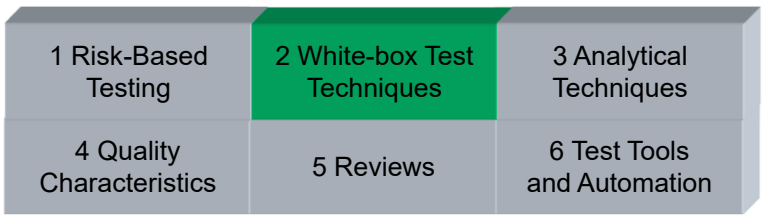
White-box Test Techniques



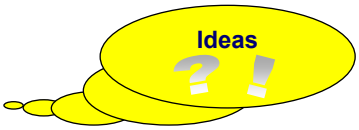
2.1

Contents

2.2

2.1 Test technique categories (Foundation level 2019)

black-box
(specification-based / behaviour-based)
- behaviour observed at interface with SUT
- don't care what happens inside the 'box'



experienced-based
- behaviour observed at interface with SUT
- may explore visible elements of structure

white-box
(structure-based / structural / glass-box)
- internal behaviour is considered

2.3

2.1 White-box test techniques

- all are dynamic
- each one focuses attention on a specific structural item
  - code statements, decision outcomes, paths etc.
- provide a systematic way of deriving test cases
  - ensure the next test case contributes something new
- thoroughness of the resulting tests can be measured
  - count the number of structural items 'covered'
    - ► compare with total number of items: report percentage
  - different techniques have different relative strengths
    - ► the stronger the technique the more thorough the testing

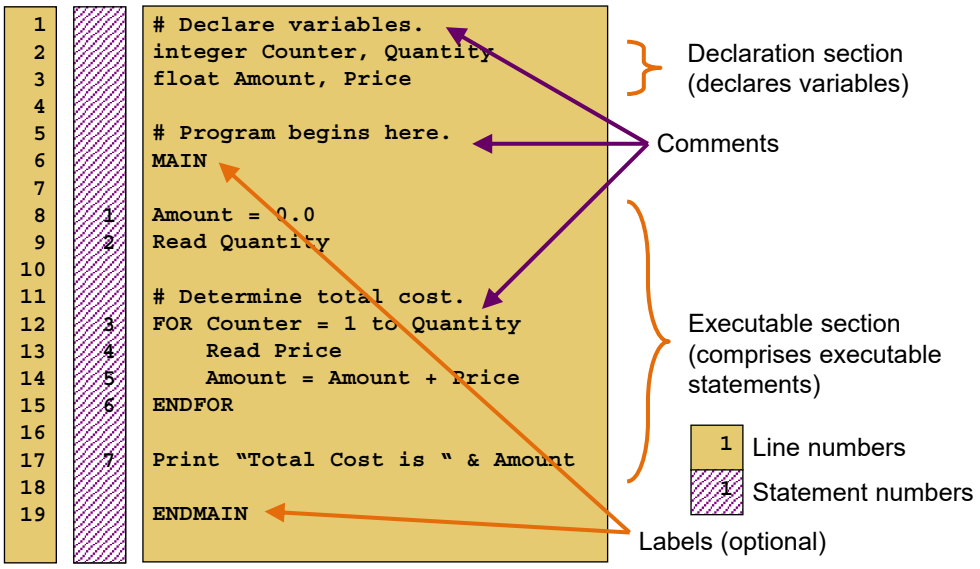
2.4

2.1 Control flow diagrams

- two styles of diagram: charts & graphs
  - pictorial representation of the potentially many paths through a piece of source code
  - show the decision points and paths through code
  - don't show the detail of the code
- uses
  - for analysis of correctness
  - highlight certain types of defect
  - help with code design, testing and maintenance
- control flow charts are usually more useful for white-box techniques
  - control flow graphs will be seen in Session 3

2.5

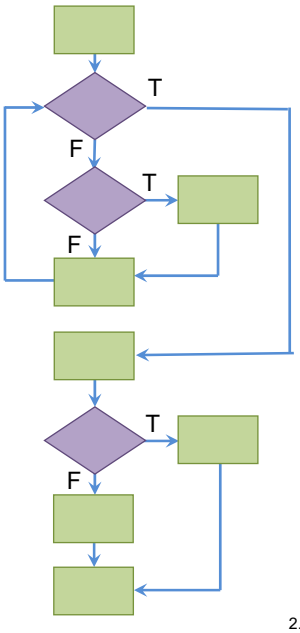
2.1 Anatomy of source code



2.6

2.1 Control flow charts

- uses rectangles and diamonds
  - diamonds represent decision statements
    - ► one decision statement per diamond
    - ► two ways out
  - rectangles represent all other statements
    - ► one or more statements per rectangle
    - ► one way out
- also uses arrows
  - represent branches, i.e. indicate possible order of execution of statements



2.7

Contents

2.8

2.2 Statement testing

- based on source code
  - purpose: identify test cases that exercise executable statements
  - executable statements are compiled into instructions
    - ► make something happen
    - ► not comments, subroutine names, declarations, endwhile etc.
- $Statement\ Coverage = \frac{Number\_of\_statements\_exercised}{Total\_number\_of\_statements}\ x\ 100\%$

Statement coverage is normally measured by a tool

- e.g.: code has 4 executable statements
  - ENDIF not executable
  - test with (a = 7) exercises all 4 of them
    - ► statement coverage = 100%
  - test with (a = 6) exercises only 3 of them
    - ► statement coverage = 75%

1	read(a)
2	IF a > 6 THEN
3	b = a * 2
4	ENDIF
5	print b

2.9

2.2 Statement testing with control flow chart

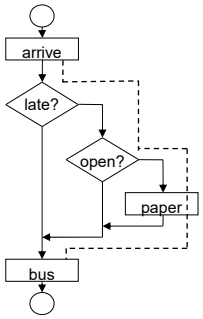
- to see statement coverage manually need control flow chart

Pseudocode:

```
Arrive at bus station
If bus is late
    If shop is open
        Buy newspaper
    End If
End If
Go to bus
```

How many test cases are needed for 100% statement coverage?

Control flow chart ➔



Dashed line = flow of test case which covers all statements in the code

For 100% statement coverage we need only 1 test case

2.10

2.2 Statement testing: considerations

- Applicability
  - should be considered as a minimum for all code being tested
- Limitations/Difficulties
  - decisions not considered
    - ► even 100% statement coverage may not detect some defects in the logic
- Example defects
  - wrong data used
  - wrong message output
  - wrong calculation performed

2.11

Contents

2.12

2.3 Decision testing

- based on source code
  - purpose: identify test cases that exercise decision outcomes
  - each different outcome will take a different path through the next part of the code
- $Decision\ Coverage = \frac{Number\_of\_decision\_outcomes\_exercised}{Total\_number\_of\_decision\_outcomes} x\ 100\%$

Decision coverage is normally measured by a tool

- e.g.: code has 2 decision outcomes
  - test with (a = 7) exercises 1 of them
    - ► statement coverage = 50%
  - test with (a = 6) exercises the other 1
    - ► statement coverage now = 100%

```
1  read(a)
2  IF a > 6 THEN
3     b = a * 2
4  ENDIF
5  print b
```

2.13

2.3 Decision testing with control flow chart

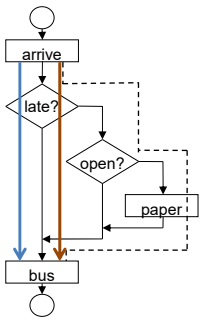
- see it manually with a control flow chart

Pseudocode:

```
Arrive at bus station
If bus is late
    If shop is open
        Buy newspaper
    End If
End If
Go to bus
```

How many test cases are needed for 100% decision coverage?

Control flow chart ➔



Dashed line = flow of test case which covered all statements in the code

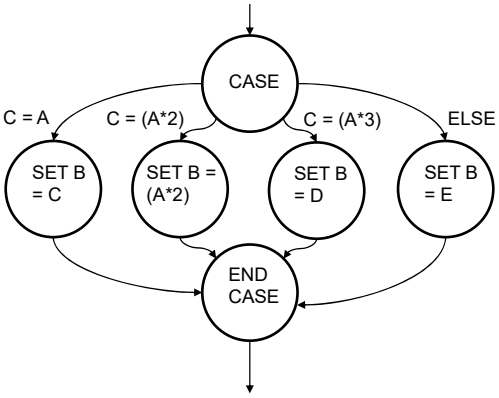
For 100% statement coverage we need 2 more = 3 test cases

2.14

2.3 Decision testing: case statements

- case statement is decision with multiple possible outcomes

```
CASE
        IF C = A
                SET B = C
        IF C = (A * 2)
                SET B = (A*2)
        IF C = (A * 3)
                SET B = D
        ELSE
                SET B = E
ENDCASE
```

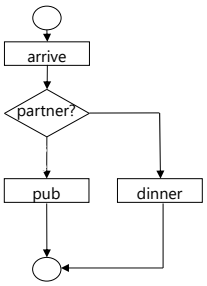


- May be > 3 outcomes so can't easily represent with diamond box
- so instead of a control flow chart we use a control flow graph
- 1 test case needed for each condition, including default, so 4 needed here

2.15

2.3 Decision testing: considerations

- Applicability
  - when the code is important or critical
  - guarantees statement coverage at 100%
    - ► reverse may be true but not guaranteed
      - Arrive home
      - IF partner home
      - Help prepare dinner
      - ELSE
      - Go to pub



- Limitations/Difficulties
  - may require more test cases than statement coverage
    - ► may be problematic when time is an issue
  - does not test how a decision with multiple conditions is made
    - ► may fail to detect defects caused by combinations of conditions
- Example defects
  - wrong action taken; incorrect output

2.16

Exercise

Statement and Decision Testing

2.17

Contents

2.18

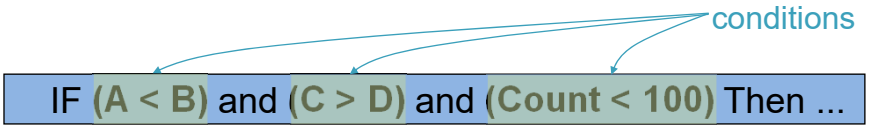
2.4 Modified Condition/Decision Coverage (MC/DC)

- tests how a decision is made when it has >1 condition
  - e.g. the 'decision predicate'
    IF ((age > 17) AND (testMark < 65)) OR (empFlag = No) ...
  - has 3 'atomic conditions'
  - Decision testing only verifies overall outcome
  - for safety-critical code also need to verify that
    - ► each atomic condition works correctly for both True and False
    - ► each atomic condition independently has correct effect on overall outcome
- we'll look first at the weaker techniques of Condition testing and Decision Condition testing
  - both removed from syllabus in 2019
  - but helpful steps on way from Decision testing to MC/DC

2.19

2.4 Towards MC/DC: Condition testing

- all 'atomic' conditions within decision statements must take both true and false values



- 2 test cases required:
  - conditions T, F, T (FALSE outcome)
  - conditions F, T, F (FALSE outcome)

Oops! We've not achieved decision coverage (needs both a TRUE and a FALSE overall outcomes).

