

Cambridge AS & A Level · Computer Science 9618 · Notes

Software development AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/software-development

Last checked 5 October 2026

Read first: [Algorithm design and problem-solving](#), [Programming](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This topic is examined in Paper 2. By the end of it you should be able to:

1. **Explain why a program development life cycle is used:** it splits the work of making a program into clear stages.
2. **Explain why there are different life cycles**, and choose one to suit the program being built: the **waterfall**, **iterative** and **rapid application development (RAD)** models.
3. **Describe the principles, benefits and drawbacks** of each of these three models.
4. **Describe the five stages:** analysis, design, coding, testing and maintenance, and say what happens (and what is produced) in each.
5. **Use a structure chart** to break a problem into modules and show the parameters passed between them: say what a structure chart is for, draw one for a given problem, and write the matching pseudocode from one.
6. **Explain what a state-transition diagram is for:** documenting an algorithm by showing its states and the inputs that move it between them.
7. **Explain how faults in programs are exposed and avoided.**
8. **Find and name the three kinds of error** (syntax, logic and run-time) and **correct them**.
9. **Describe the testing methods:** dry run, walkthrough, white-box, black-box, integration, alpha, beta, acceptance and stub testing, and choose suitable test data for each.
10. **Explain why a test strategy and a test plan are needed**, and what they contain.
11. **Choose test data for a test plan:** normal, abnormal, extreme and boundary data.
12. **Explain why a program needs maintenance after release**, and tell apart **corrective**, **adaptive** and **perfective** maintenance.
13. **Read an existing program and change it** to add to what it does.

Structure charts and state-transition diagrams are also used in the *Algorithm design and problem-solving* note, and trace tables and the three error types in the *Programming* note. This note recaps them and adds what this topic asks for.

Understand it

Why a life cycle?

A program that a customer will rely on is too big to write in one go. A **program development life cycle** splits the job into stages, each with its own aim and its own documents. That gives the project a plan and deadlines, lets a team share the work, makes it easy to see how far the project has got, and means problems are found early, when they are cheap to fix.

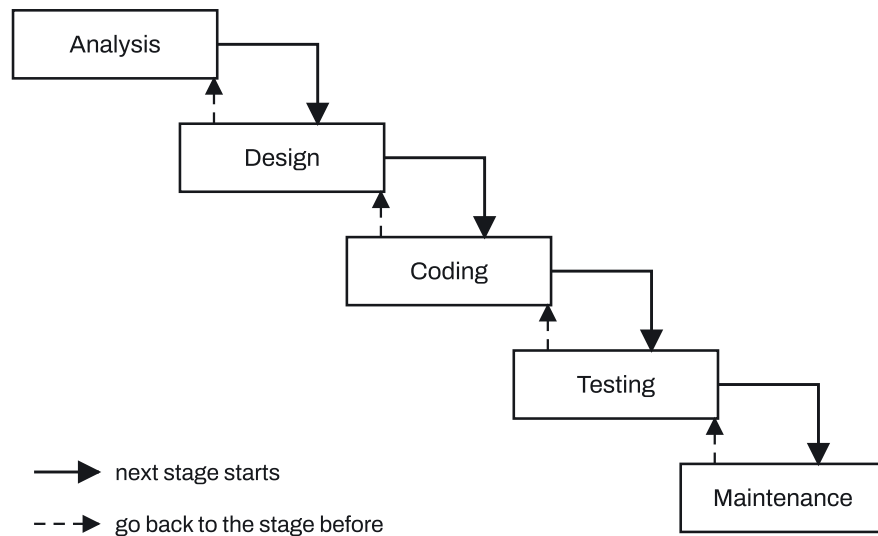
The five stages

Stage	What happens	Typical activities and documents
Analysis	Find out exactly what the program must do	Interview, observe or survey the client and users; study the current system and its problems; feasibility study; the requirements specification (the agreed list of what the program must do)
Design	Decide how the program will meet the requirements	Decomposition and structure charts ; algorithms (pseudocode, flowcharts, state-transition diagrams); data structures, file and record layouts, identifier tables; screen and report layouts; validation rules; the test plan
Coding	Write the program in a chosen programming language	Turn the algorithms into code, often in an IDE (auto-complete, syntax checking); compile it; fix syntax errors; test each module as it is written
Testing	Check the program works and meets the requirements	Dry runs and trace tables, walkthroughs, white-box and black-box tests, integration, alpha, beta and acceptance testing
Maintenance	Keep the program useful after it has been released	Fix faults users report; change it for new requirements, new hardware or new laws; make it faster or easier to use

Exam questions often describe an activity and ask which stage it belongs to. Ask yourself: *is this finding out what is needed* (analysis), *planning how* (design), *writing code* (coding), *checking it works before release* (testing) or *changing it after release* (maintenance)?

The waterfall model

In the **waterfall** model the stages happen one after another, like water falling down steps. Each stage is finished, and its documents signed off, before the next one starts.



- The **downward arrows** mean the output of each stage (for example the design documents) is handed on and becomes the starting point of the next.
- The **upward arrows** mean that if a problem is found, the team can go back to the stage before to fix it, then work down again.

Benefits: simple to understand and manage; each stage has clear deliverables and a deadline; progress is easy to measure; it suits projects whose requirements are clear, fixed and agreed at the start; the documentation is thorough.

Drawbacks: the client sees no working program until late on; a change in requirements, or a mistake found late, means going back up the steps, which is slow and costly; it suits badly a project whose requirements are unclear or likely to change.

