

Exercise walk-through

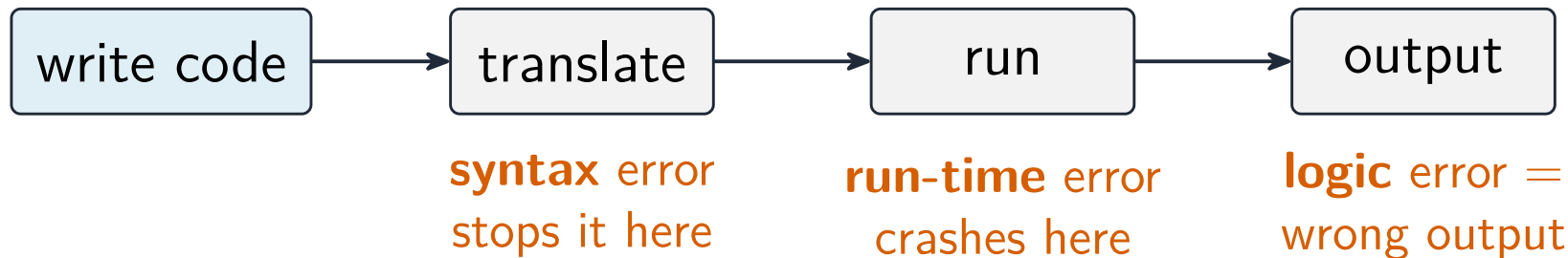
12.3.1

Test strategy, test plans and choosing a testing method

A-Level Computer Science

Three kinds of error

- a **syntax error** breaks the language's grammar – caught at translation, and the program will not run until it is fixed
- a **run-time error** happens while running (divide by zero, file not found, index out of range): it crashes or raises an exception
- a **logic error** runs perfectly and gives the **wrong output** – find it with a **dry run** or a trace table



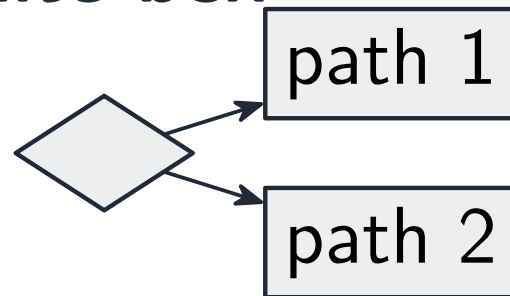
Testing methods

- **black-box testing** – from the spec, ignoring the code
- **white-box testing** – every code path
- a **unit test** checks one routine, with a **stub** standing in for anything it calls; **integration testing** then checks them together
- **alpha** α (in-house), **beta** β (real users), **acceptance testing** (the customer signs it off)

black-box



white-box



Test strategy vs test plan

test strategy

the **high-level approach**:
which kinds of testing,
who runs them, and when

test plan

the **detailed list** of tests:
input data, expected output,
actual output

The plan's "expected vs actual" columns are exactly how logic errors get caught.

Choosing test data, and the stages of the life cycle

normal data

typical values the program should accept

boundary data

the values **at the edge** of the valid range, and just outside it

extreme data

the largest and smallest valid values

abnormal data

values that should be **rejected** – wrong type, out of range, empty

a stub

a placeholder for a module that does not exist yet, so the structure can be tested **top-down**

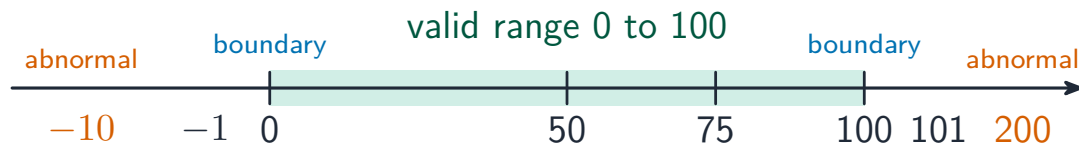
a walkthrough

reading the code aloud against a trace table, before running it at all

A life cycle manages complexity, coordinates teams, tracks progress with milestones, and makes the work reviewable. Naming **why** it exists is the essay question.

Choosing test data

Four kinds of test data for a 0–100 field:



normal (50, 75) inside; extremes (0, 100) the accepted edges; abnormal (−1, 101, −10, 200) rejected

normal

50, 75
well inside

extreme

0, 100
edges, still accepted

abnormal

−10, 200
must be rejected

boundary

0/−1, 100/101
both sides of the line

Exercise 2.1

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



Exercise 2.1

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



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

Exercises 2.2-2.3

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

[2]

test strategy

test plan

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]



Exercises 2.2-2.3

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

[2]

test strategy

test plan

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 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]



2.3

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

Exercises 2.4·2.5

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

[2]

stub

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

test data		

Exercises 2.4-2.5

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

[2]

stub

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

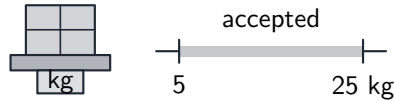
test data		

2.5

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

Exercise 3.1

3.1 A program accepts a delivery weight in kilograms. A delivery is accepted if the weight is from 5 to 25 inclusive.

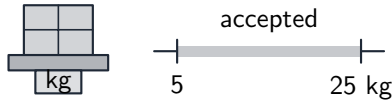


Complete the test plan. For each row give the test data, the reason it was chosen, and the expected result. [6]

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

Exercise 3.1

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			

Full solution 3.1

3.1

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

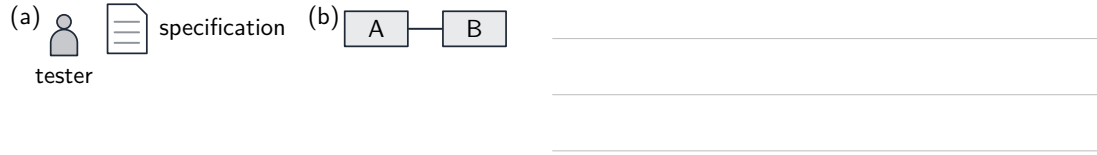
Type	Test data	Reason	Expected result
normal	15	a typical value inside the range	accepted
extreme (low)	5	the smallest value still accepted	accepted
extreme (high)	25	the largest value still accepted	accepted
boundary	4 and 5	the pair either side of the lower edge	4 rejected, 5 accepted
abnormal	30	above 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**

Exercise 3.2

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]

Exercise 3.2

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]

3.2

(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

Exercise 3.3

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

white-box

black-box

Exercise 3.3

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

white-box

black-box

3.3

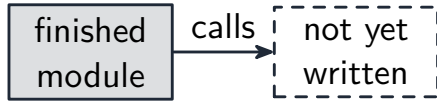
- **white-box** testing is designed from the code's internal structure, to exercise every statement, branch and loop
- it can only be carried out by someone who can see the source code, normally the programmer
- **black-box** testing is designed from the specification, and checks only that given inputs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

Exercise 3.4

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]

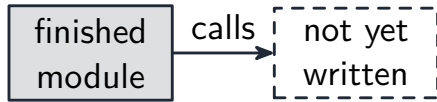


Exercise 3.4

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]



3.4

- 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

Exercise 3.5

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]



Exercise 3.5

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]



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