- % of condition outcomes exercised

$$= \frac{\text{number of condition outcomes executed}}{\text{total number of condition outcomes}} \times 100$$

$$= \frac{6}{6} = 100\%$$

2.20

2.4 Towards MC/DC: Condition vs. Decision coverage

- for the previous example:
  - condition coverage could be achieved by:
    - ►conditions T, T, T (TRUE outcome)
    - ►conditions F, F, F (FALSE outcome)
  - which does also give us 100% decision coverage
- condition coverage does not guarantee decision coverage
- for decisions comprising only one simple condition
  - e.g. A > B
  - condition coverage = decision coverage
- for decisions comprising two or more conditions
  - e.g. ((A > B) and (C < D)) or (E = 9)
  - condition coverage ≠ decision coverage

2.21

Exercise

Condition Testing

2.22

2.4 Towards MC/DC: Decision Condition testing

- builds on decision testing by combining it with condition testing
  - exercise all decision outcomes, plus
  - exercise all 'atomic' condition outcomes

IF **(A < B)** and **(C > D)** and **(Count < 100)** Then ...

- 2 test cases required:
  - atomic conditions: T, T, T (TRUE decision outcome)
  - atomic conditions: F, F, F (FALSE decision outcome)
- % of all decision and condition outcomes exercised

$$= \frac{\text{number of condition and decision outcomes executed}}{\text{total number of condition and decision outcomes}} \times 100$$

$$= \frac{8}{8} = 100\%$$

2.23

2.4 Towards MC/DC: Decision Condition testing e.g.

```
if (Exam >= 70 and Practical >= 70 and Exam + Practical >= 160) then
    print "Distinction"
else if (Exam >= 60 and Practical >= 60 and Exam + Practical >= 140) then
    print "Merit"
end if
```

	E>=70	P>=70	E+P>= 160	Out-come
1	T	T	T	T
2	F	F	F	F
3	T	F	F	F

E>=60	P>=60	E+P>= 140	Out-come
F	F	F	F
T	T	T	T

Test cases	Exam	Practical	Expected Outcome
1	80	80	Distinction
2	40	40	nothing
3	80	60	Merit

2.24

2.4 Towards MC/DC: coupling

- coupled conditions
  - e.g. IF (a < 10) or ((b > c) and (a > 100)) THEN
  - in this example, 'a' appears in 2 separate conditions
  - both conditions cannot be TRUE at the same time
  - so condition combinations T, T, T and T, F, T are not possible
    - ► eliminate impossible combinations

  - e.g. IF (Copies > 0 and Copies < 25) THEN
  - in this example, 'Copies' appears in both conditions
  - if 'Copies > 0' is FALSE, 'Copies < 25' must be TRUE
  - if 'Copies < 25' is FALSE, 'Copies > 0' must be TRUE
  - so condition combination F, F is not possible

2.25

Exercise

Decision Condition Testing

2.26

2.4 MC/DC: Modified Condition Decision Coverage

- builds on Decision Condition testing
  - every condition shown to independently affect decision outcome

IF **(A < B)** and **(C > D)** and **(Count < 100)** THEN

- minimum number of test cases required = n + 1
  - where n is the number of conditions
  - TTT, TTF, TFT, FTT
- percentage covered

Beware of coupling – this might increase the number needed

$$= \frac{\text{number of conditions shown to independently affect the decision outcome}}{\text{total number of conditions that independently affect the decision outcome}} \times 100$$

2.27

2.4 MC/DC example 1

(A > B) and (C < D)

- step 1: draw the truth table
- step 2: eliminate impossible combinations (none here)

	A>B	C<D	Outcome
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

- step 3: select pairs of condition combinations for each condition that shows its effect on the outcome
  - ►rows 1 and 3 show effect of changing first condition
  - ►rows 1 and 2 show effect of changing second condition
  - ►start with both True for 'and'
  - ►start with both False for 'or'
- step 4: specify a test case for each chosen combination

	A	B	C	D	Expected
1	6	3	2	4	?
2	6	3	4	2	?
3	3	6	2	4	?

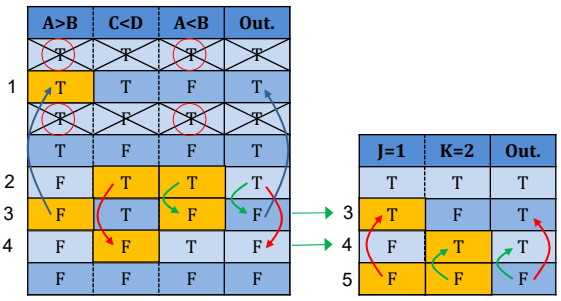
Test cases

2.28

2.4 MC/DC example 2

if (A > B) or ((C < D) and (A<B))
then doThis
else if (J=1 or K=2)
then doThat

- step 1: draw the truth tables
- step 2: eliminate impossible combinations (2 here: 1st and 3rd conditions cannot be True at same time)
- step 3: identify pairs of combinations that show effect of changing each condition
- step 4: specify a test case for each chosen combination in both tables, accounting for 'fall through' combinations.



	A	B	C	D	J	K	Expected
1	6	3	2	4			doThis
2	3	6	2	4			doThis
3	1	1	2	4	1	2	doThat
4	3	6	4	2	1	3	doThat
5	1	1	4	2	2	3	nothing

Test cases

2.29

2.4 MC/DC: short-circuiting

- compiler short-circuiting
  - done by some compilers to optimise code by avoiding unnecessary evaluations
    - ► e.g. IF (A < 5 or B > 5) THEN
    - ► if 'A < 5' is TRUE then decision outcome will always be TRUE regardless of value of 'B > 5'
    - ► there is no difference between condition combinations T, F and T, T
    - ► so some compilers don't bother to evaluate 'B > 5'
  - reduces the number of white-box tests that can be run

2.30

2.4 MC/DC: considerations

- Applicability
  - should be used when dealing with safety critical software where any failure may cause a catastrophe
- Limitations / Difficulties
  - coupling can add complexity
    - ► can specify that only uncoupled atomic conditions must be tested
  - short-circuiting may affect ability to attain MC/DC coverage since some required tests may not be achievable

2.31

Exercise

Modified Condition/Decision Testing

2.32

Contents

2.33

2.5 Multiple Condition testing

- all combinations of conditions taking both True and False values in every decision statement, e.g.

```
IF (A < B) and (C > D) and (Count < 100) THEN
```

- minimum number of test cases required = 2^n
  - where n is the number of conditions
- 8 test cases required:
  - TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF
- % of multiple condition combinations

$$= \frac{\text{number of condition combinations executed}}{\text{total number of condition combinations}} \times 100$$

2.34

2.5 Compound decision statements

- multiple interdependent decision statements
  - e.g.

```
IF (Level = "Gold" or Days > 9) THEN
    Do this
ELSEIF (Level = "Silver" and Points > 50,000) THEN
    Do that
ELSE
    Do something else
ENDIF
```

  - only if 1st decision outcome is FALSE will the 2nd decision be executed
  - must design test cases carefully:

Test Case	Level	Days	Points	Expected
1	Gold	2	4	?
2	Silver	2	50,001	?
3	Silver	2	50,000	?

2.35

2.5 Multiple Condition testing example 1

(A > B) and (C = 2)

- step 1: draw the truth table
- step 2: eliminate impossible combinations (none here)
- step 3: specify a test case for each (remaining) combination (4 in this case)

A>B	C=2	Outcome
T	T	T
T	F	F
F	T	F
F	F	F

A	B	C	Expected
6	3	2	?
6	3	4	?
3	6	2	?
3	6	4	?

2.36

2.5 Multiple Condition testing example 2

(A > B) or ((C < D) and (A<B))

- step 1: draw the truth table
- step 2: eliminate impossible combinations (2 here: 1st and 3rd conditions cannot be True at the same time)
- step 3: specify a test case for each (remaining) combination (6 in this case)

A>B	C<D	A<B	Out.
~~T~~	~~T~~	~~T~~	~~T~~
T	T	F	T
~~T~~	~~F~~	~~T~~	~~T~~
T	F	F	T
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

A	B	C	D	Expected
6	3	2	4	?
6	3	4	2	?
3	6	2	4	?
1	1	2	4	?
3	6	4	2	?
1	1	4	2	?

2.37

2.5 Multiple Condition testing example 3

if (A > B) or ((C < D) and (A<B))
then doThis
else if (J=1 or K=2)
then doThat

- step 1: draw the truth tables
- step 2: eliminate impossible combinations
- step 3: specify a test case for each (remaining) combination in both tables, accounting for 'fall-through' combinations

test cases that give False outcome for the 1st decision statement go on (fall-through) to the 2nd decision: test cases 4, 5 and 6

	A>B	C<D	A<B	Out.
	~~T~~	~~T~~	~~T~~	~~T~~
1	T	T	F	T
	~~T~~	~~F~~	~~T~~	~~T~~
2	T	F	F	T
3	F	T	T	T
4	F	T	F	F
5	F	F	T	F
6	F	F	F	F

	J=1	K=2	Out.
4	T	T	T
5	T	F	T
6	F	T	T
7	F	F	F

	A	B	C	D	J	K	Expected
1	6	3	2	4			doThis
2	6	3	4	2			doThis
3	3	6	2	4			doThis
4	1	1	2	4	1	2	doThat
5	3	6	4	2	1	3	doThat
6	1	1	4	2	2	2	doThat
7	1	1	4	2	2	3	nothing

Test cases

2.38

2.5 Multiple Condition testing: considerations

- multiple condition coverage guarantees MC/DC
- Applicability
  - embedded software with very high reliability requirements
    - ► syllabus e.g.: telephone switches that are expected to last 30 years
- Limitations
  - needs many tests (2^n)
  - MC/DC more feasible in most situations
  - coupling reduces number of feasible combinations
  - short-circuiting may reduce the number needed
    - ► if know how compiler does it