The iterative model

In the **iterative** model the program is built in a series of cycles. The first cycle produces a small working version with the core features; each later cycle goes round analysis, design, coding and testing again to add or improve features, until the whole program is done.

Benefits: a working version exists early; the client can give feedback after each cycle, so changing requirements can be handled; problems are found sooner and each cycle is smaller and easier to test; the riskiest parts can be built first.

Drawbacks: the whole system has to be planned well enough at the start to be split into cycles; it needs more client involvement and more management; the total cost and finish date are harder to fix in advance; later cycles may expose problems in the overall design.

Rapid application development (RAD)

RAD puts speed and the user first. Instead of long analysis and design, the developers build **prototypes** (quick working models) and show them to the client again and again, changing them after each round of feedback. Different modules are often built in parallel by small teams, each against its own deadline (a "time box"), and existing code and library modules are reused wherever possible.

Benefits: the program is produced quickly; the client is involved all the way through, so the final program is closer to what they want; requirements can change during development; early prototypes catch misunderstandings early.

Drawbacks: it needs a client who can give feedback often; it needs skilled developers and suits only programs that can be split into modules; there is less documentation, so the program can be harder to maintain; speed can lead to weaker overall design, and it is harder to manage on a large project.

Choosing a model

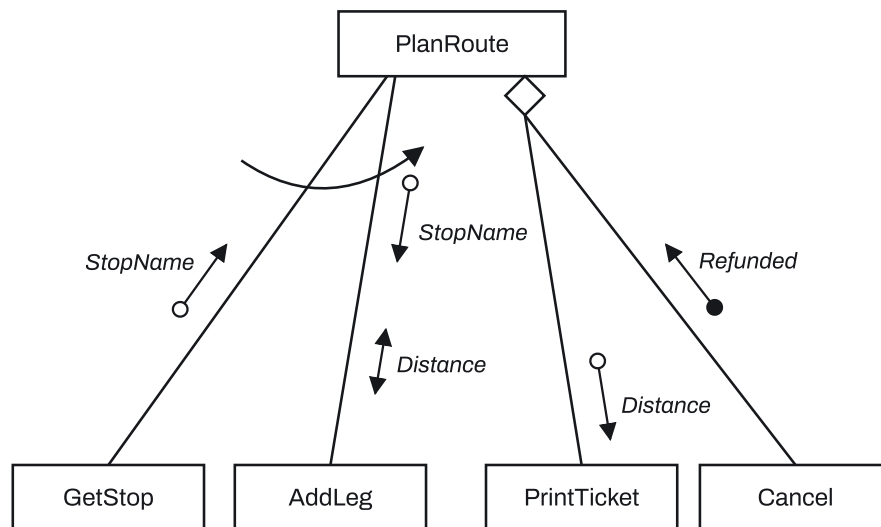
Different programs need different life cycles because projects differ in **how clear and fixed the requirements are**, how big and complex the program is, how fast it is needed, how much the client can be involved, the size and skill of the team, and the budget.

- Requirements clear and fixed, a safety-critical or contract-based job: **waterfall**.
- A large program that can be released in parts, with requirements that will grow: **iterative**.
- Needed fast, requirements unclear or changing, client available to test prototypes: **RAD**.

Structure charts

A **structure chart** shows how a problem has been broken into modules (decomposition), which module calls which, and the data passed between them. It is drawn during **design**. The full set of symbols is in the *Algorithm design and problem-solving* note; in short:

- each box is a module; a line down from a box to another means "calls"; modules are called from **left to right**;
- an arrow with an **open circle** is data: down for a parameter, up for a returned value;
- an arrow with a **filled circle** is a flag (a BOOLEAN value);
- a **two-headed arrow** is a parameter passed **by reference** (BYREF): the value goes down and the changed value comes back;
- a **diamond** where the lines leave a box: the module calls **one** of those modules, chosen at run time (selection);
- a **curved arrow** across lines: the module calls those modules **repeatedly**, in order from left to right (iteration).



From a chart to pseudocode headers. Read the arrows on each line:

- an arrow **up** means the module returns a value, so it is a FUNCTION ... RETURNS <type>;
- no arrow up means a PROCEDURE;
- each arrow **down** is a parameter in the brackets, with its type;
- a **two-headed** arrow is a parameter with BYREF in front of it.

For the chart above:

```
FUNCTION GetStop() RETURNS STRING
PROCEDURE AddLeg(StopName : STRING, BYREF Distance : REAL)
PROCEDURE PrintTicket(Distance : REAL)
FUNCTION Cancel() RETURNS BOOLEAN
```

From a chart to the calling module's pseudocode. The curved arrow becomes a loop with the calls inside it, in left-to-right order. The diamond becomes an IF (or CASE) that calls one module or the other. A function call goes on the right of an assignment; a procedure is used with CALL. Worked example 3 writes PlanRoute this way.

A structure chart shows **what** is called and **what data** is passed, but not the details of **how** each module works, or the exact conditions: those come from the program notes or the pseudocode.

