Exercise walk-through

12.3.1 Test strategy, test plans and choosing a testing method

A-Level Computer Science

You must be able to

- show understanding of ways of exposing and avoiding faults in programs
- show understanding of the methods of testing available, and select an appropriate method and appropriate data for a given situation
- show understanding of the need for a test strategy and a test plan, and their likely contents
- choose appropriate test data, including **normal**, **abnormal** and **extreme**/**boundary** values, each with its expected result
- analyse an existing program and make amendments to enhance functionality

1 The method

Method — Exposing a fault, and avoiding one

The syllabus separates these, and a question that asks for one will not accept the other.

- **Exposing** a fault (finding one that is already there): testing against a test plan; a **dry run** 手工跟踪 with a trace table; a **walkthrough** 走查 with colleagues; the IDE's debugger – breakpoints, single stepping, watching variables.
- **Avoiding** a fault (stopping it being written): designing before coding, with a structure chart and pseudocode; modular code with meaningful identifiers and comments; **validation** of every input; handling exceptions rather than letting a run-time error crash the program; the IDE's syntax checking as you type.
- A walkthrough is worth knowing well: colleagues read the code without the author's assumptions, it checks the code against the **design** rather than only against test data, it spreads knowledge of the code for later maintenance, and it needs no test data and no working computer, so it can happen early.

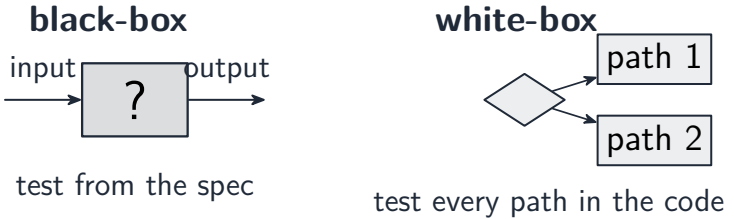
Method — Which method, and when

Method	Written from	Done by	Use it when
dry run / trace table	the algorithm	the programmer, on paper	no computer is needed and the logic is suspect
walkthrough	the code and the design	the author with colleagues	early, to check the code against the design
white-box 白盒测试	the code's internal structure	someone who can see the code	every statement, branch and loop must be exercised
black-box 黑盒测试	the specification only	a tester or user, with no sight of the code	only inputs and outputs matter
integration	the module interfaces	the developers	modules pass alone but may not pass data correctly
alpha	the whole program	the developers, in house	before any outsider sees it
beta	the whole program	a sample of real users, in their own environment	to find faults real use exposes
acceptance	the requirements	the customer	to decide the product is fit for purpose, before paying

Method — Which method, and when

- The order at the end is fixed and often asked: module testing, then **integration**, then **alpha**, then **beta**, then **acceptance**.
- **Integration** testing exists because modules that each pass on their own can still fail together, when the data passed between them is of the wrong type or in the wrong order.

Method — Which method, and when

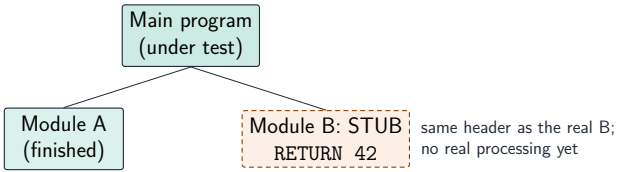


Black-box testing works from the specification; white-box tests the code's internal paths

Method — Testing before every module exists

A **stub** 桩 is a placeholder for a module that has not been written yet. It has the **real header** – the same name, parameters and return type – but its body simply returns a fixed value.

That lets testing start from the top down: the module above it can be called and checked now, because every call it makes is answered. Without stubs, nothing can be tested until everything is finished.



Main can be run and tested before B is written; the stub is replaced by the real module later

Method — Strategy and plan are not the same thing

- A **test strategy** is the **high-level approach**: which kinds of testing will be used at which stage, who is responsible for each, what test data will be needed, and the criteria for moving to the next stage.
- A **test plan** is the **detailed list of individual tests**: for each one, the module or feature under test, the input data, the **reason** that data was chosen, the **expected** result, a space for the **actual** result, and what to do if they differ.
- The plan is written at the **design** stage, from the specification. Writing it after the code exists tests what the program happens to do rather than what it is supposed to do.

Method — Strategy and plan are not the same thing

- Testing does not end at release. A **system** needs **continuing** maintenance, and the three kinds are told apart by what prompted them: **corrective** maintenance fixes a fault found in live use – one the test plan missed; **adaptive** maintenance responds to a change outside the program, such as a new operating system or a change in the law; **perfective** maintenance adds or improves functionality the user has asked for, when nothing was broken. After any of them, re-run the existing tests to confirm the change has not broken what already worked.

Method — Writing test-plan rows that prove something

Worked example. *A component passes if its weight, measured to the nearest gram, is within 3 g of a target of 50 g – that is, from 47 g to 53 g inclusive. Draw up the test-plan rows.*

Type	Test data	Reason	Expected result
normal	50	a typical value inside the range	accepted
extreme	47	the smallest value still accepted	accepted
extreme	53	the largest value still accepted	accepted
boundary	46 and 47	the pair either side of the lower edge	46 rejected, 47 accepted
boundary	53 and 54	the pair either side of the upper edge	53 accepted, 54 rejected
abnormal	`"heavy"`	not a number at all	rejected

Method — Writing test-plan rows that prove something

- **Every row must carry its expected result**, or the plan proves nothing: without it there is nothing to compare the actual result against.
- **Extreme and boundary are the pair most often confused.** An extreme value sits **inside** the range and is **accepted**. A boundary test is always a **pair**, one either side of an edge – which is exactly where an off-by-one error hides.

