A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

12.3.1 Test strategy, test plans and choosing a testing method

Name: _____ Class: _____ Date: _____ **Total: 30 marks**

KEY WORDS black-box 黑盒测试 • walkthrough 走查 • stub 桩 • run-time error 运行时错误 • white-box 白盒测试

Objective

Build the skills to answer the **testing** questions that sheet 12.3 only names. That sheet identifies the three kinds of error and the three kinds of test data; this one is about the decisions around them: how a fault is **exposed** as against **avoided**, which testing method suits a given situation, what a **test strategy** 测试策略 and a **test plan** 测试计划 each contain, and how to write test-plan rows that actually prove something.

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 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

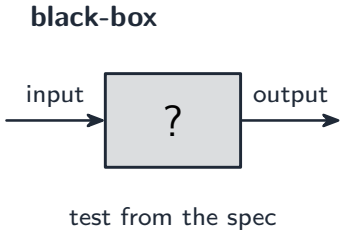
■ Exposing a fault, and avoiding one

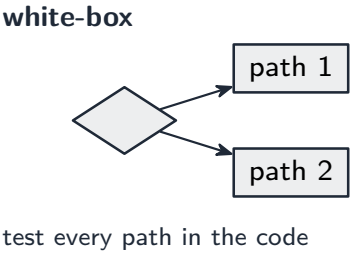
The syllabus separates these, and a question that asks for one will not accept the other.

- **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.

Which method, and when

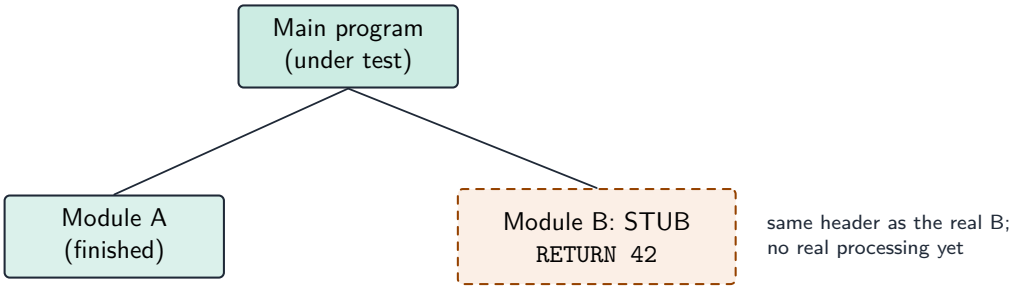




Method	Written from	Done by	Use it when
dry run / trace table	the algorithm	the programmer, on paper	no computer is needed and the logic is suspect
walkthrough	the code and the design	the author with colleagues	early, to check the code against the design
white-box 白盒测试	the code's internal structure	someone 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

- 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.

Testing before every module exists



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

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.

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.
- 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.

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

- **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

Now use the methods above.

2.1 State **two** ways of **exposing** a fault in a program, and **two** ways of **avoiding** faults when writing one. [4]

2.2 State the difference between a **test strategy** and a **test plan**. [2]

2.3 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. [2]

2.4 State what a **stub** is, and give **one** reason for using one. [2]

2.5 Besides the input data, state **one** thing that every row of a test plan must contain. [1]

3 Exam-style questions

3.1 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. [6]

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

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

(a) A tester who is not allowed to see the source code must check the program against the specification. [2]

(b) Two modules each work correctly on their own, but the program fails when they are used together. [2]

3.3 Explain the difference between **white-box** and **black-box** testing, and state who is able to carry out each. [4]

3.4 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. [3]

3.5 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. [2]

Solutions

Each answer shows the steps, then the **result**.

2.1

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

2.2

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

2.3

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

2.4

- 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
- 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

2.5

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

3.1

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

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

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

3.2

(a) **black-box** testing

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

(b) **integration** testing

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

3.3

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

3.4

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

3.5

- the plan is written from the **specification**, so it tests what the program is **supposed** to do
- 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