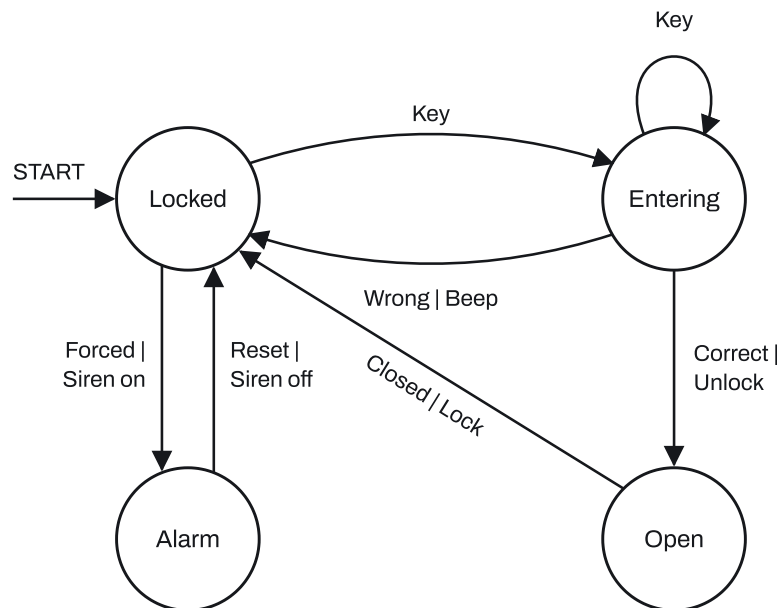
State-transition diagrams

A **state-transition diagram** documents an algorithm (often one that controls a device) as a set of **states** and the **events** (inputs) that move it from one state to another. It is drawn during design, and it shows clearly how the system must react to every input in every state, so nothing is forgotten.

- Each **state** is a circle with its name.
- Each **transition** is an arrow from the old state to the new one, labelled with the input that causes it, then a bar, then any output it produces. With no output the label is just the input.

- An arrow that leaves a state and loops back to the **same** state is an input that changes nothing.
- An arrow from nowhere (often labelled START) points to the starting state. A **halting state** has no arrows leaving it.

Here is a keypad door lock. The inputs are *Key* (a digit is pressed), *Correct* and *Wrong* (the code is checked), *Closed* (the door shuts), *Forced* (the door is pushed open while locked) and *Reset* (a guard resets the alarm). The outputs are *Beep*, *Unlock*, *Lock*, *Siren on* and *Siren off*.



The same diagram as a **state-transition table**, one row per arrow:

Current state	Input	Output	Next state
Locked	Key	(none)	Entering
Entering	Key	(none)	Entering
Entering	Wrong	Beep	Locked
Entering	Correct	Unlock	Open
Open	Closed	Lock	Locked
Locked	Forced	Siren on	Alarm
Alarm	Reset	Siren off	Locked

It has 7 transitions; 6 of them change the state and 5 have an output. To follow a sequence of inputs, start in the current state and for each input follow the arrow with that label. Starting in Locked, the inputs Key, Key, Correct, Closed give the outputs Unlock then Lock, and the lock ends in Locked.

Three kinds of error

- A **syntax error** breaks the grammar rules of the programming language, such as a misspelt keyword, a missing `ENDIF` or a bracket that is never closed. The compiler or interpreter finds it and the program can't be translated until it is fixed.
- A **logic error** is a mistake in the algorithm itself: the program runs but gives the wrong result, such as `>` used where `>=` is needed, the wrong variable added to a total, or decisions in the wrong order. The compiler **can't** find it, because every statement is valid; only testing shows it.
- A **run-time error** happens while the program is running: it tries to do something impossible, so it crashes or stops responding. Examples: dividing by zero, an array index outside the bounds, converting a string that isn't a number with `STR_TO_NUM`, reading past the end of a file, or a loop whose ending condition is never met (an infinite loop, which makes the program freeze).

Correcting an error means giving the fixed statement, not just describing the fault: "change `IF Floor > -2` to `IF Floor >= -2`".

Exposing and avoiding faults

Faults are **avoided** by good design and good habits, and **exposed** by testing.

- **Avoid them:** a clear requirements specification; decomposition into small modules that each do one job and can be tested on their own; **library routines**, which are already tried and tested and give you code that would be hard to write yourself; **constants** for fixed values, so the value can't be changed by accident, is written in only one place (one edit updates it everywhere) and has a name that explains it; meaningful identifiers, comments and indentation; local variables instead of global ones; validating every input.
- **Expose them:** dry runs with trace tables, walkthroughs, and testing with planned test data; debugging tools in an IDE.

Finding errors with an IDE

An **integrated development environment (IDE)** helps while you code and while you debug.

- **While coding:** an editor with **auto-complete** of keywords and identifiers, **dynamic syntax checking** (errors underlined as you type), **pretty-printing** (automatic indentation and colouring of keywords), **expanding and collapsing** blocks of code, and **context-sensitive prompts** (for example, the parameters a function needs).
- **While debugging a logic error**, use them in this order:
 1. set a **breakpoint** on a statement just before the part you suspect: the program runs normally and pauses when it reaches that line;
 2. **single-step**: run one statement at a time from there;
 3. watch the variables in the **watch window** (or report window), which shows their current values after each step, until one changes to a value you didn't expect. That statement is where the error is.

- **Commenting out** a statement (putting the comment symbol in front of it) makes the compiler ignore it without deleting it.