2.39

Exercise

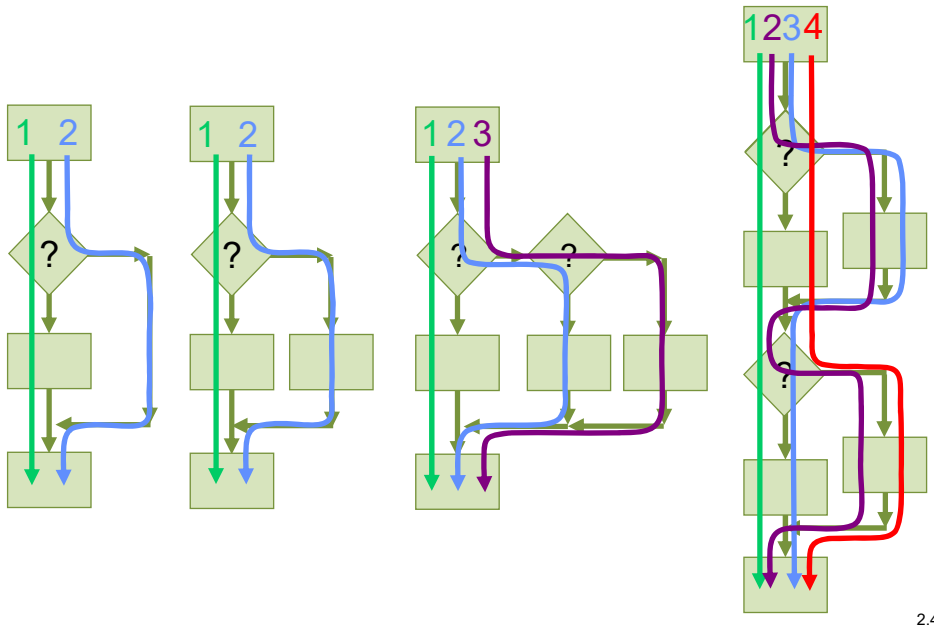
Multiple Condition Testing

2.40

Contents

2.41

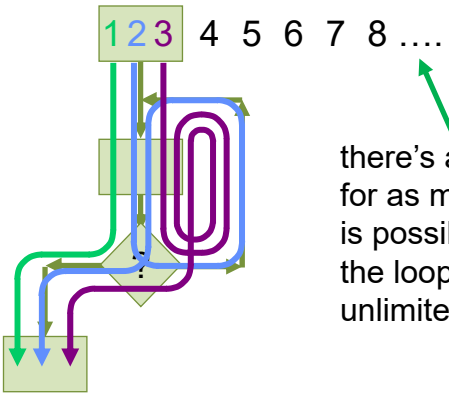
2.6 Basis path testing



2.42

2.6 Basis path testing with loops

when the code has loops:



2.43

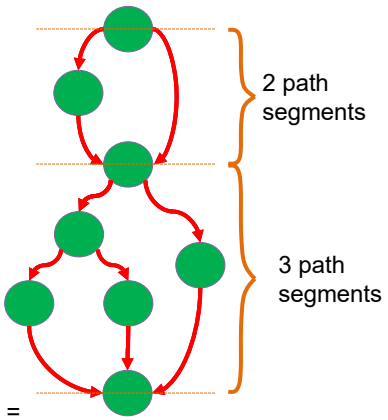
2.6 Basis path coverage

- percentage of paths exercised by a test suite

$$= \frac{\text{number of paths exercised}}{\text{total number of paths}} \times 100$$

if not a small simple piece of code, and particularly when it contains loops, this number will soon become too large for complete testing

paths:
T, T, T
T, T, F
T, F
F, T, T
F, T, F
F, F



Total number of paths =
no. of paths to 1st common node * no. of paths to 2nd node * no. of paths to 3rd node ...
here = 2 x 3 = 6

2.44

2.6 Basis path testing: the Simplified Baseline Method

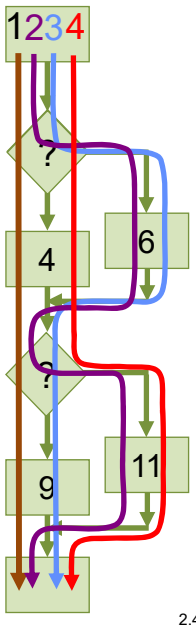
1. **Create a control graph for the 'specification item' being considered**
2. **Choose a 'baseline' path**
   - the most important path to test: use judgement / consult user – risk-based approach?
3. **Generate a second path**
   - change outcome of 1st decision on baseline path
   - identify rest of new path by ensuring that its max. no. of decision outcomes is same as for baseline path
4. **Generate the third path**
   - start with baseline path and change outcome of its 2nd decision
   - if a multiway decision is encountered, e.g. a case statement, each of its outcomes should be exercised before moving on to the next decision
5. **Generate further paths**
   - continue to change decision outcomes on the baseline path until all have been changed
   - as each new decision is considered, choose its most important outcome first
6. **Generate remaining paths**
   - when all decision outcomes on baseline path have been covered, apply same approach to the 2nd path, then to the 3rd ... until all decision outcomes in this specification item have been exercised

2.45

2.6 Basis path testing: example

```
1.  Read numInStock
2.  Read numOrdered
3.  IF (numOrdered <= numInStock)
4.        Print "In stock"
5.  ELSE
6.        Print "No stock"
7.  ENDIF
8.  IF (numOrdered > 10)
9.        Print "Discount applies"
10. ELSE
11.       Print "No discount"
12. ENDIF
```

1. **Baseline path**
2. **change 1st baseline decision**
3. **change 2nd baseline decision**
4. **change 1st decision on 2nd path**



2.46

2.6 Basis Path testing: coverage

- Coverage
  - McCabe's Simplified Baseline Method should ensure 100% coverage of 'linearly independent paths'
    - ► unique paths that don't loop

$$= \frac{\text{number of paths exercised}}{\text{total number of paths ignoring multiple paths around loops}} \times 100$$

    - ► count each loop structure once only
  - quantity of these paths should match McCabe's cyclomatic complexity for the code (see later)
  - guarantees 100% decision coverage
    - ► and therefore statement coverage
  - syllabus says more thorough than Decision testing with only relatively small increase on quantity of tests

2.47

2.6 Basis Path testing: considerations

- Applicability
  - safety critical software
  - good complement to the techniques covered earlier because focus is on paths rather than just decision statements
    - ► probably why syllabus says it's more thorough
- Limitations
  - for complex code need specialized tools
  - so often limited to systems whose safety or mission criticality justifies the expense

2.48

Exercise

Basis Path Testing

2.49

Contents

2.50

2.7 API testing

- Application Programming Interface
  - published specification of an application interface that programs use to communicate with each other
    - ► equivalent of graphical user interface for software
    - ► others can build their own applications that interact with ours
- e.g. Adobe Acrobat® provides API function:
  - PDDocInsertPages (doc1, afterPage, doc2, startPage, numPages, flags)
    - ► **doc1** – reference to document into which pages are inserted
    - ► **afterPage** – page number after which pages are inserted
    - ► **doc2** – reference to document containing pages to be inserted
    - ► **startPage** – page number of first page to be inserted
    - ► **numPages** – number of pages to be inserted
    - ► **flags** – flags that indicate the information to be inserted

2.51

2.7 API testing: test design

- not really a test technique
  - type of software that needs a special approach to testing
- focus on input values and evaluation of response
  - parameters passed in/out and returned values
  - negative testing
    - ► e.g. page numbers: negative, too large and non-integer
      - – afterPage, startPage, numPages
  - combinatorial testing
    - ► e.g. single and multi-page inserts into single and multi-page documents each with different combinations of flags
      - – doc1, doc2, numPages
  - performance testing
    - ► interface is 'loosely coupled' (different processor at each end
    - ► possibility of timing problems or lost transactions

2.52

2.7 API testing: considerations

- Applicability – increasingly common for
  - operating systems calls
  - service-oriented architectures (SOA, e.g. SOAP and REST)
  - remote procedure calls (RPC)
  - web services
  - distributed apps, especially those using software containerization
- Limitations
  - testing an API directly usually needs specialized tools
  - no real GUI so must create test harness
- Coverage
  - no specific measure
  - minimum: all calls to the API, all valid parameter values + sample of invalid parameter values (EP?)
- Types of defect
  - interface & data handling issues, timing problems, lost / duplicate transactions

2.53

Contents

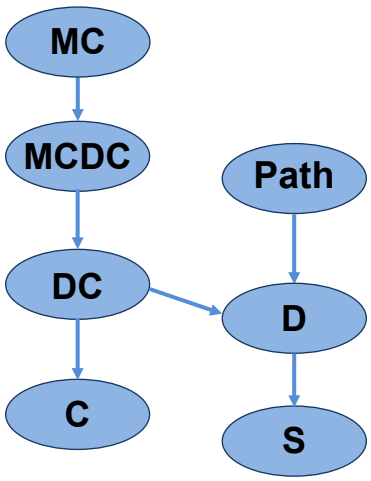
2.54

2.8 Selecting a white-box test technique

- context dependent
  - risk and criticality ➔ thoroughness needed ➔ 'coverage metric'
  - coverage metric = statement / decision / MC/DC coverage etc.
  - so this determines the technique needed if it has a metric
  - API testing as needed according to risk and criticality
- standards or regulations may dictate their use
  - IEC61508 has a scale of Safety Integrity Levels (SIL)
  - used in various industry sectors (automotive, rail, nuclear etc.)
  - gives recommendations for test coverage at each level
    - ► see Study Guide
- coverage expected to be 100% for critical systems
  - less is meaningless – the 5% not covered might be the most vital
  - could be less for non-critical systems if given baseline minimum
- best done after functional and non-functional testing
  - tools measure coverage those tests give, then add tests for what's missing

2.55

2.8 Partial ordering of white-box techniques



Adapted from BS 7925-2

Key:

MC = Multiple Condition Coverage

MCDC = Modified Condition/Decision Coverage

DC = Decision Condition Coverage

C = Condition Coverage

Path = Path Coverage

D = Decision Coverage

S = Statement Coverage

2.56

2.8 Example standards

- Airborne systems (Standard DO-178B)

Integrity Level	Testing Requirement
A. Catastrophic: failure may cause lack of critical function needed to safely fly or land the plane.	MC/DC
B. Hazardous: failure may have a large negative impact on safety or performance.	Decision Coverage
C. Major: failure is significant, but less serious than A or B.	Statement Coverage
D. Minor: failure is noticeable, but with less impact than C.	No requirement for structural testing.
E. No effect: failure has no impact on safety.	

