Past-paper walk-through

Program Testing and Maintenance ▪ 12.3

eXercise

Questions in this set

- 9618/21 J25 Q6 ▪ 12 marks
- 9618/22 J25 Q2 ▪ 8 marks
- 9618/22 N24 Q1 ▪ 9 marks
- 9618/22 N24 Q5 ▪ 7 marks
- 9618/21 J24 Q5 ▪ 6 marks
- 9618/22 J24 Q5 ▪ 6 marks

Reading the mark scheme

- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 6

9618/21 J25 ▪ Q6 ▪ 12 marks

A program monitors the speed of vehicles as they move around a large building site.
Each vehicle contains a sensor which reads an integer value that represents the speed of the vehicle. The value is expected to be in the range 0 to 60 inclusive.
The sensors cannot read values less than 0.
A program module has been written to validate the values read by the sensors.
(a) A test plan is needed to fully test the module.
Complete the table. The first line has been completed for you.
Assume the sensors generate only integer values.

Question 6

Type of test data	**Test data value**	**Expected outcome**
normal	36	data item is accepted

Question 6

[4]

Question 6

Type of test data	Test data value	Expected outcome
normal	36	data item is accepted

[4]

Question 6

9618/21 J25 ▪ Q6 ▪ 12 marks

A program monitors the speed of vehicles as they move around a large building site.
Each vehicle contains a sensor which reads an integer value that represents the speed of the vehicle. The value is expected to be in the range 0 to 60 inclusive.
The sensors cannot read values less than 0.

A program module has been written to validate the values read by the sensors.

(b) Write efficient pseudocode for the procedure `Sort()` **[8]**

Solution 6

(a) Worked steps

A test plan needs each **type** of test data, a **value** of that type, and the **expected outcome** — and one mark is given per complete row, so a row missing any of the three scores nothing.

The three types, and what each is for:

- **Normal** — a value comfortably inside the accepted range, which the module should **accept**. For a range of 0 to 60, something like 15 or 36.
- **Boundary (extreme)** — the values at the very edge of the range, on both sides of it. 0 and 1 at the bottom, 59 and 60 at the top, all **accepted**.
- **Abnormal (erroneous)** — a value outside the range, which the module should **reject**. Anything 61 or above.

Solution 6

The question says the sensors generate only integers, and that matters: it rules out 60.5 as an abnormal value, so the abnormal case has to be 61 or more.
Boundary data is the point of the exercise. Almost every validation bug is an off-by-one at the edge — `> 60` written where `>= 60` was meant — and a plan containing only normal and abnormal data would never find it. That is why the boundary rows come in pairs, one either side of the limit. **Mark scheme**

- Type of test data
- Test data value
- Expected outcome normal
- 36 data item is accepted boundary/ extreme/ normal

Solution 6

- 0 / 1 data item is accepted boundary/ extreme/ normal
- 59 / 60 data item is accepted abnormal
- >= 61 data item is rejected normal
- 15 data item is accepted
- Mark as follows:
- 1 mark for each row with (Type and Value and Outcome)

Solution 6

(b) Worked steps

"Efficient" is doing work in this question: a plain bubble sort is not the answer, one that **stops early** is.

```
PROCEDURE Sort()
    DECLARE Temp, J, Boundary : INTEGER
    DECLARE NoSwaps : BOOLEAN

    Boundary <- 1999
    REPEAT
        NoSwaps <- TRUE
        FOR J <- 1 TO Boundary
            IF Reading[J, 1] > Reading[J + 1, 1] THEN
                Temp <- Reading[J, 1]
                Reading[J, 1] <- Reading[J + 1, 1]
                Reading[J + 1, 1] <- Temp
                Temp <- Reading[J, 2]
                Reading[J, 2] <- Reading[J + 1, 2]
                Reading[J + 1, 2] <- Temp

                NoSwaps <- FALSE
            ENDIF
```

Solution 6

Three things earn the efficiency marks:

- **The `NoSwaps` flag.** If a whole pass makes no swap the data is already sorted, so the procedure stops instead of running the remaining passes.
- **`Boundary` shrinking by one each pass.** After pass n the largest n values are already at the end, so re-comparing them is wasted work.
- The `UNTIL` tests **both** conditions, so the loop ends either when nothing moved or when the boundary reaches the start.