Testing methods

Method	What it is
Dry run	Work through the code by hand, recording each variable's value in a trace table , to check it does what it should
Walkthrough	The programmer takes others (or a reviewer) through the code or algorithm line by line, usually with test data and a trace table; errors show up as an unexpected value or an unexpected path. It needs the program listing or algorithm
White-box	Testing with knowledge of the code: test data is chosen so that every path through the code is used, and the result of each path is checked
Black-box	Testing without looking at the code : put in test data and compare the actual output with the expected output from the specification
Stub	A module that isn't written yet (or doesn't work) is replaced by a simple dummy module (a stub) that returns a fixed, known value or outputs a message such as "MoveTo called", so the rest of the program can be tested
Integration	Modules that have each been tested on their own are combined, one at a time, and tested together to check they work with each other
Alpha	Testing by the developers (or people in the developer's company) near the end of development, before anyone outside sees it
Beta	A nearly finished version is given to a group of real users outside the company, who use it normally and report faults before the full release
Acceptance	The client tests the finished program against the requirements specification to agree that it does what they asked for; the last stage before it is handed over

The usual order is: each module tested as it is written (dry run, white-box; stubs where needed), then **integration**, then **alpha**, then **beta** (for software sold to many users) or **acceptance** (for a program made for one client).

Test strategy, test plan and test data

A **test strategy** is the overall approach to testing: which methods will be used at which stage, who will test, and what counts as passing. A **test plan** lists every individual test so that the program is tested thoroughly and nothing is forgotten, and so tests can be repeated after a change. Each row of a test plan gives:

- a test number and **what is being tested** (its purpose);
- the **test data**, and its type;
- the **expected result**;

- (when the test is run) the **actual result**, and whether it passed or what action was taken.

Test data types, for a value that must be a whole number from -2 to 14 inclusive:

Type	Meaning	Examples	Expected
Normal	Ordinary values well inside the range	3, 10	accepted
Extreme	The largest and smallest values that are still valid	-2 and 14	accepted
Boundary	Each edge of the range tested from both sides: the last valid value and the first invalid one	-2 and -3 ; 14 and 15	-2 , 14 accepted; -3 , 15 rejected
Abnormal (erroneous)	Values that should be rejected	40 , -9	rejected

If the question says the data is always an **integer**, abnormal test data must still be an integer (one outside the range), not text or a decimal. Each test in a plan should test something different: two normal values from the middle of the range add nothing.

Maintenance

A program in use still needs changes, because faults appear, the world around it changes and users want more.

- **Corrective** maintenance fixes errors (bugs) found after release, for example a calculation that gives a wrong answer in a rare case.
- **Adaptive** maintenance changes the program because its **requirements or environment** have changed: new hardware or a new operating system, a new law or tax rule, a change to the library modules or other software it uses, or new tasks the client now needs.
- **Perfective** maintenance improves a program that already works correctly: making it faster, using less memory, or easier to use, without changing what it is meant to do.

Changing an existing program

To add to what a program does, first read it until you can say what each line does (a quick dry run helps). Then make the smallest change that does the job, and say where it goes: "replace line 21 with ...", "insert after line 16 ...", "move line 30 to just before line 24". Typical changes: an extra CASE clause or an OTHERWISE, two CASE clauses combined into one, a CASE rewritten as IF ... ELSE, a conditional loop replaced by a count-controlled one, a new parameter or output. Afterwards, test again with the old test data as well as new data, to check nothing that worked has broken.

Key facts and formulas

There is no formula list for 9618. Learn these facts.

Formula	Status
Life-cycle stages in order: analysis, design, coding, testing, maintenance	Learn it
Analysis produces the requirements specification; design produces structure charts, algorithms, data structures, interface layouts and the test plan	Learn it
Waterfall: stages done once each, in order, each finished before the next; down arrows pass each stage's output on, up arrows return to the previous stage	Learn it
Iterative: the program is built in cycles, each adding features to a working version	Learn it
RAD: quick prototypes, repeated client feedback, modules built in parallel to time limits, reuse of code	Learn it
Structure chart: curved arrow means repeated calls; diamond means one call chosen at run time; two-headed arrow means a BYREF parameter	Learn it
Structure chart arrows: down is a parameter, up is a returned value (FUNCTION); filled circle is a BOOLEAN flag	Learn it
State-transition arrow label: the input, then a bar, then the output (input alone if there is no output)	Learn it
Syntax error: grammar rules broken, found by the compiler; logic error: runs but wrong result; run-time error: illegal operation while running (crash or freeze)	Learn it
Test data: normal (well inside), extreme (largest and smallest valid), boundary (each edge from both sides), abnormal (should be rejected)	Learn it
White-box: every path through the code; black-box: inputs and outputs only, code not seen	Learn it
Stub: dummy module that returns a known value or outputs a message in place of an unfinished module	Learn it
Integration: tested modules combined and tested together; alpha: by the developers; beta: by real users before release; acceptance: by the client against the requirements	Learn it
Maintenance: corrective fixes errors, adaptive meets new requirements or environments, perfective improves performance or usability	Learn it
IDE debugging order: breakpoint, then single-stepping, while a watch window shows the variables	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (70 questions, 2021–2025), the types came up this often. 44 of the 70 questions mix types, so the counts add up to more than 70.