- Safety related systems (IEC-61508)

Safety Integrity Level (SIL)	SIL Description	Testing Requirement
4 (most critical)	Varies depending on which standard is used.	**MC/DC** highly recommended.
3		**Statement** & **decision** coverage highly recommended
2		**Statement** coverage highly recommended, **decision** coverage recommended
1 (least critical)		**Statement** and **decision** coverage recommended

2.57

2.8 Code structure

Technique	Code Structure Required	Applicability according to ISTQB
Statement Testing	-	Always applicable
Decision Testing	Code must contain at least one decision statement.	Should be considered when the code being tested is important or even critical
Condition Testing	Code must contain at least one decision statement with a Boolean expression that has two or more conditions.	Not given (probably because it does not guarantee decision coverage)
Decision Condition Testing		Not given (probably because MC/DC is stronger for little extra cost)
Modified Condition Decision Testing		Should be used for safety-related and other high integrity software
Multiple Condition Testing		As for MC/DC and may be used to test embedded software with a high reliability requirement
Path Testing	Code must contain at least two decision statements in series (i.e. not nested).	Safety critical systems

2.58

Exercise

Technique Selection

2.59

Summary: key points

- structure-based testing:
  - targets structural items, e.g. executable statements, decision outcomes, paths
  - percentage covered and technique strength indicate strength (thoroughness)
- measurable techniques covered:
  - statement and decision testing
  - modified condition/decision (via condition & decision condition)
  - multiple condition, basis path
- other structural techniques:
  - API testing
- selecting techniques
  - risk, criticality, standards/regulations may dictate or guide
  - complements functional & non-functional testing

2.60

Session 2

White-box Test Techniques

From the ISTQB Glossary

code coverage: The coverage of code.

coverage: The degree to which specified coverage items have been determined or have been exercised by a test suite expressed as a percentage.

coverage item: An attribute or combination of attributes that is derived from one or more test conditions by using a test technique that enables the measurement of the thoroughness of the test execution.

structural testing: See white-box testing.

structure-based testing: See white-box testing.

white-box testing: Testing based on an analysis of the internal structure of the component or system.

2.1 Introduction

White-box test techniques are also known as structural, structure-based or glass-box techniques (before 2019 the ISTQB preferred to call them structure-based). The nickname 'glass-box' is particularly apposite because it implies that we can see through the 'walls' of the system into the details of its construction. These techniques apply to code and other elements of structure, e.g. module call trees, menus, business process flow charts etc. They all use some aspect of the system's architecture and/or flow as the basis for test design.

All white-box techniques help us to derive tests systematically by focusing our attention on some specific type of structural item. Each technique uses a different structural item. For example, statement testing focuses our attention on executable statements within the code whereas path testing focuses our attention on executable paths through the code.

Coverage items

The thoroughness of the testing performed can be measured objectively by counting the number of structural items exercised or 'covered' and comparing this with the total number present. For this reason such structural items are also known as 'coverage items'. Covering an item more than once does not increase the count. Structural coverage only considers whether or not a structural item has been covered or not; it makes no difference if an item has been covered only once, twice or many times. Coverage measures can be used as criteria for test completion. However, achieving 100% coverage with any white-box technique does not mean that testing is complete; rather, it indicates that the technique no longer suggests any useful tests. It may then be time to consider a different technique.

Thoroughness and strength

Different white-box techniques provide different levels of thoroughness of testing. This may also be referred to as 'strength': the more thorough the technique is, the 'stronger' it is said to be. The more thorough the testing, the more effort is required to achieve it, so we generally have to make a trade-off between the thoroughness of the testing that we wish to achieve and the amount of effort we can afford to spend on achieving that thoroughness.

The syllabus covers the following white-box techniques:

- Statement Testing
- Decision Testing
- Modified Condition/Decision Coverage (MC/DC) Testing
- Multiple Condition Testing
- Basis Path Testing
- API Testing

The first four of these are listed in order of their relative strength, starting with the weakest (statement testing) and ending with the strongest (multiple condition testing). These are all based on decision predicates and will find broadly the same types of defect. No matter how complex a decision predicate may be, it will always evaluate to either TRUE or FALSE and this will determine the path subsequently taken through the code.

Each of these four techniques is aimed at a different type of structural item and so has its own "coverage metric" (way of measuring coverage). The same is true of basis path testing. described in the following sections has its own way of doing this.

The strength of basis path testing cannot be compared directly with the previous four techniques because focuses on a different kind of structural element. API testing is different again, as it is a description of a type of testing rather than a systematic technique and does not have a specific coverage measure associated with it.

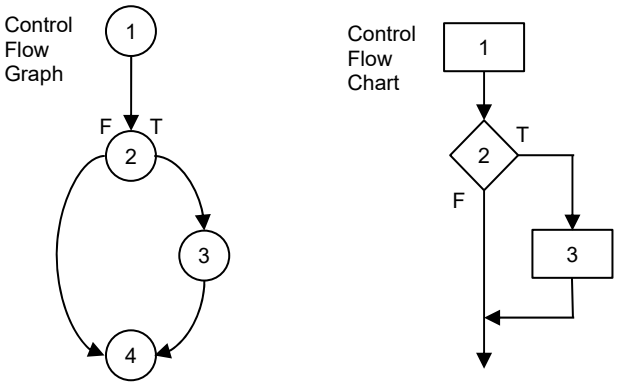
The white-box techniques listed above are amongst those most commonly used. There are others. Although not mentioned in the syllabus we will describe two others, Condition testing and Decision Condition testing, because they are steps on the way from Decision testing to MC/DC and the easiest way to learn about MC/DC is to learn these first.

Control flow diagrams

A control flow diagram is a pictorial representation of the possible flow of control through source code, i.e. it shows the paths that execution could follow. It does not show what the code is doing but does give us a view on how it is structured. We can analyse the structure and look for defects and anomalies. There are two common ways to draw a control flow diagram: control flow charts and control flow graphs. Control flow graphs use only circles and these are called nodes.

A simple piece of code and its corresponding control flow graph and control flow chart are shown below.

1. Read balance
2. IF (balance < 0) THEN
3. Print "Account overdrawn"
4. ENDIF



Control flow charts use boxes and straight lines. Each statement is represented by a box. An unconditional statement is represented by a rectangular box. A conditional statement, that is, a point at which a decision is made and the logic then flows along either one path or another, is drawn as a diamond. The logic flows between boxes are shown by lines or arrows that use 90-degree angles if they can't go straight to their end point. There can be only one line out of an unconditional statement, but there must be two out of every conditional one.

Control flow graphs use circles and curves arrows. A decision statement is represented by a single node (circle) and can be distinguished from unconditional nodes because it has more than one arrow coming out of it. We have labelled the arrows coming out of the decision nodes in both control flow diagram above with T and F to denote True and False. This way we will know which arrow is taken for each decision outcome.

Of these two ways, the control flow chart is more abstract, the diagram more physical. The control flow chart, above right, does not have a box for the ENDIF because it's not an executable statement (see 2.2 below); it merely marks the point where two paths come together after diverging at a decision point and doesn't actually make anything happen. The control flow graph, above left, has a node for the ENDIF statement because it is logically significant even though nothing actually happens at that point.

In both types of diagram, consecutive unconditional statements may be shown in a single node or box. It makes the diagram quicker to draw and easier to read. This is called a linear code sequence and is the basis of the Linear Code Sequence and Jump (LCSAJ) test technique which is no longer in scope of this syllabus (removed in 2122). A group of 2 or more statements must always be executed together in the same order so there's only one way for the logic to flow through them.

Although it won't always be followed in this Study Guide, there's a convention that the True path goes to the right of the decision node or box, and the False path goes to the left of the node (in a control flow graph) or down from the diamond box (in a control flow chart). That's because, usually, more things happen on the True path than on the false path; so, starting at the top left of the paper or screen, doing it this way makes better use of the available space and avoids long thin diagrams.

WARNING: Now that we have described the formal notation for the control flow diagrams, it is important to realise that these notations are not always strictly applied. You may see many variations. Typically, though, it is fairly easy to recognise the main features (nodes, branches / edges / path segments) and interpret them correctly.

Note: LOs 2.2.1, 2.3.1, 2.4.1, 2.5.1, and 2.6.1, below, refer to a '**specification item'**. This includes items such as sections of code, requirements, user stories, use cases and functional specifications.

2.2 Statement Testing

Learning Objective

TTA 2.2.1 K3 Write test cases for a given specification item by applying the Statement test technique to achieve a defined level of coverage

From the ISTQB Glossary

executable statement: A statement which, when compiled, is translated into object code, and which will be executed procedurally when the program is running and may perform an action on data.

statement: An entity in a programming language, which is typically the smallest indivisible unit of execution.

statement coverage: The coverage of executable statements.

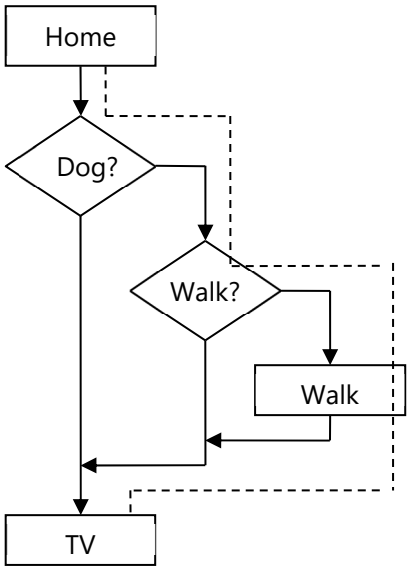
statement testing: A white box test design technique in which test cases are designed to execute statements.

Statement coverage is the percentage of executable statements that are exercised by a test or test suite. This is calculated as:

$$Statement\ Coverage = \frac{Number\_of\_statements\_exercised}{Total\_number\_of\_statements}\ x\ 100\%$$

A control flow diagram allows the structure of code to be visualised so that we can see the logic flows and map our coverage. For example, here is some pseudo-code and, to its right, the corresponding control flow chart:

```
Arrive home
IF I have a dog
        IF the dog needs a walk
                Take dog for a walk
        END IF
END IF
Watch television
```



In all cases I'll eventually watch TV, but if I have a dog and s/he needs a walk I'll do that first. If I don't have a dog then the 'walk?' decision doesn't have to be made, so I can go immediately to watch TV without checking.

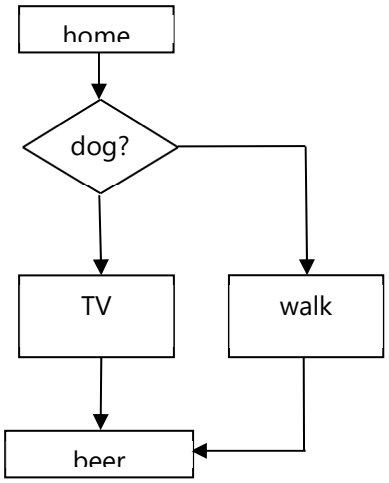
Note the definition of executable statement, above. Comments, sub-routine names, variable declarations and some other types of statement are not executable because they don't get compiled into executable object code. So we haven't drawn a box for either 'END IF', because they only bring two logic paths back together and nothing actually happens at that point. ENDIF, ENDFOR, ENDWHILE and other such statements in source code are not executable and so don't count.