Solution 6

△ **Swap both columns.** `Reading` is a 2D array holding a speed and the vehicle ID that produced it; swapping only the speed detaches every reading from its vehicle. The result looks perfectly sorted and is completely wrong — a failure no test of the *order* would catch, which is why the two swaps are commented in the mark scheme's own solution. **Mark scheme**

- Example solution:
- PROCEDURE Sort()
- DECLARE Temp, J, Boundary : INTEGER
- DECLARE NoSwaps : BOOLEAN
- Boundary <- 1999
- REPEAT
- NoSwaps <- TRUE
- FOR J <- 1 TO Boundary
- IF Reading[J, 1] > Reading[J+1, 1] THEN
- //first swap sensor value
- Temp <- Reading[J, 1]
- Reading[J, 1] <- Reading[J+1, 1]

Question 2

9618/22 J25 ▪ Q2 ▪ 8 marks

A program to calculate the pay of employees working for a company is being designed.

(a) Stepwise refinement has been used in the design of the program.

Describe stepwise refinement. **[2]**

Question 2

Hours worked	Value of sales	Bonus pay
between 1 and 40 inclusive	2000 or less	0
between 1 and 40 inclusive	above 2000	50
above 40	2000 or less	10
above 40	above 2000	100

Question 2

9618/22 J25 ▪ Q2 ▪ 8 marks

A program to calculate the pay of employees working for a company is being designed.

(b)(i) The module is tested using white-box testing.

State **two** tests, using valid data, that can be used to test different paths through the program.

Test one:

Test two: **[2]**

(b)(ii) The program to calculate pay uses a number of modules which are each called from different places in the program.

The program is to be tested using stub testing before all the program modules have been completed.

Describe stub testing. **[2]**

(b)(iii) The program has been completed and compiles successfully.

Question 2

The program is tested using black-box testing.
Identify and describe **one** type of error that black-box testing could detect.
Description **[2]**

Solution 2

(a)

Worked steps

Two marks means two distinct ideas, and the second is the one candidates leave out.

- Stepwise refinement breaks a problem into smaller steps, adding detail at each stage.
- It continues **until each step is simple enough to be written directly as lines of code**.

Saying only "it breaks the problem down" is half the answer; what makes it *stepwise* is the stopping condition.

Mark scheme

- Mark as follows:
- 1. To increase the level of detail of the algorithm // To break the problem / task into smaller steps ...
- 2. ... until steps can be directly translated into lines of code // from which it can be programmed

Solution 2

(b)(i) Mark scheme

- One mark for each of set test values
- Hours worked between 1 and 40 inclusive
- Sales value = 2000
- Bonus Pay: 0
- Hours worked above 40
- Sales value = 2000
- Bonus Pay: 10
- Hours worked between 1 and 40 inclusive

Solution 2

- Sales value 2000
- Bonus Pay: 50
- Hours worked above 40
- Sales value above 2000
- Bonus Pay: 100 max 2

Solution 2

(b)(ii) Worked steps

The technique is testing with **stubs**.

- Write a simple stand-in module for each unfinished one, with the same name and parameters.
- Each stub returns a fixed expected value, or outputs a message to show it was called.
- The finished modules can then be tested in place, because every call they make still returns something sensible — the program runs end to end long before it is complete.

Solution 2

Mark scheme

- One mark per point
- ▪ Simple modules are written to replace each of the unfinished modules.
- ▪ Each simple module will return an expected value / will output a message to show it has been called.