Question type	Questions
Identify, explain and correct errors (syntax, logic, run-time)	14
Name and compare life-cycle stages and models	13
Identify or describe a testing method	12
Draw or complete a structure chart	11
Read a structure chart: explain symbols, write module headers	11
Complete a state-transition diagram or its table	10
Dry run code with a trace table	10
Choose test data or complete a test plan	9
Amend or improve existing pseudocode	8
Name and justify types of maintenance	6
Use IDE features to debug or code	5
Explain ways of avoiding faults	5
Decompose a problem into sub-modules	4

1. Identify, explain and correct errors (syntax, logic, run-time) (14 questions)

- Naming an error type:** give the name and a description that fits it: syntax (rules of the language broken, so it won't compile), logic (runs, but the result is wrong), run-time (an illegal operation while running, so it crashes or freezes). If asked for "two other types", leave out the one already named in the question.
- Finding errors in given pseudocode:** check every operator (> or <, & or +, ← or =), every starting value, every loop condition and every function's arguments. For each error, quote the statement and write the corrected statement.
- Statements that could cause a run-time error:** look for a division (the divisor could be 0), an array index that comes from a parameter or a calculation (it could be outside the bounds), a string function whose position or length could be too big or negative, a conversion of a string that might not be numeric, and a variable used before it has a value. Quote the statement and say which value makes it fail.
- Freezing:** a conditional loop (WHILE or REPEAT) whose ending condition can never become true.
- Logic errors in a flowchart or in decisions:** try one value from each range. If a range never reaches its result, the decisions are in the wrong order or a test is wrong: say which decision to swap or change.

Variants:

- "Why doesn't the compiler find it?"** Because every statement is valid syntax; the compiler checks grammar, not what the program is meant to do.
- A design error with BYREF:** a procedure that changes a parameter passed by reference also changes the caller's variable (for example a loop counter), so the calling loop goes wrong. The fix is BYVAL, or a local copy.

- **"Give test data that shows the error":** choose values that take the faulty path, and say what wrong output they produce.

2. Name and compare life-cycle stages and models (13 questions)

1. **Activity to stage:** decide whether the activity finds out what is needed (analysis), plans how (design), writes or compiles code (coding), checks the program before release (testing) or changes it after release (maintenance). One mark per row.
2. **Documents:** analysis gives the requirements specification (also interview notes, survey results, a feasibility study); design gives structure charts, algorithms, data structures, identifier tables, interface designs and the test plan.
3. **Why different models:** the choice depends on the project: how clear the requirements are, how fast the program is needed, its size and complexity, the team's skills and size, the budget, and how involved the client can be.
4. **Why waterfall doesn't suit a scenario:** look for clues in the question: requirements that will change, a need to be first or fast, a client who needs to see it working early. Waterfall fixes requirements at the start and produces a working program only at the end, and going back is slow. Suggest RAD (or iterative).
5. **RAD questions:** principle (prototypes improved through client feedback; parallel development of modules); benefits (fast, client involved, changing requirements handled); drawback (needs constant client involvement and skilled developers, less documentation, may scale badly to big projects).
6. **Completing the waterfall diagram:** the five stages in order; down arrows mean each stage's output feeds the next, up arrows mean going back to the previous stage to correct a problem.

3. Identify or describe a testing method (12 questions)

1. **Matching descriptions to methods:** "as modules are combined" is integration; "without looking at the code" is black-box; "every path through the code" is white-box; "by the client" is acceptance; "by users outside the company before release" is beta; "by the developers" is alpha.
2. **Stub testing** (the favourite): a dummy module replaces each unfinished or faulty module; it does nothing complicated but returns a known value, or outputs a message saying it was called, so testing of the rest of the program can continue. Both points are needed.
3. **Walkthrough:** go through the code a line at a time, with test data, recording values in a trace table; an unexpected value or path shows an error. It needs the program listing or algorithm (and the test data with expected results).
4. **White-box test data:** pick one set of values for each path through the code, with the expected result of each.

4. Draw or complete a structure chart (11 questions)

1. **Boxes** in the right hierarchy: each module directly below the one that calls it, called modules in left-to-right order.

2. **Selection:** "calls either A or B" or "decided at run time" means a diamond where the lines leave the calling box.
3. **Iteration:** "repeatedly calls" means a curved arrow across the lines to the modules that are repeated.
4. **Arrows from each header or note:** each parameter is an arrow down beside the line; a returned value is an arrow up; a BOOLEAN is a flag (filled circle); a BYREF parameter is one two-headed arrow, not a down arrow plus an up arrow. Label every arrow with its identifier.

5. Read a structure chart: explain symbols, write module headers (11 questions)

1. **Curved arrow:** say that the module **repeatedly** calls the named modules, and name them **in order** ("PlanRoute repeatedly calls GetStop and then AddLeg"). Saying just "a loop" loses the second mark.
2. **Diamond:** the module calls **one** of the named modules, which one being decided when the program runs.
3. **Header for each module:** arrow up means `FUNCTION Name (...) RETURNS <type>;` otherwise `PROCEDURE Name (...)`. Each down arrow is a parameter with its type; a two-headed arrow gets BYREF. Take the types from the question.
4. **Variants:** which modules are functions (those with a value going up); the effect of BYVAL (the module gets a copy, the caller's variable can't change) and BYREF (the module works on the caller's variable, so changes are kept); the purpose of a structure chart (to show the modules, their hierarchy and the data passed between them).

6. Complete a state-transition diagram or its table (10 questions)

1. **States first:** one labelled circle per state, and the START arrow into the starting state.
2. **One arrow per table row**, from current state to next state, labelled input then bar then output (input alone if there is no output). A row whose next state is the same as its current state is a loop back to that circle.
3. **Labels close to their arrows**, and both inputs and outputs on them; tick off each row as you draw it.
4. **Reading a diagram:** follow the inputs one at a time from the given state, writing down each output and the new state. Counting questions: transitions that change state (arrows between different circles), transitions with outputs, and the halting state (no arrows out).
5. **From a word description:** list the states and every event first, as a table, then draw.

