

Software development

A-Level Computer Science 9618 · Topic 12 revision notes

Everything from the 5 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. Program development life cycles

2. Structure charts

3. State-transition diagrams

4. Types of error and testing methods

5. Test data and maintenance

1 Program development life cycles

Syllabus 12.1

A **program development life cycle** is a structured sequence of stages used to plan, build and look after a program: **analysis**, **design**, **coding**, **testing** and **maintenance**. Its purpose is to make development organised and predictable, so that the program meets the user's requirements, on time and within budget. Different projects need different life cycles: the **waterfall** model completes each stage once, in order; the **iterative** model repeats the cycle to grow the program in versions; **rapid application development (RAD)** builds prototypes quickly with constant user feedback.

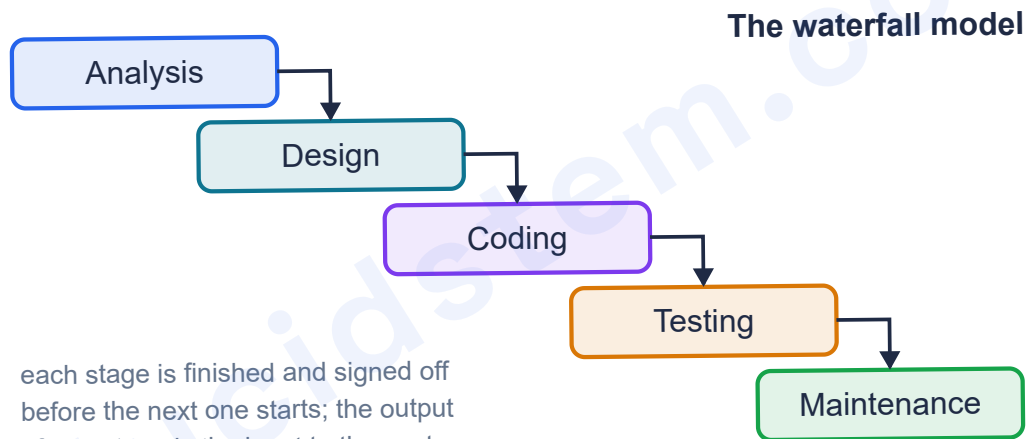


Fig. 1: in the waterfall model each stage flows down into the next, like water over a series of steps

Worked example: Choosing a life cycle

A start-up wants a new booking app for a gym. The owners are not sure exactly what features they want and need something to show investors within six weeks. Which life cycle is most suitable? Justify your choice.

1. The requirements are unclear and likely to change: this rules out waterfall, which needs fixed requirements at the start.
2. Time is very short and a working version must be shown early.
3. The owners can be involved throughout and give feedback on prototypes of the screens.
4. A booking app splits naturally into modules (accounts, timetable, bookings) that teams can build in parallel.

Answer: RAD: prototypes can be built quickly and refined with constant feedback from the owners, the requirements can change, and modules can be developed in parallel to meet the deadline

Match the project.

- ✓ **Stages:** analysis, design, coding, testing, maintenance.
- ✓ **Waterfall:** linear, one stage at a time; best for fixed, clear requirements.
- ✓ **Iterative:** repeated cycles, each adds to a working version.
- ✓ **RAD:** quick prototypes, constant user feedback, parallel time-boxed modules.

Watch out!

Name the stage that matches the activity. Writing the test plan belongs to *design*; running it belongs to *testing*. Finding out what the user needs is *analysis*, not design.

"Waterfall is old so it is bad" earns nothing. It is the right choice when requirements are stable and well understood, for example a small program for a fixed specification or safety-critical software that must be fully documented.

2 Structure charts

Syllabus 12.2

A **structure chart** is a top-down design diagram. Each **rectangle** is a **module** (a procedure or function); a **line** from a module to one below it means the upper module **calls** the lower one. Short arrows beside the lines show the **parameters** passed: an arrow with a **hollow circle** is **data**; an arrow with a **filled circle** is a **flag** (a Boolean). A **diamond** shows **selection** (only one of the modules below is called) and a **curved arrow** shows **iteration** (the modules are called repeatedly). It shows *what* each module does and what it needs, not *how* it does it.