Solution 2

(b)(iii) Mark scheme

- One mark for naming type of error and one mark for corresponding description
- Type of error: Logic (error)
- Description:
- Where the program does not behave as expected / Does not give expected result / An error in the logic of the algorithm
- Or
- Type of error: Run-time (error)
- Description:

Solution 2

- The program performs an illegal operation
- Max 2

Question 1

9618/22 N24 ▪ Q1 ▪ 9 marks

A program has been developed and released for general use. After a few months of use an error is detected where under certain circumstances it outputs an unexpected value.

(a) The error in the program needs to be corrected.

Identify the stage of the program development life cycle that this correction is made in. **[1]**

Question 1

Variable name	Used to store	Data type
`Name`	A customer name.	
`Index`	An array index.	
`Result`	The result of the division of any two non-zero numbers.	

[3]

Question 1

9618/22 N24 ▪ Q1 ▪ 9 marks

A program has been developed and released for general use. After a few months of use an error is detected where under certain circumstances it outputs an unexpected value.

(b) The program contains a function `Lookup()`. After investigation, it is found that this is the function that sometimes returns an incorrect value.

An Integrated Development Environment (IDE) is used to help locate the error.

The IDE features of watch window, single stepping and breakpoint will be used.

Explain these features including the order that they will be used in to locate the error in `Lookup()`. **[3]**

Question 1

9618/22 N24 ▪ Q1 ▪ 9 marks

A program has been developed and released for general use. After a few months of use an error is detected where under certain circumstances it outputs an unexpected value.

(c) To solve the error a programmer decides to create a new module.

The design of the new module has been completed and the module is being coded.

Identify **two** features of an IDE that will help during the coding of this new module.

1. 2. **[2]**

Question 1

9618/22 N24 ▪ Q1 ▪ 9 marks

A program has been developed and released for general use. After a few months of use an error is detected where under certain circumstances it outputs an unexpected value.

(d) The new module referred to in part (c) introduces **three** new variables.

Complete the following table by giving the appropriate data type for each.

Variable name	**Used to store**	**Data type**
`Name`	A customer name.	
`Index`	An array index.	
`Result`	The result of the division of any two non-zero numbers.	

Question 1

[3]

Solution 1

(a)

Mark scheme

- (Corrective) Maintenance

Solution 1

(b) Mark scheme

- For example:
- ▪ Set a breakpoint at the start of / within Lookup, to stop execution at a given statement
- ▪ then use single stepping to execute one statement / instruction at a time
- ▪ to display the value of variables using a watch window
- One mark for each:
- MP1

Solution 1

- Order starting with a breakpoint and an explanation – ‘stop execution at this statement / line’
- MP2
- Explanation of single stepping – execute ‘line by line’ / statements
- MP3
- Explanation of watch window – displaying the value of variable(s)

Solution 1

(c) Worked steps

Read the question's stage carefully: the design is finished and the module is **being coded**. So the features to name are the ones that help while typing, not the ones that help while debugging a program that already runs.
Two marks from:

- The **editor** itself.
- **Auto-complete or auto-correction** of syntax, including identifying undeclared variables.
- **Prettyprint / auto-indentation / structure highlighting**, which makes the block structure visible as you write.

Solution 1

- **Dynamic syntax checking**, flagging an error as the line is typed rather than at compile time.
- **Expanding and collapsing code blocks**, so a long module can be read a section at a time.
- **Context-sensitive prompts**, offering the parameters a routine expects.