7. Dry run code with a trace table (10 questions)

1. Write the starting values in the first row.
2. Go through the code one statement at a time. **Start a new row only when a value changes** (or when the question's first row shows a different pattern; copy that pattern).
3. For loops, show the loop variable each time round, and the final value that makes the loop stop.
4. Check MOD, DIV, string functions (MID, LEFT, RIGHT, LENGTH, TO_UPPER) and array indices carefully; marks are given per block of the table, so one slip costs only that block.

8. Choose test data or complete a test plan (9 questions)

1. Find the valid range from the question, and whether the values are whole numbers.
2. Give a **normal** value well inside it, both **extreme** values (lowest and highest valid), the **boundary** values just outside each end, and an **abnormal** value, each with its expected result (accepted or rejected, or the value returned).
3. Every row needs the type, the value and the outcome to earn its mark. Don't repeat a type the question already filled in, and don't use text or decimals if the question says integers.
4. **Variants**: one test for each possible return value of a function; one set of values per path for white-box testing; a single string that breaks one rule at a time for black-box testing of a validation routine.

9. Amend or improve existing pseudocode (8 questions)

1. Work out what the existing code does before changing it.
2. **Combine CASE clauses** that do the same thing: a range 1 TO 2 : and, if the clauses differ only by a number equal to the case value, use the variable itself ($\text{Index} \leftarrow \text{Index} + \text{CaseValue}$).
3. **CASE to IF**: CASE OF X MOD 2 with clauses 0 and 1 becomes IF X MOD 2 = 0 THEN ... ELSE ... ENDIF.
4. **A better loop**: a count-controlled FOR when the number of repetitions is known before the loop starts (such as LENGTH(S) characters).
5. Say exactly where the change goes (which line is replaced, or the new position of a moved line).

10. Name and justify types of maintenance (6 questions)

1. Name the type and give a reason from the scenario: corrective (a fault is found after release), adaptive (new requirements, new hardware or operating system, a new law, changed library modules), perfective (faster response, better interface, less memory).
2. "Improve the speed" is always **perfective**; "an error reported after release" is **corrective**. Reasons for adaptive maintenance must be specific: "a new law changes the tax rate", not "things change".

11. Use IDE features to debug or code (5 questions)

1. **Debugging order**: set a **breakpoint** where the suspect code starts (execution stops there), then **single-step** one statement at a time, while the **watch window** (or report window) shows the variables' values; the line where a value goes wrong is the error. Explain each feature; "a breakpoint is where the program breaks" earns nothing.
2. **Coding features**: auto-complete, dynamic syntax checking, pretty-printing, expanding and collapsing code blocks, context-sensitive prompts.
3. An IDE can't fix a logic error by itself: it only helps the programmer find it.

12. Explain ways of avoiding faults (5 questions)

1. **Constants:** the value can't be changed by mistake while the program runs; changing it needs an edit in one place only; the name makes the code clearer. Name the value from the code that should be a constant.
2. **Library routines:** already tried and tested, so likely to be free of errors; they provide code that would be hard to write yourself.
3. **Modules** (procedures and functions): code written and tested once and reused, so it isn't repeated in several places where copies could differ.
4. **Good practice:** meaningful identifiers, comments, indentation, local variables, validation of inputs.

13. Decompose a problem into sub-modules (4 questions)

1. Read the scenario and list its separate jobs in order (get the input, look something up, calculate, update a store, output).
2. Give each job a module name and a one-line use that mentions the scenario, for example `GetMember ( )`: read the member number from the card.
3. **Benefits of decomposition:** each module is smaller, so easier to write, test and debug; modules can be shared among a team; modules can be reused.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A small bakery wants an app so customers can order bread in advance. The owner isn't sure yet what the app should do, expects to change her mind once she sees it, and wants something working within a few weeks.

(a) Name the life-cycle stage for each activity. **[4]**

Activity	Stage
The developer watches staff taking orders over the phone	
The layout of each screen and the order record are planned	
The developer fixes the errors reported by the compiler	
The app is changed because a new law says allergy information must be shown	

(b) Explain why RAD is more suitable than the waterfall model for this app. **[3]**

(c) A year later, the app is changed so its pages load faster. Name this type of maintenance, and name the type in the last row of the table. **[2]**

(d) Before the app goes live, the owner checks it against what she asked for. Identify this test stage. **[1]**

(a) One mark per row: **analysis** (finding out how orders are taken now) **B1**; **design** (planning screens and records) **B1**; **coding** (fixing compiler errors while writing the program) **B1**; **maintenance** (changing a released program) **B1**.

(b) The owner's requirements are unclear and will change; waterfall fixes the requirements at the start and is slow to go back up the stages. **B1** RAD builds prototypes that the owner sees and comments on, so changes are made as she goes. **B1** RAD produces a working app quickly, which the owner wants within weeks, whereas waterfall gives a working program only at the end. **B1**

(c) Faster pages: **perfective B1**. A new law: **adaptive B1**.

(d) **Acceptance** testing. **B1**

Example 2

A lift serves floors –2 to 14. A function checks a requested floor:

```

10 FUNCTION ValidFloor(Floor : INTEGER) RETURNS BOOLEAN
11     IF Floor > -2 AND Floor <= 14 THEN
12         RETURN TRUE
13     ELSE
14         RETURN FALSE
15     ENDIF
16 ENDFUNCTION

```

(a) Complete a test plan of five tests for the function. One normal test is: 6, expected TRUE. **[4]**

(b) One test in your plan fails. State which, name the type of error and give the corrected line. **[3]**

(c) Explain why the compiler did not report this error. **[1]**

(d) The main lift program calls a procedure `MoveTo ( )` that has not been written yet. Describe how the main program can still be tested. **[3]**

(a) One mark per complete row (type, value and expected result):

Type	Test data	Expected result
Normal	6	TRUE
Extreme	–2	TRUE
Extreme	14	TRUE
Boundary	–3	FALSE

Type	Test data	Expected result
Boundary	15	FALSE
Abnormal	30	FALSE

B1 B1 B1 B1 (any four rows after the normal one).