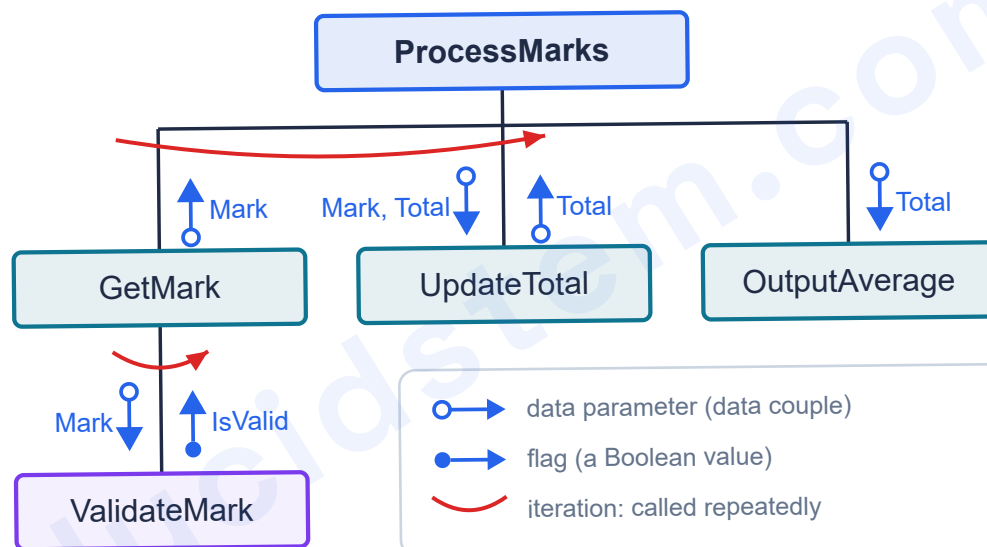


Fig. 1: a structure chart for a program that reads and validates 30 marks, totals them and outputs the average

Worked example: Decompose a problem

A program reads a customer's meter readings, calculates the bill and prints it. CalculateBill needs the Previous and Current readings and produces the Cost. Describe the structure chart.

1. Top box: ProduceBill (the whole problem).
2. Three boxes below, in the order they run, left to right: GetReadings, CalculateBill, PrintBill.
3. GetReadings passes Previous and Current up to ProduceBill (data couples).
4. Previous and Current pass down to CalculateBill; Cost passes up.
5. Cost passes down to PrintBill. No iteration or selection is needed.

Answer: ProduceBill above GetReadings, CalculateBill and PrintBill, with data couples Previous, Current (up from GetReadings, down to CalculateBill) and Cost (up from CalculateBill, down to PrintBill)

Read the arrows.

- ✓ **Box:** a module; a line down means "calls".
- ✓ **Hollow circle:** data; **filled circle:** flag (Boolean).
- ✓ **Diamond:** selection; **curved arrow:** iteration.
- ✓ **Pseudocode:** one value up = FUNCTION; down and back up = BYREF; down only = BYVAL.

Watch out!

A structure chart does not show the detailed steps inside a module, and it is not a flowchart. If a question asks for a structure chart, draw boxes, lines and parameter arrows, not decision boxes and loops of code.

Every parameter arrow needs a label and a direction. An unlabelled arrow, or one with no circle to show data or flag, will not earn the mark.

A function is called inside an expression, never with CALL: write `Mark ← GetMark()`, not `CALL GetMark()`. Procedures are called with CALL.

3 State-transition diagrams

Syllabus 12.2

A **state-transition diagram** documents an algorithm or system whose behaviour depends on its **current state** and the next **input** (event). Each **state** is a circle. Each **transition** is an arrow from one state to another, labelled with the **input** that causes it and, after a slash, any **output** it produces: [input / output]. An arrow from a dot marks the **start state**; a **double circle** marks a **halting (final) state**. The same information can be written as a **state-transition table**.