△ Breakpoints, single stepping and a variable watch window are IDE features but they belong to **testing and debugging**, which is a different stage. They are the right answer to a differently worded question on the same page, and the wrong answer here. Name the feature and say what it does for the programmer. “It has an editor” is one of the mark points, but ”an editor with syntax highlighting, so mistyped keywords are visible immediately” is the answer that cannot be misread. **Mark scheme**

Solution 1

- Features include:
- MP1
- Editor
- MP2
- Auto- (syntax) complete / auto correction // identify undeclared variables(s)
- MP3
- Prettyprint / auto-indentation / auto (structure) highlighter
- MP4
- Dynamic syntax checking
- MP5

Solution 1

- Expand / collapse code blocks
- MP6
- Context sensitive prompts
- Max 2 marks

Solution 1

(d) Mark scheme

- One mark per row:
- Variable name
- Used to store
- Data type
- Name a customer name
- STRING
- Index an array index
- INTEGER

Solution 1

- Result the result of the division of any two non-zero numbers
- REAL
- 3
- October/November
- 2024

Question 5

9618/22 N24 ▪ Q5 ▪ 7 marks

A program contains a global 1D array `Data` with 100 elements of type `INTEGER`.
The program contains a function `Process()` expressed in pseudocode as follows:
FUNCTION Process(Number : INTEGER, Label : STRING) RETURNS STRING
DECLARE Index, Count : INTEGER DECLARE ReturnValue : STRING
Count <- INT(100 / Number) Index <- Data[Number]
CASE OF (Index MOD 2) 0 : ReturnValue <-
TO$_U$PPER(*RIGHT*(*Label*, *Count*))1 : *ReturnValue* < −"∗∗∗∗∗"*ENDCASE*
RETURN ReturnValue ENDFUNCTION

(a) Run-time errors can be generated in different ways. For example, a run-time error will be generated if a function is called with invalid parameters.
The pseudocode contains three statements that could generate a run-time error.
Write the three statements **and** explain how each could generate a run-time error.

Question 5

Statement 1
Statement 2
Statement 3 **[3]**

Question 5

9618/22 N24 ▪ Q5 ▪ 7 marks

A program contains a global 1D array `Data` with 100 elements of type `INTEGER`.
The program contains a function `Process()` expressed in pseudocode as follows:

```
FUNCTION Process(Number : INTEGER, Label : STRING) RETURNS STRING DECLARE
Index, Count : INTEGER DECLARE ReturnValue : STRING
Count <- INT(100 / Number) Index <- Data[Number]
CASE OF (Index MOD 2) 0 : ReturnValue <-
TO_UPPER(RIGHT(Label, Count))1 : ReturnValue < −"*****"ENDCASE
RETURN ReturnValue ENDFUNCTION
```

(b) One type of run-time error can cause a program to stop responding ('freezing'). Identify a particular type of programming construct that can generate this type of error **and** explain why it occurs. **[2]**

Question 5

9618/22 N24 ▪ Q5 ▪ 7 marks

A program contains a global 1D array `Data` with 100 elements of type `INTEGER`.
The program contains a function `Process()` expressed in pseudocode as follows:

```
FUNCTION Process(Number : INTEGER, Label : STRING) RETURNS STRING DECLARE
Index, Count : INTEGER DECLARE ReturnValue : STRING
Count <- INT(100 / Number) Index <- Data[Number]
CASE OF (Index MOD 2) 0 : ReturnValue <-
TO_UPPER(RIGHT(Label, Count))1 : ReturnValue < −"*****"ENDCASE
RETURN ReturnValue ENDFUNCTION
```

(c) The function `Process()` contains a selection construct using a `CASE` structure.
Write pseudocode using a single selection construct with the same functionality **without** using a `CASE` structure. **[2]**

Solution 5

(a) Worked steps

A run-time error is one the compiler cannot see: the code is legal, but some **value** makes it fail as it runs. Take each statement and ask what value would break it.

- `Count <- INT(100 / Number)` — if `Number` is **zero**, this is a division by zero.
- `Index <- Data[Number]` — if `Number` is outside the array's bounds, this indexes memory the array does not have.
- `ReturnValue <- TO_UPPER(RIGHT(Label, N))` — if `N` is larger than the length of `Label` (or negative), `RIGHT` is asked for characters that are not there.