For 100% statement coverage of the pseudo-code above, we need only one test case if we choose the right one. The dashed line follows the flow of this test case: I have a dog and s/he needs a walk so I do that before watching TV, which covers all the statements in the code. It has taken only one of the possible ways out of each decision, but both of the decision statements have been executed, and for statement coverage that's all we need.

The case in which I have a dog and s/he does not need a walk goes through 4 / 5 statements = 80% coverage. If I don't have a dog, that covers only 3 / 5 statements = 60% coverage.

In the next example, we won't get 100% coverage with any single test case because two logic paths are mutually exclusive.

```
Arrive home
IF I have a dog
        Take dog for walk
ELSE
        Watch television
END IF
Go to bed
```



Either of the two possible test cases gives us 4 / 5 statements = 80% coverage and we need both to reach 100%.

Typical ad hoc testing achieves somewhere in the range 60% - 75% statement coverage.

Applicability

This level of coverage should be considered as a minimum for all code being tested.

Limitations/Difficulties

Decisions are not considered. Therefore, even high percentages of statement coverage may not detect certain defects in the code's logic.

2.3 Decision Testing

Learning Objective

TTA 2.3.1 K3 Write test cases for a given specification item by applying the Decision test technique to achieve a defined level of coverage

From the ISTQB Glossary

decision: A type of statement in which a choice between two or more possible outcomes controls which set of actions will result.

decision coverage: The coverage of decision outcomes.

decision outcome: The result of a decision that determines the next statement to be executed.

decision testing: A white box test design technique in which test cases are designed to execute decision outcomes.

In the control flow diagrams that were used as examples in 2.2 above, each decision statement's diamond has two ways out. These represent the two possible outcomes that any decision can have. Decision coverage is about testing all of these possible outcomes.

$$Decision\ Coverage = \frac{Number\_of\_decision\_outcomes\_exercised}{Total\_number\_of\_decision\_outcomes}\ x\ 100\%$$

Unconditional statements are not counted for this. Consider this code fragment:

```
If A > B
        set C = 0
Endif
```

To achieve 100% statement coverage of this code, only one test case is required in which variable A contains a value that is greater than the value of B. However, decision coverage requires each decision to have had both a true and a false outcome. Therefore, to achieve 100% decision coverage, a second test case is necessary in which variable A contains a value that is not greater than the value of B, so that the decision has a false outcome.

In the first example given above for statement testing, we need three test cases for 100% decision coverage. The one that gives 100% statement coverage must go through the Eat statement, so cannot be false for partner home or dinner ready; in fact, it exercises only 2 out of the 4 possible decision outcomes so gives only 50% decision coverage. A test case that is false for dinner ready cannot also be false for partner home, because if partner not home then dinner can't be ready; together, this and the first test case exercise 3 out of 4 outcomes = 75% coverage.

100% decision coverage guarantees 100% statement coverage: if you have taken all possible outcome paths from all decisions then you must also have been through all statements. The reverse is not necessarily true, as we saw above. Sometimes, however, as in the second example given above for statement testing, the same test cases will give 100% coverage for both. 100% statement coverage may give 100% decision coverage but does not guarantee it. We therefore say that decision coverage is "stronger" than statement coverage, since it may require more test cases to achieve the same degree of thoroughness.

Typical ad hoc testing achieves somewhere is the range 40% - 60% decision coverage.

Applicability

This level of coverage should be considered when the code being tested is important or even critical (see the table in Section 2.8).

Limitations/Difficulties

Because it may require more test cases than testing only at the statement level, it may be problematic when time is an issue. Decision testing does not consider the details of how a decision with multiple conditions is made and so may fail to detect defects caused by combinations of these conditions.

2.4 Modified Condition/Decision Coverage (MC/DC) Testing

Learning Objective

TTA 2.4.1	K3	Write test cases for a given specification item by applying the Modified Condition/Decision Coverage (MC/DC) test design technique to achieve a defined level of coverage

From the ISTQB Glossary

compound condition: Two or more single conditions joined by means of a logical operator.

condition: A logical expression that can be evaluated as True or False.

condition coverage: The coverage of condition outcomes that have been exercised by a test suite.

condition testing: A white-box test technique in which test cases are designed to execute outcomes of atomic conditions.

decision condition coverage: The percentage of all condition outcomes and decision outcomes that have been exercised by a test suite. 100% decision condition coverage implies both 100% condition coverage and 100% decision coverage.

decision condition testing: A white-box test technique in which test cases are designed to execute condition outcomes and decision outcomes.

modified condition / decision coverage (MC/DC): The coverage of all single condition outcomes that independently affect a decision outcome that have been exercised by a test suite.

modified decision / condition testing (MC/DC): A white-box test technique in which test cases are designed to exercise single condition outcomes that independently affect a decision outcome.

2.4.1 Decisions with Multiple Conditions

Compared to Decision testing, which considers the entire decision as a whole and evaluates the TRUE and FALSE outcomes in separate test cases, MC/DC testing considers how a decision is made when it includes multiple conditions. This technique is very thorough and so is often used for safety-critical systems.

In order to understand how it works it is useful to consider first the simpler techniques of Condition testing and Decision Condition testing, as they are steps along the path from Decision testing to MC/DC.

Notation

First, let's look at the syntax of the decision statement. An example with three conditions could be:

if **(a > b)** or **(c > d)** or **(e > f)** ...

A more typical example, with four conditions, could be:

if **(count > 100** and **lastTry = "yes")** or **(status = "complete"** and **abort = "yes")** ...

Decision statements can become much longer and more complex even than this. For the purposes of structural testing we do not always need to deal with all the detail of the variable names and comparison operators. What is important are the individual (atomic) conditions and for these it is simpler to substitute the letters A, B, C etc for the individual conditions. Similarly, we don't need the 'if' or 'then' words. So, we can use a simplified notation:

if continue = "yes" or (index < maximum and refNum > 0)

becomes only **A or (B and C)**

This is sufficient for our purposes and makes it easier to 'see' what is important. This is optional of course. If you prefer to use the all the detail then please carry on – use whatever works best for you. We will use the simplified notation where it seems convenient to do so.

One thing we will have to be careful of with this simple notation is coupling. This is when two or more conditions are related in some way. We will need to be sure that the relationship is obvious from the notation that we use. We'll look at an example of coupling later.

Complex decision statements

Where there are two or more conditions within a decision statement there are extra considerations that we may need to take into account. Two of these are coupling and short-

circuiting. These issues can affect the choice of test cases and the number of test cases required to achieve 100% coverage of both modified decision condition coverage and multiple condition coverage.

Another consideration is compound decision statements: that is, where one decision statement is dependent on the outcome of a previous decision statement. In these situations our test cases have to achieve a particular outcome for the first decision statement before we can start to exercise the second one.

All of these issues are discussed in separate sections below.

Coupling

When using some structural testing techniques we need to be wary of coupling. This is where the same variable appears more than once across a set of conditions within the same decision statement. For an example, consider the following Boolean expression:

distance = "Longhaul" and (**fareType = "Business"** or **fareType = "First Class"**)

The variable **fareType** occurs in two of the three conditions so these two conditions are not independent, they are 'coupled'. Because of this, not all combinations of condition values will be possible - impossible combinations must be eliminated. In this example, the latter two conditions cannot both be True at the same time.

Using our simple notation the Boolean expression above becomes: A and (B or C). However, we need to highlight the coupling of the conditions and this can be done like this: A and (B1 or B2). This is used to indicate that B1 and B2 are coupled, although not the same.

Short circuiting

Short circuiting is undertaken by some compilers to make the execution of the code more efficient. The idea is to avoid unnecessary evaluations where there are multiple conditions within a decision statement. Consider the Boolean expression **A and B**. If the first condition evaluates to False the outcome of the whole expression will be False, regardless of the value of the second condition. Compilers that short circuit will not evaluate the second condition in such a situation.

Short circuiting may affect our ability to achieve 100% coverage in any of the techniques that look at individual conditions. In order to take it into account we need to know the order in which the compiler will evaluate the conditions. It could be from left to right, right to left, or some other scheme. There is no standard. It seems likely that any exam questions that require short circuiting to be applied will make the order evaluation clear ☺

Compound decision statements

Consider this compound decision statement:

```
if (A and B)
        do this
else
        if (C or D)
                do that
endif
```

This is actually a series of two decision statements in which the second decision statement will only be executed if the outcome of the first decision statement is False. So we may need more test cases designed specifically to exercise the condition value combinations of the second decision statement.

Below we will see examples of this and how it affects our use of some of the test techniques.

2.4.2 Condition Testing

Condition testing takes things further than Decision testing by considering the individual 'atomic' conditions within each decision statement. It seeks to exercise both condition outcomes (i.e. true and false) for each atomic condition within the decision statement.

For a simple decision statement that has just one condition, condition testing is the same as decision testing, However, for complex decision statements that have more than one condition, condition testing may require more test cases than decision testing to achieve the same measure of coverage.

Condition coverage is the number of condition outcomes exercised divided by the total number of condition outcomes. This is calculated by:

$$Condition\ Coverage = \frac{Number\_of\_condition\_outcomes\_executed}{Total\_number\_of\_condition\_outcomes}\ x\ 100\%$$

Remember that the relationship between condition coverage and decision coverage depends on the nature of the decisions within the code being measured. If all decision statements comprise a single condition, then condition coverage is equivalent to decision coverage. However, if any decision statement comprises two or more conditions then the two coverage measures will be different. Consider the following code fragment.

```
if a > b and c > d and e > f then
    print "Decision is true"
end if
```

The decision statement contains three conditions (a > b, c > d and e > f). Condition coverage requires each condition to have had both a true and false outcome. This can be achieved with two test cases shown in the table below.

Test Case	a > b	c > d	e > f	Decision Outcome
1	T	F	T	F
2	F	T	F	F

2.4.3 Decision Condition Testing

The two test cases that we used for Condition testing to give 100% condition coverage do not, unfortunately, achieve 100% decision coverage. If we had chosen different ones then we might have achieved both but it isn't guaranteed. For this reason, Condition testing is not commonly used. Instead, a stronger technique called Decision Condition testing can be used.

Decision Condition testing, as its name suggests, is a combination of the two techniques Decision testing and Condition testing. For 100% decision condition coverage a set of test cases is required that achieves both 100% decision coverage and 100% condition coverage. For the code fragment:

```
if a > b and c > d and e > f then
    print "Decision is true"
end if
```

decision condition coverage can be achieved with these two test cases:

Test Case	a > b	c > d	e > f	Decision Outcome
1	T	T	T	T
2	F	F	F	F

These two give us both 100% condition coverage and 100% decision coverage and therefore 100% decision condition coverage. Whenever we achieve 100% decision condition coverage we are guaranteed to have achieved 100% condition coverage and 100% decision coverage.

$$Decision\ Condition\ Coverage = \frac{Number\_of\_decision\_\&\_condition\_values\_executed}{Total\_number\_of\_decision\_\&\_condition\_values}\ x\ 100\%$$

Let's now look at a different example.