State-transition diagram for a car alarm

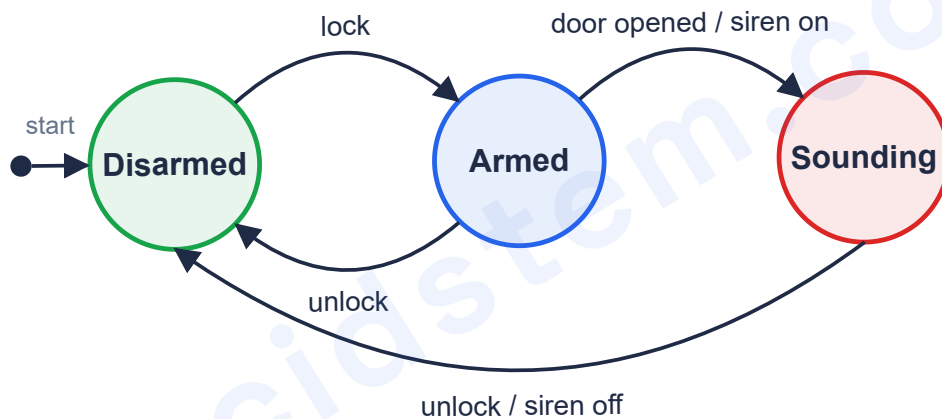


Fig. 1: the alarm starts Disarmed; locking arms it; opening a door while armed sounds the siren; unlocking always disarms it

Worked example: Trace a sequence

The keypad in Fig. 2 receives the inputs wrong, wrong, correct. Give each state it passes through and the final state.

1. Start state: 0 fails.
2. wrong: 0 fails moves to 1 fail.
3. wrong: 1 fail moves to 2 fails.
4. correct: 2 fails moves to Open, a halting state.

Answer: 0 fails, 1 fail, 2 fails, Open: the door opens

State, input, next.

- ✓ **Circle:** a state; **double circle:** halting state; **arrow from a dot:** start.
- ✓ **Arrow label:** input / output.
- ✓ **Trace:** from the start state, follow one arrow per input; no arrow means no change.
- ✓ **Table:** current state, input, output, next state; one row per arrow.

Watch out!

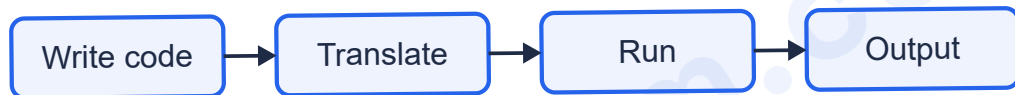
Label every arrow with the input that causes it. An arrow with no label, or a state with no way out when the system should continue, loses marks.

Do not confuse a state-transition diagram with a flowchart. A flowchart shows the steps of a process in order; a state-transition diagram shows states and the inputs that move the system between them.

4 Types of error and testing methods

Syllabus 12.3

A **syntax error** breaks the grammar rules of the language, so the translator cannot translate the code. A **logic error** is a mistake in the algorithm: the program runs but gives the wrong result. A **run-time error** happens while the program is running and makes it crash or hang, for example dividing by zero or using an array index outside its bounds. Faults are **avoided** by good design and programming practice and **exposed** by testing: **dry runs**, **walkthroughs**, **white-box** and **black-box** testing, **stub** and **integration** testing, then **alpha**, **beta** and **acceptance** testing.



avoid faults

good design,
clear names,
IDE checks

syntax error

breaks the rules
of the language;
not translated

run-time error

program crashes
or hangs, e.g.
divide by zero

logic error

program runs but
the result is
wrong

the translator reports syntax errors; only testing and dry runs expose logic errors

Fig. 1: where each type of error shows itself as a program goes from source code to output

Worked example: Locate, identify, correct

CountPasses should return how many of the 30 marks are passes. A pass is 50 or more. The code above contains three errors. Locate each, identify its type and correct it.

1. `ENDFOR` is not a keyword: a FOR loop must end with NEXT. This is a **syntax error**; correct it to `NEXT Index`.
2. `FOR Index ← 0 TO 30` uses `Marks[0]`, which is outside the array bounds 1 to 30. This crashes the program while it runs: a **run-time error**. Correct it to `FOR Index ← 1 TO 30`.
3. `Marks[Index] > 50` does not count a mark of exactly 50 as a pass. The program runs but returns the wrong count: a **logic error**. Correct it to

Answer: Syntax: ENDFOR to NEXT Index; run-time: loop from 1, not 0; logic: >= 50 instead of > 50

When does it show?

