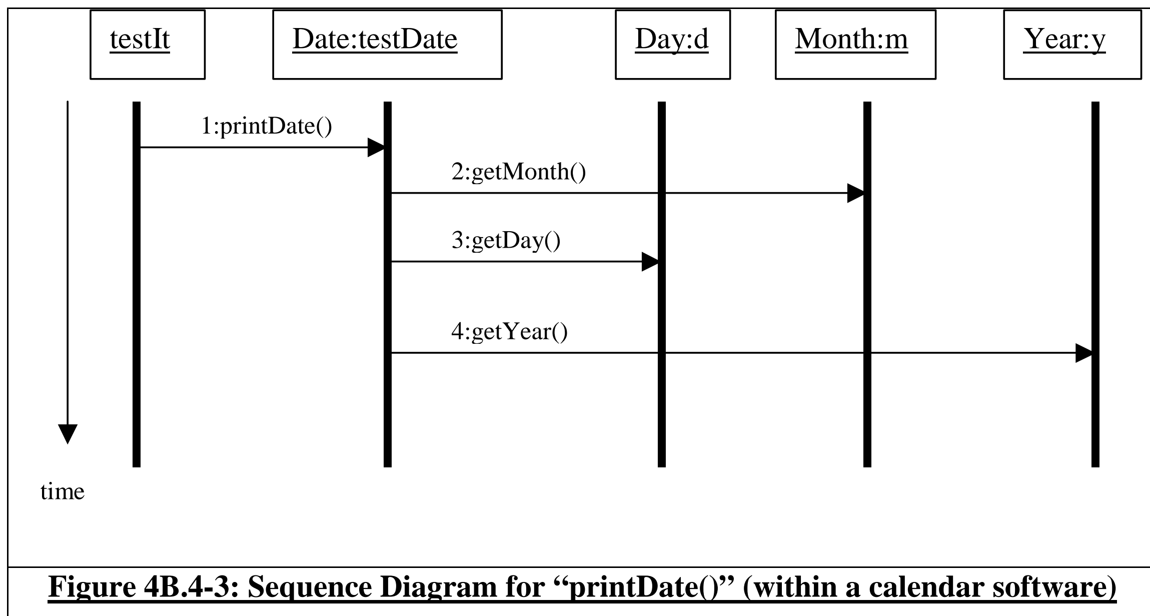


Figure 4B.4-3: Sequence Diagram for "printDate()" (within a calendar software)

An actual integration test for this sequence diagram would have pseudocode similar to:

```
1. testDriver
2.        m.setMonth(1)
3.        d.setDay(15)
4.        y.setYear(2002)
5.        Output("expected value is 1/15/2002"
6.        Output("actual output is")
7.        Date.printDate()
8. End testDriver
```

Statements 2, 3 and 4 use previously unit-tested methods to set the expected output in the classes to which messages are sent.

Note: Object Oriented System Testing & UML Use Cases

We have already had one example of this and a printed handout (**Section 4A+**) will present another.