Example with coupling and a compound decision

The following code fragment has a compound decision statement (the second decision is only executed if the first decision outcome is False). The conditions in the second decision statement are coupled with those in the first decision statement (e.g. if Exam >= 70 is True, then Exam >= 60 will also be True).

```
IF Exam >= 70 and Practical >= 70 and Exam + Practical >= 160 THEN
        Print "Distinction"
ELSE IF Exam >= 60 and Practical >= 60 and Exam + Practical >= 140 THEN
        Print "Merit"
END IF
```

We can cover all the condition outcomes and all the decision outcomes with just three, carefully chosen, combinations:

Id	Exam >= 70	Practical >= 70	Exam + Practical >= 160	Decision Outcome	Exam >= 60	Practical >= 60	Exam + Practical >= 140	Decision Outcome
1	T	T	T	T	-	-	-	-
2	F	F	F	F	F	F	F	F
3	T	F	F	F	T	T	T	T

The final step is to design the test cases that will exercise these combinations:

Test Case	Exam	Practical	Expected Outcome	Id
1	80	80	Distinction	1
2	40	40	nothing	2
3	80	60	Merit	3

2.4.4 Modified Condition/Decision Coverage (MC/DC)

MC/DC testing builds on the previous technique by adding one more requirement. As with Decision Condition testing, we are required to exercise every decision outcome (decision coverage) and every condition outcome (condition coverage). The additional requirement is that we must show that each condition can independently affect the overall outcome of the decision.

Consider again our example decision statement:

if **a > b** and **c > d** and **e > f** ...

The minimum number of test cases required to achieve 100% modified condition decision coverage = (n + 1) where n is the number of conditions. In our example there are three conditions so we will need at least 4 test cases to achieve 100% coverage. These are shown in the table below.

Test Case	a > b	c > d	e > f	Decision Outcome
1	T	T	T	T
2	F	T	T	F
3	T	F	T	F
4	T	T	F	F

Test case 1 sets all the conditions to True and this also makes the decision outcome True. Test case 2 demonstrates that by changing the value of the first condition while keeping the

values of the other conditions the same, the decision outcome is changed. Similarly test case 3 demonstrates the effect of changing only the second test condition and test case 4 demonstrates the effect of changing only the third test condition.

With these four test cases we can also see that we have achieved 100% decision coverage and 100% condition coverage. 100% modified condition decision coverage guarantees 100% coverage of both of these weaker coverage measures.

$$Modified\ condition\ decision\ coverage = \frac{Number\_of\_conditions\_shown\_to\_independently\_affect\_the\_decision}{Total\_number\_of\_conditions}\ x\ 100$$

Second MC/DC example

Now we'll look at another example. Consider the following logical expression:

if **a > b** or **c > d** or **e > f** then

This is similar to the previous expression in that it has the same number of conditions and indeed the same conditions but this time the Boolean operators are OR rather than AND.

The minimum number of test cases required to achieve 100% modified condition decision testing is again 4 (n + 1 where n is the number of conditions). These are shown in the table below.

Test Case	a > b	c > d	e > f	Decision Outcome
1	F	F	F	F
2	T	F	F	T
3	F	T	F	T
4	F	F	T	T

Test case 1 sets all the conditions to False, which makes the decision outcome False. Test case 2 demonstrates that by changing the value of the first condition while keeping the values of the other conditions the same, the decision outcome is changed. Similarly test case 3 demonstrates the effect of changing only the second test condition and test case 4 demonstrates the effect of changing only the third test condition.

Where to start?

Compare the two sets of test cases. For the first expression, where the AND operator is used, we start by setting the conditions to True. For the second expression, where the OR operator is used, we start by setting the conditions to False. This is a pattern that we need to follow, that is, operands of an AND operator should be set True for the first test case, whereas operands of an OR operator should be set False for the first test case.

Third MC/DC example

Now we'll consider the Boolean expression:

A or (**B** and **C**).

The easiest way to identify the test cases required for 100% modified condition decision coverage is to start by writing out all the combinations of condition values. Once you have become familiar with the technique you may well prefer to miss this first step and go straight to selecting the required condition combinations. Here is the full set of combinations for the Boolean expression above:

	A or (B and C)			Decision Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

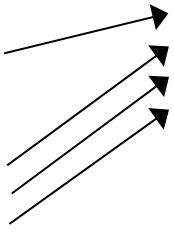
We start by considering the pair of conditions within the parentheses: (B and C). As this uses the 'and' operator, we will need to start with both conditions set to True. This will be either combination 1 (TTT) or combination 5 (FTT), both of which result in a decision outcome of True. Since the first condition in the expression (A) is operated on by the 'or' operator, the A will need to be set False so it is combination 5 (FTT) that we should start with.

With our first combination chosen (FTT), changing the value of B will give us FFT, combination 7. Changing the value of C will give us FTF, combination 6. Both of these result in a decision outcome of False, so in conjunction with our first combination they demonstrate that conditions B and C can independently affect the outcome of the decision. These choices are shown in the table below. The highlighted values show which combinations are used to demonstrate the effect of changing the values of conditions B and C.

	A or (B and C)			Decision Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Finally, we need one more combination to show the effect of changing the value of A from False to True. Because it is the 'or' operator we need to start with the value of A set False and the value of (B and C) set False. We can reuse either of the combinations 6 and 7. We've arbitrarily chosen 6 (FTF) which means we need to select the combination (TTF) that is, combination 2, for our final combination. We could have chosen combination 7 (FFT) which would have led us to select combination 3 (TFT) for our final combination.

	A or (B and C)			Decision Outcome
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	T
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



Test Case	A or (B and C)			Decision Outcome
1	T	T	F	T
2	F	T	T	T
3	F	T	F	F
4	F	F	T	F

The last step is to specify the test cases as we have done in the table on the right hand side above.

MC/DC example involving coupling

As mentioned previously we need to be wary of coupling. This is where the same variable appears more than once across a set of conditions. For an example, consider the following Boolean expression:

distance = "Longhaul" and (**fareType = "Business"** or **fareType = "First Class"**)

Using our simple notation this becomes: A and (B or C). However, the variable fareType occurs in conditions B and C so they are not independent but are said to be 'coupled'. Because of this, not all combinations of condition values will be possible - impossible combinations must be eliminated. In this example, conditions B and C are coupled, they cannot both be True at the same time. So in the table below we have eliminated combinations 1 and 5.

It is also better to highlight the coupling of the conditions like this: A and (B1 or B2). This is used to indicate that B1 and B2 are coupled, though not the same.

	A and (B1 or B2)			Decision Outcome
~~1~~	~~T~~	~~T~~	~~T~~	~~T~~
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
~~5~~	~~F~~	~~T~~	~~T~~	~~T~~
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Having eliminated the impossible combinations we have to work with the remaining combinations. In this case we start with combination 4, change B1 and B2 in turn using combinations 2 and 3, then finally need combination 6 to show the effect of changing A in respect of combination 2. This will normally work out OK for modified decision condition coverage, but does reduce the number of test cases required for multiple condition coverage (discussed in the next sub-section).

We will usually find the required combinations to achieve 100% modified condition decision coverage. However, it is possible that this will not be the case, we will not be able to achieve 100% modified condition decision coverage. This will happen if the logic of the Boolean expression is, well, a little odd! The next example shows this.

Another MC/DC example involving coupling

Consider this Boolean expression:

(**isMale = True** and **age >= 65**) or (**isMale = False** and **age >=60**)

In our simple notation this becomes: (A and B1) or (not A and B2). However, the two occurrences of condition A are mutually exclusive so when the first A is True, the second A must be False and vice versa, so the second occurrence becomes 'not A'. These interdependencies eliminate combinations 1, 2, 5, 6, 11, 12, 15 and 16 in the table below. Conditions B1 and B2 are also coupled: if B1 is True, B2 must be True also. Similarly, if B2 is False, B1 also must be False. These interdependencies cause combinations 4 and 10 to be eliminated (in addition to those already eliminated).

	(A and B1) or (not A and B2)				Decision Outcome
1	T	T	T	T	T
2	T	T	T	F	T
3	T	T	F	T	T
4	T	T	F	F	T
5	T	F	T	T	T
6	T	F	T	F	F
7	T	F	F	T	F
8	T	F	F	F	F
9	F	T	T	T	T
10	F	T	T	F	F
11	F	T	F	T	F
12	F	T	F	F	F
13	F	F	T	T	T
14	F	F	T	F	F
15	F	F	F	T	F
16	F	F	F	F	F

This leaves only 6 combinations for us to work with! In this case, 100% multiple condition coverage can be achieved only to the extent of covering all possible combinations, just 6 rather than 16 for this Boolean expression.

Similarly, 100% modified condition decision coverage is strictly impossible as it is only conditions B1 and B2 that can be shown to independently affect the decision outcome! A compromise can be used that recognises that conditions A and not A are opposites, so we allow both conditions to change. This is shown with combinations 7 and 13 and is highlighted by thick borders to the cells.

Note that the rule for calculating the minimum number of test cases required to achieve 100% modified condition decision coverage (n + 1, where n is the number of conditions) may not apply if any of the conditions are coupled.

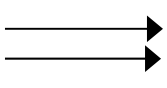
MC/DC example with a compound decision statement

Now let us consider modified condition decision coverage for the compound decision statement below.

```
if (A and B)
        do this
else
        if (C or D)
                do that
endif
```

We need 3 test cases to achieve 100% modified condition decision coverage of the first decision statement, 2 of which have a False outcome and will go on to exercise the second decision statement. The second decision statement also requires 3 test cases to achieve our target coverage. Two of these test cases we have already, so we need add only one more to achieve 100% modified condition decision coverage of the code, making a total of 4 test cases.

	A and B		Decision Outcome
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F



	C or D		Decision Outcome
5	T	T	T
6	T	F	T
7	F	T	T
8	F	F	F

Note that in this example it doesn't matter which combination of the second decision statement is covered by which test case. This is because the two decision statements comprise only independent conditions. Here are the test cases.

Test Case	Inputs				Expected Outcome*
	A	B	C	D	
1	T	T	-	-	Do this
2	T	F	T	F	Do that
3	F	T	F	T	Do that
4	F	F	F	F	Nothing

*Note that we should not really be stating an expected outcome because we do not have a specification. Expected outcomes should be derived from an understanding of what the software should do, not what it actually does!