- ✓ **Syntax:** breaks the language rules; found by the translator.
- ✓ **Logic:** runs, but wrong result; found by testing or a dry run.
- ✓ **Run-time:** crashes or hangs while running, e.g. divide by zero.
- ✓ **Methods:** dry run, walkthrough, white-box, black-box, stub, integration, alpha, beta, acceptance.

Watch out!

A program that crashes is not a logic error. If it stops while running (divide by zero, index out of range), it is a run-time error; if it finishes but gives the wrong answer, it is a logic error.

Alpha and beta are not about version numbers. Alpha testing is done *in-house* by the developer's staff; beta testing is done by *real users outside* the company.

5 Test data and maintenance

Syllabus 12.3

Normal data is typical data that should be accepted. **Abnormal** (erroneous) data should be rejected: out of range or the wrong type. **Extreme** data is the largest or smallest value that is still accepted. **Boundary** data tests each side of a limit: the last value accepted and the first value rejected. A **test strategy** sets out the overall approach to testing; a **test plan** lists each test with its data, its type and the **expected result**. After release a program needs **maintenance**: **corrective** (fix errors), **adaptive** (change it for new requirements or a new environment) and **perfective** (improve it, for example its speed or ease of use).

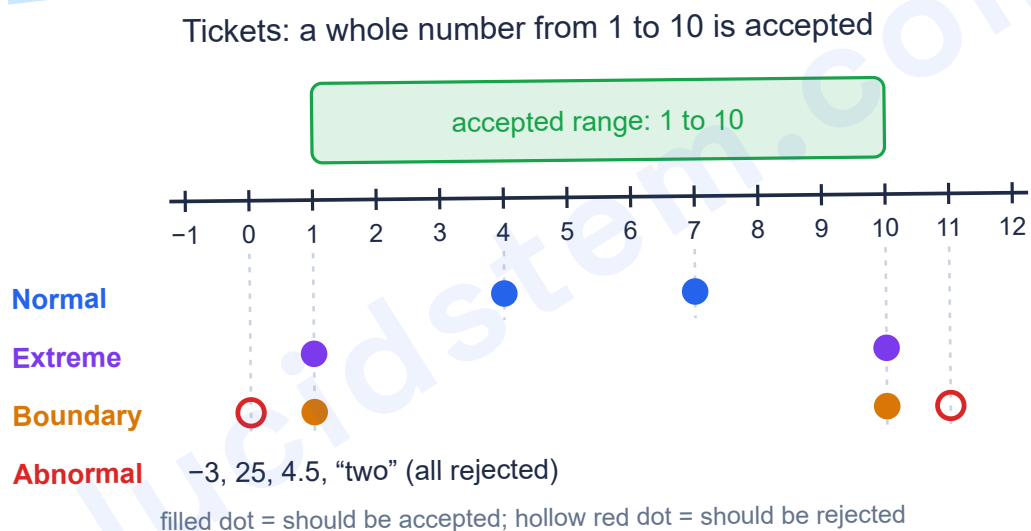


Fig. 1: test data for an input that accepts a whole number of tickets from 1 to 10

Worked example: Test data for a length check

A password must be from 8 to 16 characters long. Give test data of each type.

1. Normal: a password of 12 characters (accepted).
2. Extreme: passwords of exactly 8 and exactly 16 characters (both accepted).
3. Boundary: 7 characters (rejected) and 8 (accepted); 16 (accepted) and 17 (rejected).
4. Abnormal: 3 characters or 30 characters, or an empty string (all rejected).

Answer: Normal 12; extreme 8 and 16; boundary 7/8 and 16/17; abnormal e.g. 3, 30 or empty

Data, then expected result.

- ✓ **Normal:** typical, accepted. **Abnormal:** rejected.
- ✓ **Extreme:** largest and smallest accepted. **Boundary:** last accepted and first rejected at each limit.
- ✓ **Test plan:** number, purpose, data, type, expected result, actual result.
- ✓ **Maintenance:** corrective fixes, adaptive changes, perfective improves.

Watch out!

Always give the expected outcome with your test data, and give the actual value, not a description. "A number just outside the range" earns nothing; "11, rejected" does.

The actual result column is empty in the plan: it is filled in when the test is carried out. Do not confuse the test plan (written in design) with the test results.

Corrective maintenance fixes something that was wrong. If the program was correct but the world or the requirements changed, it is adaptive; if it worked but is being made better, it is perfective.