2 Practice

Practice 2.1 ▪ 4 marks

State **two** ways of **exposing** a fault in a program, and **two** ways of **avoiding** faults when writing one.

Solution 2.1

Method: Exposing a fault, and avoiding one — p.4

Worked steps

- exposing, any two: testing against a test plan; a dry run with a trace table; a walkthrough with colleagues; using the IDE's debugger with breakpoints and watched variables B1 B1
- avoiding, any two: designing before coding; modular code with meaningful identifiers and comments; validating every input; handling exceptions instead of crashing B1 B1

Practice 2.2 ▪ 2 marks

State the difference between a **test strategy** and a **test plan**.

Solution 2.2

Method: Strategy and plan are not the same thing — p.9

Worked steps

- a **strategy** is the high-level approach: which kinds of testing, by whom, at which stage, and the criteria to move on B1
- a **plan** is the detailed list of individual tests, each with its data, expected result and a space for the actual result B1

Practice 2.3 ▪ 2 marks

A finished program is given to a small group of real users to try in their own homes before it is released. Name this type of testing, and name the type that follows it.

Solution 2.3

Worked steps

- **beta** testing B1
- it is followed by **acceptance** testing, in which the customer decides whether the product meets the requirements B1

Practice 2.4 ▪ 2 marks

State what a **stub** is, and give **one** reason for using one.

Solution 2.4

Worked steps

- a **stub** is a placeholder for a module that has not been written yet: it has the real header but simply returns a fixed value B1
- any one reason: it lets the modules above it be tested now, so testing can proceed top-down without waiting for every module to exist B1

Practice 2.5 ▪ 1 mark

Besides the input data, state **one** thing that every row of a test plan must contain.

Solution 2.5

Worked steps

- the **expected result** (also accepted: the reason the data was chosen, or a space for the actual result) B1

3 Exam-style

Exam-style 3.1 ▪ 6 marks

A program accepts a delivery weight in kilograms. A delivery is accepted if the weight is from 5 to 25 inclusive.
Complete the test plan. For each row give the test data, the reason it was chosen, and the expected result.

Type	Test data	Reason	Expected result
normal			
extreme (low)			
extreme (high)			
boundary			
abnormal			
abnormal			

Solution 3.1

Method: Writing test-plan rows that prove something — p.11

Worked steps

- one mark per correct row: the data must match the type **and** carry a sensible reason and expected result B1 *x 6*

Type	Test data	Reason	Expected result
normal	15	a typical value inside the range	accepted
extreme (low)	5	the smallest value still accepted	accepted
extreme (high)	25	the largest value still accepted	accepted
boundary	4 and 5	the pair either side of the lower edge	4 rejected, 5 accepted
abnormal	30	above the range	rejected
abnormal	`"ten"`	not a number	rejected

Solution 3.1

- a boundary row naming only one value scores nothing: the point of a boundary test is the **pair**

Exam-style 3.2 ▪ 4 marks

For each situation, name the **most appropriate** testing method and give a reason.

Exam-style 3.2 (a) ▪ 2 marks

A tester who is not allowed to see the source code must check the program against the specification.

Solution 3.2 (a)

Method: Which method, and when — p.5

Worked steps

black-box testing B1

- it is designed from the specification only, so it needs no sight of the code: the tester supplies inputs and checks the outputs B1

Exam-style 3.2 (b) ▪ 2 marks

Two modules each work correctly on their own, but the program fails when they are used together.

Solution 3.2 (b)

Worked steps

integration testing B1

- it tests the **interfaces** between modules, which is where data of the wrong type or in the wrong order is passed B1

Exam-style 3.3 ▪ 4 marks

Explain the difference between **white-box** and **black-box** testing, and state who is able to carry out each.

Solution 3.3

Method: Which method, and when — p.5

Worked steps

- **white-box** testing is designed from the code's internal structure, to exercise every statement, branch and loop B1
- it can only be carried out by someone who can see the source code, normally the programmer B1
- **black-box** testing is designed from the specification, and checks only that given inputs produce the expected outputs B1
- it can be carried out by a tester or a user with no sight of the code B1

Exam-style 3.4 ▪ 3 marks

A programmer has finished a module that calls a second module.
The second module has not been written yet.
Describe how the programmer can test the finished module now.

Solution 3.4

Method: Testing before every module exists — p.8

Worked steps

- write a **stub** in place of the second module B1
- give it the same header – name, parameters and return type – as the real module will have B1
- have it return a fixed value, so every call the finished module makes is answered and the finished module can be tested now B1

Exam-style 3.5 ▪ 2 marks

A test plan is written at the **design** stage, before any code exists. Explain why it is not written after the program has been coded.

Solution 3.5

Method: Strategy and plan are not the same thing — p.9

Worked steps

- the plan is written from the **specification**, so it tests what the program is **supposed** to do B1
- a plan written after the code would be based on the code, so it would test what the program happens to do and would not expose a requirement the programmer misunderstood B1