MC/DC example with coupling and a compound decision statement

As we mentioned earlier, we need to look out for multiple occurrences of the same or related variables. To show an example of this we will go back to showing the full expressions.

if **order_quantity <= number_in_stock** and **payment_status = validated** then
    process the whole order
elseif **order_quantity > number_in_stock** and **payment_status = validated** then
    process a partial order
endif

In our simplified notation this can be:

if **A and B**
    do this
else
    if **not A and B**
        do that
endif

We can repeat condition B because it is the same in both conditions. We can also recognise that the second occurrence of condition A is really the inverse of the first, so we indicate this by using '**not A**' – as we have done below.

	A and B		Decision Outcome
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

	(not A) and B		Decision Outcome
5	T	T	T
6	T	F	F
7	F	T	F
8	F	F	F

Notice that, because of the coupling, combination 3 from the first decision gives us combination 5 for the second decision. Also, combination 2 for the first decision means that we will have to test combination 8 for the second decision, even though we don't actually need it for MC/DC coverage of that second decision statement. To achieve 100% MC/DC coverage of the second decision statement we require combination 6, which dictates that we use combination 4 on the first decision statement even though it is not required for coverage of the first one. We would also like combination 7 but this is impossible because when 'not A' is False, 'A' will be True, and as 'B' is True for combination 7 the outcome of the first decision will be True, thus preventing us from reaching the second decision statement.

So we need 4 test cases and these are:

Test Case	Inputs		Expected Outcome
	A	B	
1	T	T	Process whole order
2	T	F	Nothing
3	F	T	Process partial order
4	F	F	Nothing

It is not important how we represent the conditions, as we are using the simplified notation as a means to an end (i.e. to help us determine the test cases necessary to achieve a specific level of structural coverage). Use whatever works for you. The important thing is to determine the required set of test cases without spending too much time on it!

Applicability

This technique is used extensively in the aerospace software industry and many other safety-critical systems. It should be used when dealing with safety critical software where any failure may cause a catastrophe.

Limitations/Difficulties

Achieving MC/DC coverage may be complicated when there are multiple occurrences of a specific term in an expression, leading to the "coupling" considerations that are described above. One approach to addressing this issue is to specify that only uncoupled atomic conditions must be tested to the MC/DC level. The other approach is to analyze each decision in which coupling occurs on a case-by-case basis.

As also described above, short-circuiting may affect the ability to attain MC/DC coverage since some required tests may not be achievable.

2.5 Multiple Condition Testing

Learning Objective

TTA 2.4.4 K3 Write test cases for a given specification item by applying the Multiple Condition test design technique to achieve a defined level of coverage

From the ISTQB Glossary

multiple condition coverage: The coverage of combinations of all single condition outcomes within one statement that have been exercised by a test suite.

multiple condition testing: A white-box test technique in which test cases are designed to execute outcome combinations of atomic conditions.

Multiple condition testing aims to exercise all combinations of condition values. Multiple condition coverage is therefore the number of condition outcome combinations exercised by a set of tests. This is calculated by:

$$Multiple\ Condition\ Coverage = \frac{Number\_of\_condition\_outcome\_combinations\_executed}{Total\_number\_of\_condition\_outcome\_combinations} \ x\ 100\%$$

Multiple condition coverage requires the conditions in every decision statement to be exercised in all combinations of their outcomes. So for the code fragment shown below, eight test cases are needed such that the three conditions in the decision statement are exercised with all of the outcome combinations, as shown in the table.

```
if a > b and c > d and e > f then
    print "Decision is true"
end if
```

The minimum number of test cases required to achieve 100% multiple condition coverage is 2^n where n is the number of uncoupled conditions (coupling will reduce the number of combinations that are possible, as explained above). The decision statement above consists of three uncoupled conditions (**a > b**, **c > d**, and **e > f**) so there are 2 to the power of 3 combinations = 2^3 = 8 combinations and therefore 8 test cases are required:

Test Case	a > b	c > d	e > f	Decision Outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Note that 100% condition combination coverage guarantees 100% modified condition decision coverage as well as 100% decision coverage and 100% condition coverage.

Example with a compound decision statement

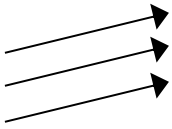
Consider this compound decision statement:

```
if (A and B)
        do this
else
        if (C or D)
                do that
endif
```

This is actually a series of two decision statements in which the second decision will only be executed if the outcome of the first decision is False. So we may need more test cases designed specifically to exercise the condition value combinations of the second decision statement.

To calculate the minimum number of test cases required to achieve 100% multiple condition or modified condition decision coverage, we need to consider each decision statement in turn and then identify which of the test cases aimed at covering the first decision statement will have a false outcome and so go on to exercise the second decision statement.

	A and B		Decision Outcome
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F



	C or D		Decision Outcome
5	T	T	T
6	T	F	T
7	F	T	T
8	F	F	F

The first decision statement requires 4 test cases to achieve 100% multiple condition coverage; three of these result in a False outcome for this decision statement, so they go on to exercise the second one. The second decision statement also requires 4 test cases to achieve 100% multiple condition coverage, so we need to add one more test case that will have a False outcome from the first decision. We therefore need a minimum of 5 test cases to achieve 100% multiple condition coverage for the sample piece of code. These are shown below:

Test Case	Inputs				Expected Outcome*
	A	B	C	D	
1	T	T	-	-	Do this
2	T	F	T	T	Do that
3	F	T	T	F	Do that
4	F	F	F	T	Do that
5	F	F	F	F	Nothing

*Note that we should not really be stating an expected outcome because we do not have a specification. Expected outcomes should be derived from an understanding of what the software should do, not what it actually does!

For test case 5 we could have chosen different values for inputs A and B. This test case requires the first decision to have a False outcome. Since all the combinations of input values A and B that result in a False outcome have already been exercised by test cases 2, 3 and 4, we can choose to repeat any one – we have chosen to repeat the values used by test case 4.

Applicability

This technique is useful In the rare cases when it is necessary to test all possible combinations of atomic conditions that a decision may contain, for example, when testing embedded software which is expected to run reliably without crashing for long periods of time (e.g., telephone switches that were expected to last 30 years).

Limitations/Difficulties

Because the number of test cases can be derived directly from a truth table containing all of the atomic conditions, this level of coverage can easily be determined. However, the sheer number of test cases needed makes MC/DC coverage more feasible in most situations.

Short-circuiting may reduce the number of actual test cases needed, depending on the order and grouping of logical operations that are performed on the atomic conditions.

2.6 Basis Path Testing

Learning Objective

TTA 2.6.1 K3 Write test cases from a given specification item by applying McCabe's Simplified Baseline Method

From the ISTQB Glossary

branch: A transfer of control from a decision point.

feasible path: A path for which a set of input values and preconditions exists which causes it to be executed.

infeasible path: A path that cannot be executed by any set of input values and preconditions.

path: A sequence of events, e.g., executable statements, of a component or system from an entry point to an exit point.

path coverage: The coverage of paths.

path testing: A white-box test technique in which test cases are designed to execute paths.

Path testing in general consists of identifying paths through the code and then creating tests to cover them. Ideally, it would be good to test every unique path. However, except for very small and simple code modules, this would be an impossibly large task (Foundation level Principle 2: exhaustive testing is impossible). It becomes impractical very quickly when there are loops in the code, because each iteration of the loop counts as an individual path: a path that travels round a loop 3 times is different from the path that travels round the same loop 4 times, even if those parts of the paths outside of the loop are identical.

Basis path testing provides a way to reduce the challenge to a manageable number of tests. There is more than one method.

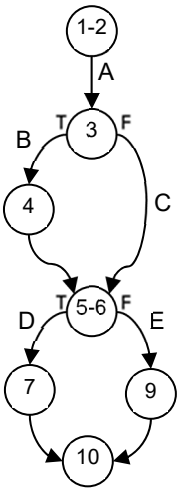
Boris Beizer recommends a systematic approach in]Beizer90]. The syllabus prefers the Simplified Baseline Method developed by McCabe [McCabe96].

1. **Create a control graph for the specification item being considered.**
2. **Choose a 'baseline' path through the code.** This should be the most important path to test: judgement will be needed here, perhaps in consultation with the user / customer, and a risk-based approach could be used.
3. **Generate a second path** by changing the outcome of the first decision on the baseline path and identifying the rest of the path by ensuring that its maximum number of decision outcomes is the same as on the baseline path.
4. **Generate the third path** by starting with the baseline path and changing the outcome of its second decision. (If a multiway decision is encountered, e.g. a case statement, each of its outcomes should be exercised before moving on to the next decision).

5. **Generate further paths** by continuing to change decision outcomes on the baseline path until all have been changed. As each new decision is considered, choose its most important outcome first.
6. **Generate remaining paths.** When all the decision outcomes on the baseline path have been covered, apply the same approach to the second path, then to the third, and so on until all decision outcomes in this 'specification item' have been exercised.

As an example, consider the code below. The control flow graph is also shown and the path segments are labelled A, B, C, D and E. Note that path segments B, D and E are each made up of two branches.

```
5.  Read numInStock
6.  Read numOrdered
7.  IF (numOrdered <= numInStock) THEN
8.      Print "In stock"
9.  ENDIF
10. IF (numOrdered > 10) THEN
11.     Print "Discount applies"
12. ELSE
13.     Print "No discount"
14. ENDIF
```



There are four paths through this code. They are listed in the table below, together with the tests that cover them, in the order in which they were identified. We established that the most important path is the one in which the item is in stock and discount applies, so that was chosen as the baseline. For the second path we changed the decision outcome at node 3. For the third path we returned to the first one and changed its decision outcome at node 6. Having now changed all the decision outcomes on the baseline path we took the second path and changed its decision outcome at node 6.

Path	numInStock	numOrdered	Expected Result
A, B, D	12	11	In stock. Discount applies.
A, C, D	11	12	Discount applies.
A, B, E	5	4	In stock. No discount.
A, C, E	8	9	No discount.

Path coverage is sometimes mistakenly taken for branch or decision coverage because both these techniques seek to cover 100% of the *path segments* through the code. However, path coverage is really about covering all paths through the code from beginning to end. This amounts to testing all combinations of path segments, not just individual segments.

Applicability

The Simplified Baseline Method of path testing, described above, is often used for testing safety critical software. It is a good complement to the techniques covered above because its focus is on paths through the code rather than just the decision statements.

Limitations/difficulties

While this technique can be performed manually, for more complex code specialized tools are needed that may be very expensive. Availability of such tools is limited. Therefore, it is often limited to those systems whose safety or mission criticality justifies the expense.