(b) The test with -2 fails: the function returns FALSE **B1**. It is a **logic error B1**. Line 11 should be `IF Floor >= -2 AND Floor <= 14 THEN B1`.

(c) The statement is valid syntax; the compiler checks the grammar, not whether the program does what was intended. **B1**

(d) **Stub testing B1**: a simple dummy `MoveTo ( )` is written in place of the real one **B1**, which only outputs a message such as "MoveTo called with 6" (or returns a fixed known value), so the calls to it can be checked and testing of the rest continues **B1**.

Example 3

The structure chart for the route planner is the second diagram in "Understand it". `StopName` is a `STRING`, `Distance` is `REAL` and `Refunded` is `BOOLEAN`.

(a) Explain the meaning of the curved arrow and of the diamond in this chart. [4]

(b) Write the pseudocode header for each of the four lower modules. [4]

(c) `PlanRoute ( )` keeps calling its first two modules until `GetStop ( )` returns "END". It then prints a ticket if `Distance` is more than 0, and otherwise cancels. Write pseudocode for `PlanRoute ( )`. [5]

(a) The curved arrow means `PlanRoute` calls modules **repeatedly B1**: each time, it calls `GetStop` and then `AddLeg`, in that order **B1**. The diamond means `PlanRoute` calls **only one** of `PrintTicket` and `Cancel B1`, which one being decided when the program runs **B1**.

(b)

```
FUNCTION GetStop() RETURNS STRING
PROCEDURE AddLeg(StopName : STRING, BYREF Distance : REAL)
PROCEDURE PrintTicket(Distance : REAL)
FUNCTION Cancel() RETURNS BOOLEAN
```

B1 for each correct header: `GetStop` returns a value so it is a function; the two-headed arrow makes `Distance` a `BYREF` parameter of `AddLeg`; the filled-circle arrow up makes `Cancel` a function returning `BOOLEAN`.

(c)

```
PROCEDURE PlanRoute()
    DECLARE StopName : STRING
```

```

DECLARE Distance : REAL
DECLARE Refunded : BOOLEAN
Distance ← 0
StopName ← GetStop()
WHILE StopName <> "END"
    CALL AddLeg(StopName, Distance)
    StopName ← GetStop()
ENDWHILE
IF Distance > 0 THEN
    CALL PrintTicket(Distance)
ELSE
    Refunded ← Cancel()
ENDIF
ENDPROCEDURE

```

B1 procedure header, declarations and `Distance` set to 0; **B1** a loop that ends when "END" is returned; **B1** `GetStop()` used as a function (its value assigned) and `AddLeg` called with both parameters inside the loop; **B1** an IF that calls exactly one of the two modules; **B1** `Cancel()` assigned to a BOOLEAN and `PrintTicket` called with `Distance`.

Because `Distance` is passed by reference, each call of `AddLeg` adds to the caller's `Distance`. For example, if `GetStop()` returns "Mill", "Quay", "END" and `AddLeg` adds 2.5 for Mill and 4.0 for Quay, `Distance` ends at 6.5 and `PrintTicket(6.5)` is called.

Example 4

```

10 FUNCTION Scramble(Word : STRING) RETURNS STRING
11     DECLARE Pos : INTEGER
12     DECLARE Ch : CHAR
13     DECLARE Code : STRING
14     Code ← ""
15     Pos ← 1
16     WHILE Pos <= LENGTH(Word)
17         Ch ← MID(Word, Pos, 1)
18         IF Pos MOD 2 = 1 THEN
19             Code ← Code & TO_UPPER(Ch)
20         ELSE
21             Code ← Ch & Code
22         ENDIF
23         Pos ← Pos + 1
24     ENDWHILE
25     RETURN Code
26 ENDFUNCTION

```

- (a) Complete a trace table for `Scramble("planet")` and give the value returned. **[5]**
- (b) State a more appropriate loop structure and justify your choice. **[2]**
- (c) Line 23 is deleted by mistake. Describe what happens and name the type of error. **[2]**
- (d) Write a new line 21 so that the function returns the letters in their original order, with odd-position letters in upper case and even-position letters in lower case. **[1]**

(a)

Pos	Ch	Code
1	'p'	"P"
2	'l'	"lP"
3	'a'	"lPA"
4	'n'	"nlPA"
5	'e'	"nlPAE"
6	't'	"tnlPAE"
7		

One mark for each block: positions 1–2, 3–4, 5–6, **B1 B1 B1**; the final Pos of 7 that ends the loop **B1**; the returned value "tnlPAE" **B1**. Odd positions are added at the end in upper case; even positions are put on the front, so they come out reversed.

(b) A count-controlled loop (`FOR Pos ← 1 TO LENGTH(Word)`) **B1**, because the number of repetitions (the length of the string) is known before the loop starts **B1**.

(c) Pos stays 1, so the condition `Pos <= LENGTH(Word)` is always true and the loop never ends: the program stops responding **B1**. This is a **run-time error** (an infinite loop) **B1**.