The pattern in all three: a variable used where only a **restricted range** of values is valid, with nothing checking it first. **Mark scheme**

Solution 5

- MP1
- Count <- INT(100 / Number)
- Number could be zero (giving a divide by zero)
- MP2
- Index <- Data[Number]
- Potential error: Value Number could be outside the range of array indices
- MP3
- ReturnValue <- TO_UPPER(RIGHT(Label, Count))
- Potential Error: Number to extract may be too big / negative / out of range for use in the RIGHT function // Label has insufficient characters

Solution 5

- MP4
- RETURN RetVal
- Potential Error: There is no value to be returned // there is no variable named
- RetVal
- Mark as follows:
- 1 mark for each statement and description
- Max 3 marks

Solution 5

(b)

Worked steps

- Construct: a **conditional loop** — pre-condition (WHILE) or post-condition (REPEAT).
- Explanation: if nothing inside the loop can ever make the terminating condition true, the loop never ends, so the program stops responding.

A count-controlled FOR loop cannot do this, because its number of iterations is fixed before it starts — which is exactly why the answer must name a conditional one.

Mark scheme

- MP1
- Construct: A (pre/post) conditional loop
- MP2
- Explanation: The terminating condition is never satisfied

Solution 5

(c) Worked steps

A two-branch CASE is just an IF, so rewrite it as one — and keep both branches, because a missing ELSE leaves `ReturnValue` undefined.

```
IF Index MOD 2 = 0 THEN
    ReturnValue <- TO_UPPER(RIGHT(Label, Count))
ELSE
    ReturnValue <- "****"
ENDIF
```

Solution 5

- `IF … THEN … ELSE … ENDIF` with the condition tested the same way round as the CASE had it.
- The **same** assignments in the same branches: the CASE's matched value becomes the THEN, its OTHERWISE becomes the ELSE.

Solution 5

`MOD 2 = 0` is the test for an even index — writing `= 1` swaps the branches, which is the easy way to lose the second mark. **Mark scheme**

- Example solution:
- IF Index Mod 2 = 0 THEN
- ReturnValue <- TO_UPPER(RIGHT(Label, Count))
- ELSE
- ReturnValue <- "****"
- ENDIF
- Mark as follows:
- MP1
- IF...THEN...ELSE...ENDIF
- MP2
- Both correct assignments and the correct test/logic
- 2
- October/November

Question 5

9618/21 J24 ▪ Q5 ▪ 6 marks

5 A global 1D array of strings contains three elements which are assigned values as shown:

```
Data[1] ← "aaaaaa"
Data[2] ← "bbbbbb"
Data[3] ← "cccccc"
```

Procedure `Process()` manipulates the values in the array.

The procedure is written in pseudocode as follows:

```
PROCEDURE Process(Format : STRING)
   DECLARE Count, Index, L : INTEGER
   DECLARE Result : STRING
   DECLARE C : CHAR

   Result ← "****"
```

Question 5

```
    FOR Count ← 1 TO LENGTH(Format) STEP 2
        C ← MID(Format, Count, 1)
        L ← STR_TO_NUM(MID(Format, Count + 1, 1))

        Index ← (Count + 1) DIV 2

        CASE OF C
            'X' : Result ← TO_UPPER(Data[Index])
            'Y' : Result ← TO_LOWER(Data[Index])
            'Z' : Result ← "**" & Data[Index]
        ENDCASE

        Data[Index] ← LEFT(Result, L)
    NEXT Count

ENDPROCEDURE
```

Question 5

(a) Complete the trace table by dry running the procedure when it is called as follows:

```
CALL Process("X3Y2W4")
```

Count	C	L	Index	Result	Data[1]	Data[2]	Data[3]

Question 5

Question 5

[6]

(b) The procedure is to be modified. If variable `C` is assigned a value other than `'X'`, `'Y'` or `'Z'`, then procedure `Error()` is called and passed the value of variable `C` as a parameter.