Coverage

The technique described above should ensure full coverage of all the 'linearly independent paths' – that is, unique paths that don't loop. The number of these paths should match the cyclomatic complexity of the code and so can be calculated (see later for explanation of cyclomatic complexity). Depending on the complexity of the code, it may be useful to use a tool to check that full coverage of the basis set of paths has been achieved.

Coverage is measured as the number of linearly independent paths executed by the tests divided by the total number of linearly independent paths in the test object, normally expressed as a percentage. Achieving 100% basis path coverage (disregarding multiple paths around loops) guarantees 100% decision coverage (and accordingly 100% statement coverage too). According to the syllabus, path testing provides more thorough testing than decision testing and usually with only a relatively small increase in the number of tests.

2.7 API Testing

Learning Objective

TTA 2.7.1 K2 Understand the applicability of API testing and the kinds of defects it finds

From the ISTQB Glossary

application programming interface: A type of interface in which the components or systems involved exchange information in a defined formal structure.

An Application Programming Interface (API) is a published specification that describes how one software program can communicate with another. (The syllabus calls it "code", but strictly speaking code is what we implement it with, not what it intrinsically is). An API is effectively the equivalent of a graphical user interface (GUI) for use by other software products. Testing one is similar to testing a GUI: the focus is on the choice of input values and the evaluation of returned data.

The syllabus points out that API testing is not really a technique; it calls it a "type of testing", but even that doesn't sit well with the Foundation definition of test types, and it might be better to regard it as a special type of test object that requires a particular approach to testing. The API itself isn't the test object: the interface that has been built with it is the test object.

APIs are often used in a client/server relationship where one process supplies some kind of functionality to other processes. API testing is therefore relevant if the product we are testing provides an API for the customers of the product to use – allowing them to build their own applications that interact with our product. It is our responsibility to test the API of our product.

For example, Adobe Acrobat® provides an API that allows developers to write applications that interact with Adobe Acrobat directly. Here is one of the functions provided (this has been simplified a little):

PDDocInsertPages (doc1, mergeAfterPage, doc2, startPage, numPages, insertFlags)

This function allows pages from one PDF documented to be inserted into a different PDF document. Returns true if successful, false if not. The parameters are:

doc1	–	reference to the document into which pages are inserted.
mergeAfterPage	–	page number in doc1 after which pages from doc2 are inserted.
doc2	–	reference to the document containing the pages that are inserted into doc1.
startPage	–	page number of first page to be inserted.
numPages	–	the number of pages to be inserted.
insertFlags	–	flags that determine what information is inserted (e.g. whether or not to include bookmarks and/ or threads)

Testing an API is similar to testing a GUI: the focus is on the choice of input values and the evaluation of returned data. Negative testing of APIs can also be important, because programmers using APIs to access external services may find creative ways of using the interfaces (often unintentionally). For example, in the InsertPages function above, the numPages parameter should always be a positive integer and have a maximum value equal to the number of pages in doc2. Simple programming errors can lead to these conditions being violated. APIs therefore need to have robust error handling built in and this will need testing.

Combinatorial testing may be useful where an API contains several parameters, particularly if some parameters are optional or where data values allowed with each parameter can be partitioned. In the example above, we may wish to test different combinations of insertFlags. Combinatorial testing may also be useful where two or more APIs are used in combination to test across the different interfaces.

APIs are frequently loosely coupled, in that the different ends of the interface will be controlled by different processors. This results in the possibility of lost transactions or timing problems, so performance testing focused on timing problems may be appropriate. It also necessitates thorough testing of the recovery and retry mechanisms.

An organization that provides an API interface must ensure that all services have very high availability; this often requires strict reliability testing by the API publisher, as well as good infrastructure support.

Applicability

API testing is becoming more important as more systems become distributed or use remote processing as a way of off-loading some work to other processors. Examples include

- operating systems calls
- service-oriented architectures (SOA, e.g. SOAP and REST)
- remote procedure calls (RPC)
- web services
- distributed applications, particularly those using software containerization [Burns18] which results in the division of a software program into several containers which communicate with each other using mechanisms such as those listed above.

Limitations/difficulties

Testing an API directly usually requires a Technical Test Analyst to use specialized tools. Because there is usually no real GUI associated with an API, tools may be needed to set up the execution environment, invoke the API, enter the data and see the result.

Coverage

API testing is a description of a type of testing; it does not denote any specific level of coverage. At a minimum the API test should exercise all calls to the API as well as all valid parameter values and a reasonable sample (equivalence partitioning might be useful here) of invalid parameter values.

Types of defect

The types of defect that can be found by testing APIs are many and varied. Interface issues are common, as are data handling issues, timing problems, loss of transactions and duplication of transactions.

2.8 Selecting a White-box Test Technique

Learning Objective

TTA 2.8.1 K4 Select an appropriate white-box test technique according to a given project situation.

The context of the system under test will influence its product risk and criticality levels, and these in turn will, as the syllabus states, help to determine "the required coverage metric" (and hence the white-box test technique to use) and the depth of coverage to be achieved". In other words, if the SUT's criticality demands MC/DC coverage to be measured (the "coverage metric") then the MC/DC technique must be used. Whichever the technique, the more critical the system is, the higher the level of coverage that is needed, and this usually means that correspondingly more time and resource will be needed to achieve it.

Standards

Sometimes the level of coverage required may be derived from standards that apply to the software system. For example, if the software were to be used in an airborne environment, it may be required to conform to standard DO-178B (in Europe, ED-12B). This standard identifies 5 levels of integrity and requires different coverage levels to be achieved for each integrity level as shown in the table below.

Integrity Level	Testing Requirement
A. Catastrophic: failure may cause lack of critical function needed to safely fly or land the plane.	MC/DC
B. Hazardous: failure may have a large negative impact on safety or performance.	Decision Coverage
C. Major: failure is significant, but less serious than A or B.	Statement Coverage
D. Minor: failure is noticeable, but with less impact than C.	No requirement for structural testing.
E. No effect: failure has no impact on safety.	

Likewise, IEC-61508 is an international standard for the functional safety of programmable electronic safety-related systems. This standard has been adapted in many different areas, including automotive, rail, manufacturing process, nuclear power plants and machinery. These standards use 4 safety integrity levels and either recommends or highly recommends coverage levels as shown in the table below.

Safety Integrity Level (SIL)	SIL Description	Testing Requirement (100% coverage)
4 (most critical)	Varies according to the industry sector that has adopted it. E.g., the automotive sector's implementation is called ASIL; it has ASIL levels A, B, C and D of which D is the most critical and is defined as a failure that could cause serious injury or death	MC/DC 'Highly recommended' (so are 100% Branch [Decision] and Statement coverage, which are of course guaranteed by 100% MC/DC)
3		MC/DC 'Recommended'; Branch [Decision] and Statement 'Highly recommended'
2		Statement coverage 'Highly recommended', Branch [Decision] coverage 'Recommended'
1 (least critical)		Branch [Decision] and Statement coverage both 'Recommended'

Other considerations

The structure of code can influence the choice of structural testing techniques. For example, for decision testing to be relevant the code must contain at least one decision statement! The following table describes the code constructs that need to be present before each structural testing technique becomes appropriate. The table also gives statements about the applicability of each technique as stated by ISTQB in the syllabus.

Technique	Code Structure Required	Applicability (According to ISTQB)
Statement Testing	Always applicable	Should be considered as a minimum for all code being tested
Decision Testing	Code must contain at least one decision statement	Should be considered when the code being tested is important or even critical
Condition Testing	Code must contain at least one decision statement with a Boolean expression that has two or more conditions	No longer mentioned in syllabus (was in older one); not generally recommended because it does not guarantee decision coverage
Condition Decision Testing	Same as for Condition Testing	No longer mentioned in syllabus (was in older one); not generally recommended because MC/DC is stronger for little extra cost
MC/DC	Same as for Condition Testing	Should be used for safety-related and other high integrity software
Multiple Condition Testing	Same as for Condition Testing	Rare circumstances; may be used to test embedded software with a high reliability requirement
Basis Path Testing	Code must contain at least two decision statements in series (i.e. not nested).	Mission critical software; good addition to the other methods

In modern systems, it is rare that all processing will be done on a single system. API testing should be instituted whenever some of the processing is going to be done remotely. The criticality of the system will determine how much effort should be invested in API testing.

Using white-box techniques

White-box test techniques serve two purposes: test measurement and test case design. They are often used in the first instance to assess the amount of testing performed by tests derived from functional techniques. They are then used to design additional tests with the aim of increasing the test coverage. So, white-box techniques are normally used after an initial set of tests has been derived using black-box techniques.

Coverage measurement is best done using tools, and there is a variety of such tools on the market. These tools can help to increase productivity and quality. They help us to increase quality by ensuring that more structural items are tested, so defects within those structural items can be found. They may also increase productivity and efficiency by highlighting tests that may be redundant, i.e. tests that exercise the same structure as other tests (although this is not necessarily a bad thing!).

Coverage techniques are a good way of generating additional test cases that are different from existing tests, and in any case they help ensure breadth of testing in the sense that test cases that achieve 100% coverage in any measure will be exercising all parts of the software. There is also danger in these techniques. 100% coverage does not mean 100% tested. Coverage techniques measure only one dimension of a multi-dimensional concept. Two different test cases may achieve exactly the same coverage but the input data of one may find a defect that the input data of the other doesn't. Furthermore, coverage techniques measure coverage of the software code that has been written but cannot say anything about the software that has not been written. If a function has not been implemented in the software, it is the functional testing techniques that are more likely to reveal the omission.

***** Principle 7 - Absence-of-errors is a fallacy *****

In common with all structural testing techniques, coverage techniques are best used on areas of software code where more thorough testing is required. Safety critical code, code that is vital to the correct operation of a system, and complex pieces of code are all examples of where structural techniques are particularly worth applying. They should always be used in addition to functional testing techniques rather than as an alternative to them.

The Foundation syllabus makes the point that structural techniques can be used at any test level. They are, however, most often used at component and component integration levels.

The Foundation Level syllabus also states that, at these levels, the code under test can be part of the test basis – that is, the body of knowledge that is used to determine expected behaviour. It's vital to remember that this is true only for determining the pre-conditions and inputs that will allow you to reach those parts of the code that you want to test. If that code is also used to predict the expected results of running it then the test will prove only that the code does what it appears to do, whether that is right or wrong, and if the test designer has read the code correctly then of course it will – except only in the very rare case of a defect in the compiler. To verify that the code is doing what it should do, it is essential to use the specification or some other part of the test basis to determine expected outcomes. This is especially important when testing safety-critical software, which is exactly where the stronger code-based techniques are most often used. We have seen an organisation getting this wrong and had to warn them that they could be sent to prison and/or sued out of business if code that they had 'tested' this way failed and thereby, caused death or serious injury!