(d) `Code ← Code & TO_LOWER(Ch)` **B1**. `Scramble("planet")` would then return "PlAnEt".

Common mistakes

- **Vague stage reasons.** Examiners see answers that repeat the question's words ("the development stage") or give documents such as "flowchart" or "trace table" for analysis. Analysis produces a **requirements specification**; name a real stage and a real document.
- **Not using the scenario when comparing models.** "Waterfall is old" earns nothing. Say what in the scenario (changing requirements, a tight deadline) makes waterfall a poor fit, then name RAD or iterative.
- **Half-describing stub testing.** Most candidates say a dummy module replaces the missing one, but forget that it returns a **known value** or outputs a message. Say both. A stub replaces a **module**, not the whole program, and it isn't the same as integration testing.

- **Mixing up integration testing with other methods.** Integration is about **combining** modules that were tested on their own; "integrated test data" isn't a thing.
- **Structure charts:** missing the diamond when the notes say "either", drawing a BYREF parameter as a separate parameter plus a separate return value, and explaining a curved arrow as just "a loop" without naming the modules called and their order.
- **State-transition diagrams:** forgetting the START arrow, leaving the output (or input) off a label, and putting labels so far from their arrows that it isn't clear which they belong to.
- **Repeating normal test data.** Two values from the middle of the range test the same thing. Test both ends, just outside both ends, and one clearly invalid value, and keep to whole numbers when the question says the data is an integer.
- **Describing features by their names.** "Single stepping steps through the program" or "a breakpoint is a point where it breaks" earn nothing: say that a breakpoint **pauses execution at a chosen statement** and single-stepping **runs one statement at a time**.
- **Vague adaptive maintenance reasons.** "The program needs updating" is not enough; give a cause such as new hardware, a change in the law or new client requirements.
- **Giving a textbook definition instead of answering about the program.** If the question shows code, quote the statement that causes the error and say which value makes it fail.

How to get the marks

- **"Identify" or "State"** needs just the name ("acceptance testing", "perfective"). If one answer is asked for, only the first one counts, so don't list several.
- **"Describe"** a method needs its main features (stub: dummy module **and** known result). **"Explain"** needs the reason too (why a breakpoint helps: it stops execution where you want to start looking).
- **Errors in code:** one mark for quoting the faulty statement, one for the explanation or the corrected statement. Write the whole corrected line.
- **Structure charts and state-transition diagrams** are marked in groups: all boxes or states, then groups of correct arrows and labels. Extra arrows can cost marks, and labels must be next to their arrows.
- **Module headers:** each part is a mark: PROCEDURE or FUNCTION, parameters with types, BYREF where the arrow is two-headed, and RETURNS <type>.
- **Test plans:** a row earns its mark only with the type, the value and the expected result all correct.
- **Trace tables** are marked in blocks; write values exactly as the code produces them (quotation marks for strings, the upper or lower case that the code produces).
- **Comparing life cycles:** a benefit or drawback "compared to waterfall" must say what is better or worse ("prototypes let the client see the program early" rather than "it is better").
- **Amending code:** say exactly where the change goes and give the full new statement.

Quick check

1. Name the life-cycle stage for each: (i) a state-transition diagram is drawn for the controller; (ii) a trace table is completed using test data; (iii) a fault reported by users six months after release is fixed; (iv) the current paper-based system is observed.
2. Name the type of maintenance: (i) a payroll program is changed because the income tax rate changes; (ii) a search is rewritten so results appear in half the time; (iii) a report that printed the wrong month is fixed.
3. A field stores the number of players in a team, a whole number from 4 to 16 inclusive. Give one value of each test data type (normal, extreme, boundary, abnormal), with the expected result.
4. Use the door-lock state-transition diagram in "Understand it". The lock is in state Locked and these inputs occur: Key, Key, Wrong, Forced, Reset, Key, Correct. Give the outputs in order and the final state.
5. Name the type of error: (i) `FOR Count ← 1 TO 10 ... ENDWHILE`; (ii) an average is found by dividing the total by 10 when 12 values were added; (iii) `Item ← List[Index]` with Index equal to 0 for an array declared `ARRAY[ 1 : 20 ]`.

Answers

1. (i) Design. (ii) Testing. (iii) Maintenance (corrective). (iv) Analysis.
2. (i) Adaptive: a new requirement (the law changed). (ii) Perfective: it already worked, now it is faster. (iii) Corrective: an error found after release.
3. Normal: 10, accepted. Extreme: 4 (or 16), accepted. Boundary: 3 (or 17), rejected (and 4 or 16 accepted). Abnormal: 40, rejected.
4. Key: Locked to Entering. Key: stays in Entering. Wrong: output Beep, back to Locked. Forced: output Siren on, to Alarm. Reset: output Siren off, to Locked. Key: to Entering. Correct: output Unlock, to Open. The outputs are Beep, Siren on, Siren off, Unlock (4 outputs), and the final state is **Open**.
5. (i) Syntax error: a FOR loop must end with `NEXT Count`. (ii) Logic error: it runs but the average is wrong. (iii) Run-time error: index 0 is outside the array bounds, so the program crashes.

Past questions to try

- [9618/23/M/J/22 Q8](#): completing the waterfall model and explaining its arrows.
- [9618/23/O/N/22 Q1](#): benefits of a life cycle, RAD against waterfall, and adaptive maintenance.
- [9618/22/M/J/22 Q1](#): IDE features for testing and debugging.
- [9618/21/O/N/24 Q7](#): a structure chart from six module headers.