Exercise walk-through

Program Testing and Maintenance ▪ 12.3

eXercise

You must be able to

- identify **syntax**, **run-time** and **logic** errors
- choose **normal**, **boundary** and **abnormal** test data
- describe testing methods and the three types of **maintenance**

Method — Types of error

Error	When	Example
syntax	at translation — won't run	missing `ENDIF`, misspelt keyword
run-time	while running — crashes	divide by zero, array index too big
logic	runs but **wrong output**	`+` used for `-`, off-by-one loop

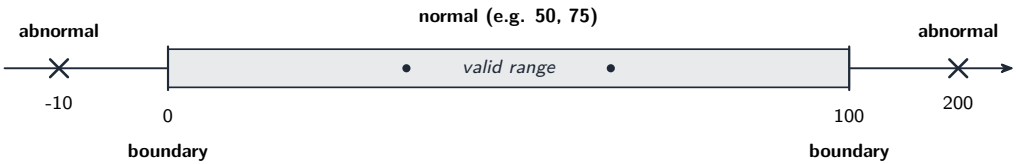
Logic errors are hardest — the only sign is a wrong result, so **trace** and **test** carefully.

Method — Choosing test data

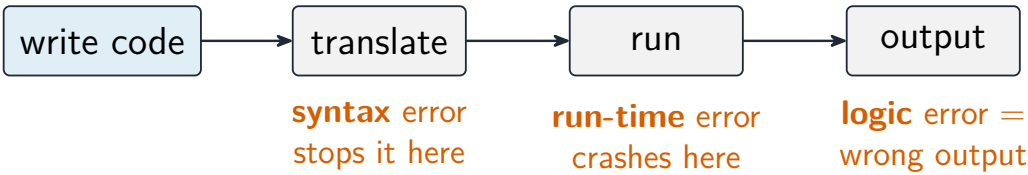
For a field that accepts **0 to 100**, test three kinds:

- **normal** — typical valid values the program should accept (`50`, `75`)
- **extreme** — the **largest and smallest valid** values (`0` and `100`)
- **boundary** — a pair *either side* of a limit, where off-by-one bugs hide (`100` accepted, `101` rejected)
- **abnormal** (erroneous) — values that must be **rejected** (`-10`, `200`, `"abc"`)

Method — Choosing test data



Method — Choosing test data



The error types that testing aims to catch

Method — Methods and maintenance

- **dry run** — trace on paper in a **trace table**; **white-box** tests every path from the code; **black-box** tests inputs→outputs from the spec; **alpha** (in-house) then **beta** (real users); **acceptance** (the customer decides).
- Maintenance: **corrective** (fix bugs), **adaptive** (cope with a changed environment), **perfective** (improve features/speed).

Practice 2.1 ▪ 2 marks

State the difference between a **syntax error** and a **logic error**.

Solution 2.1

Method: Types of error

Worked steps

A **syntax error** breaks the language's grammar and is caught at **translation** (the program won't run) B1; a **logic error** lets the program run but produces the **wrong output** B1.

Practice 2.2 ▪ 1 mark

Give **one** example of a **run-time** error.

Solution 2.2

Method: Types of error

Worked steps

Any one: divide by zero; array index out of range; file not found; invalid type conversion B1.

Practice 2.3 ▪ 3 marks

A field accepts an age from **1 to 120**. Give one **normal**, one **boundary** and one **abnormal** value.

Solution 2.3

Method: Choosing test data

Worked steps

Any valid set, e.g. normal `40`; boundary `1` or `120`; abnormal `0`, `121` or `"abc"` B1 B1 B1.

Practice 2.4 ▪ 1 mark

State what a **dry run** is.

Solution 2.4

Worked steps

Tracing the code **by hand on paper**, recording each variable's value (usually in a trace table) B1.

Exam-style 3.1 ▪ 3 marks

Identify the **type of error** in each case.
(a) the program will not translate because an `ENDWHILE` is missing (b) the program stops with an error when it divides by zero (c) the program runs but the calculated average is always wrong

Solution 3.1

Method: Types of error

Worked steps

(a) **syntax** error; (b) **run-time** error; (c) **logic** error B1 *each*.

Exam-style 3.2 ▪ 4 marks

A field accepts a percentage **0–100**. Complete the test plan with suitable **test data** and a **reason** for each row.

Type	Test data	Reason
normal		
boundary (low)		
boundary (high)		
abnormal		

Solution 3.2

Method: Choosing test data

Worked steps

Any valid plan, e.g.: normal 50 (inside the range); boundary (low) 0 (lowest accepted); boundary (high) 100 (highest accepted); abnormal 150 or −5 (outside — must be rejected) B1 *per correct row: valid data **and** matching reason.*

Exam-style 3.3 ▪ 3 marks

State and describe the **three** types of maintenance.

Solution 3.3

Method: Choosing test data

Worked steps

Corrective — fixing bugs found in use; **adaptive** — changing it to keep working in a new environment (new OS, law); **perfective** — improving features or performance B1 B1 B1.

Exam-style 3.4 ▪ 1 mark

State what **regression testing** checks.

Solution 3.4

Worked steps

That a **change has not broken** existing, previously-working features (B1).

Exam-style 3.5 ▪ 2 marks

A program calculates the average mark of a class. It translates without error, but for one class it outputs an average of 0.
(a) **Identify** the type of error this is, and **explain** how you decided.
(b) **Describe** two ways of **exposing** a fault like this before the program is released.
(c) **Describe** two programming practices that help **avoid** faults of this kind in the first place.
(d) The program is later required to report the **highest** mark as well as the average. **Outline** the amendments you would make, and **state** what you would do afterwards to be sure the average still works.
(e) **Explain** why this later change counts as **perfective** maintenance rather than corrective or adaptive.

Solution 3.5

Method: Methods and maintenance

Worked steps

(a) A **logic error** B1 — the program translates and runs to completion, so the syntax is valid and nothing crashes, but the output is wrong B1.
(b) Any two, developed: **dry run / trace table** — work through the code by hand with sample values, recording each variable, which exposes a division by the wrong count B1 B1. **White-box testing with a debugger** — set a breakpoint and step through, watching the running total and the counter B1 B1. Also acceptable: a walkthrough with a colleague; black-box testing with normal, boundary, abnormal and extreme data.
(c) Any two, developed: **modular design** — split the calculation into a small function

Solution 3.5

that can be tested on its own B1 B1. **Meaningful identifiers and comments**, so `Total / Count` is obviously wrong when `Count` is zero B1 B1. Also acceptable: validating input before use; initialising every variable explicitly; defensive checks such as testing for a zero divisor.

(d) Add a variable to hold the maximum, initialised to the **first mark** rather than zero B1; inside the existing loop compare each mark and update it when the mark is larger B1; extend the output statement B1. Then **re-run the existing tests** for the average — regression testing — to confirm the change has not broken what already worked B1.

(e) It adds new **functionality** that the user has asked for, rather than fixing a fault (which would be corrective) or responding to a change in hardware or operating system (adaptive) B1 B1.