Question 5

This modification can be implemented by adding a **single line** of pseudocode.

(i) Write the single line of pseudocode.

Question 5

(ii) State where this new line should be placed.

Solution 5

(a)

Mark scheme

- One mark per zone.
- 6

Solution 5

(b)(i)

Mark scheme

- OTHERWISE : CALL Error(C)
- 1

Solution 5

(b)(ii) Worked steps

- The `OTHERWISE` clause goes **after the last labelled clause** — after the `'Z'` clause — and **before** `ENDCASE`.

That position is forced by how a `CASE` is evaluated: the labels are tested **in order**, and the first match wins. `OTHERWISE` is the catch-all, so it can only be correct in the one place where every specific label has already been tried.

Solution 5

Put it earlier and it swallows the cases below it — every value would match it, and the remaining clauses would be unreachable. Most languages will not warn about that; the program simply always takes the catch-all branch.

Solution 5

The same rule appears in every language with this construct: `default` in a Java or C `switch`, `else` at the end of an `if … elif` chain, `OTHERWISE` here. In each, the general case is written last because it is tested last.

Mark scheme

- After the 'Z' clause in the CASE construct // before the ENDCASE
- 1
- 9618/21
- May/June 2024

Question 5

9618/22 J24 ▪ Q5 ▪ 6 marks

5 A program is being designed in pseudocode.

The program contains a global 1D array `Data` of type string containing 200 elements.

The first element has the index value 1.

A procedure `Process()` is written to initialise the values in the array:

```
PROCEDURE Process(Label : STRING)
   DECLARE Index : INTEGER
   Index ← 0
   INPUT Data[Index]
   WHILE Index < 200
      Index ← Index + 1
      CASE OF (Index MOD 2)
         0 : Data[Index] ← TO_UPPER(Label)
         1 : Data[Index] ← TO_LOWER(Label)
```

Question 5

```
            OTHERWISE : OUTPUT "Alarm 1201"
        ENDCASE
    NEXT Index
    OUTPUT "Completed " & Index & " times"
ENDPROCEDURE
```

(a) **(i)** The pseudocode contains **two** syntax errors and **one** other error.

Identify the errors.

Question 5

[3]

(ii) The procedure contains a statement that is **not** needed.

Identify the pseudocode statement **and** explain why it is **not** needed.

Question 5

[2]

(b) After correcting all syntax errors, the pseudocode is translated into program code which compiles without generating any errors.

When the program is executed it unexpectedly stops responding.

Identify the type of error that has occurred.

Solution 5

(a)(i) Worked steps

Read the fragment for three separate faults, not three symptoms of one.

1. **Mismatched loop keywords** — a `WHILE` loop closed with `NEXT Index`. A WHILE ends with `ENDWHILE`; `NEXT` belongs to a FOR.
2. **`&` used to join a number to a string** — concatenation works on strings, so an integer must be converted first with `NUM_TO_STR`.
3. **The array is indexed outside its bounds** — the loop reaches an index the array does not have (element 0, or one past the end).

The first two are **syntax** errors: the compiler rejects them before the program runs. The third is not — that code is perfectly legal. **Mark scheme**

Solution 5

- One mark per error:
- Syntax:
- 1. NEXT Index (should be ENDWHILE)
- 2. '&' used to concatenate an integer (in OUTPUT statement)
- Other:
- 3.
- Accesses element outside range // Accesses element 0/3

Solution 5

(a)(ii) Mark scheme

- One mark per point:
- Statement:
- ▪
- The OTHERWISE statement
- Explanation:
- ▪
- The result of MOD 2 can only be 0 or 1/2

Solution 5

(b)

Worked steps

- A **run-time** error.

That is the point of the question: the out-of-bounds index only becomes a fault when a particular value is reached while the program executes, which is exactly what distinguishes a run-time error from a syntax one.

Mark scheme

- Run-time
- 1
- 9618/22
- May/June 2024