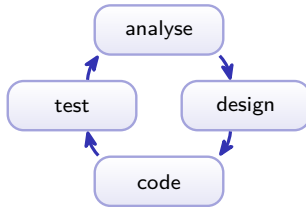


Software Development

AS Computer Science ■ Topic 12

A-Level Computer Science



What we will cover

- 1 The life cycle 12.1
- 2 Structure charts 12.2
- 3 State-transition diagrams 12.2.1
- 4 Testing and maintenance 12.3
- 5 Test strategy and plans 12.3.1
- 6 Recap



1

The life cycle

Sheet 12.1 – what you need to know

You must be able to:

- name each stage of the life cycle from an activity
- compare waterfall, iterative and RAD
- use stub and acceptance testing

What the exam asks

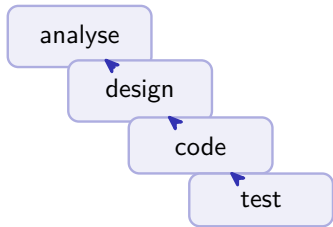
“Complete the table by writing the name of the life cycle stage for each activity.” [5]

“Complete the table to define a test plan to thoroughly test the operation of the function.” [4]

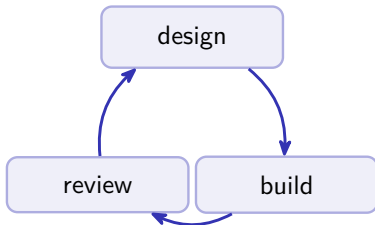
“Identify a method to continue testing the main program before the errors in these modules have been corrected...” [3]

Practise on exercise sheet 12.1, then try the real exam questions on past-paper sheet 12.1.

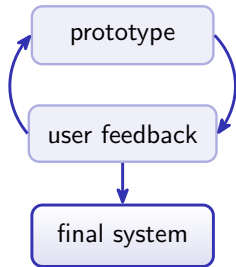
Development life-cycle models



waterfall: linear



iterative: repeated passes



RAD: quick prototypes

Waterfall 瀑布模型: stable needs. **Iterative** 迭代模型/**RAD** 快速应用开发/**Agile** 敏捷: needs discovered or changing.

Program development life cycle

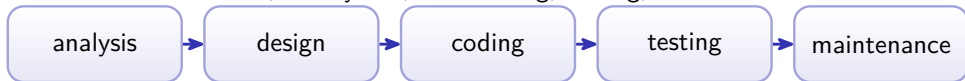
Tabel Simbol-simbol flowchart program

No.	Simbol	Keterangan
1		Kotak Terminal (Terminal Symbol/Terminator) ➤ simbol dimulainya (Start/Begin) dan diakhirinya suatu program (Finish/End).
2		Kotak Proses (Processing Symbol) ➤ tempat untuk menuliskan perintah/operasi aritmatika.
3		Kotak Input dan Output (Input/Output Symbol) ➤ tempat untuk memberikan masukan (input) atau menampilkan keluaran (output) dari suatu operasi.
4		Kotak Keputusan (Decision Symbol) ➤ tempat pertanyaan/pengujian kondisi, berisikan operasi perbandingan logika, dengan dua nilai keluaran (YA/BENAR/TRUE dan TIDAK/SALAH/FALSE).
5		Lingkarang Penyambung (Connector Symbol) ➤ digunakan jika diagram alir belum selesai dan akan berlanjut ke sampingnya pada halaman yang sama.
6		Off-page Connector Symbol ➤ digunakan jika diagram alir belum selesai dan akan berlanjut ke halaman yang berbeda.
7		Looping Symbol/Preparation Symbol ➤ digunakan untuk memberikan nilai awal pada suatu variabel/counter dan menggambarkan proses yang perulangan.
8		Arah Proses (Direction) ➤ untuk menunjukkan arah aliran proses program.

A flowchart plans a program's logic during the design stage of the cycle

The standard stages

Analysis decides *what* is needed; **design** decides *how* – data structures, algorithms, modules, interface, file layouts; then coding, testing, maintenance.



analysis = *what*; design = *how*; then code, test and maintain.

Name the stage from the activity

activity	stage
interview the users; write the requirements specification	analysis
draw structure charts; write the identifier table	design
write the program from the pseudocode	coding
run the test plan; correct the errors	testing
add a feature after the program is in use	maintenance

“Identify a method to keep testing the main program before the errors in these modules are corrected.” [3]

Stub testing: replace each faulty module with a **stub**, a simple module with the same **interface** that returns a fixed value or outputs a message, so the main program can still run.

Acceptance testing: the **client** tests the finished program against the requirements.

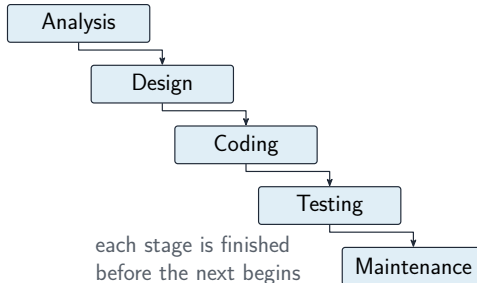
Choosing a life cycle

No single life cycle fits every project, so there are different ones: the choice follows the project.

Factor	Waterfall	Iterative / RAD / Agile
requirements 需求	clear and fixed at the start	unclear or changing
expected change	little	frequent
risk	low, well understood	high – find problems early
users	sign off once	give feedback every cycle

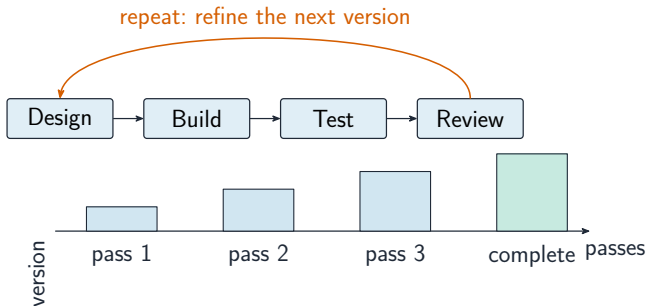
The waterfall model

A **linear** sequence, each stage finished before the next. Clear and well-documented, good for **stable requirements** – but poor at mid-project change, and the customer sees nothing working until the end.



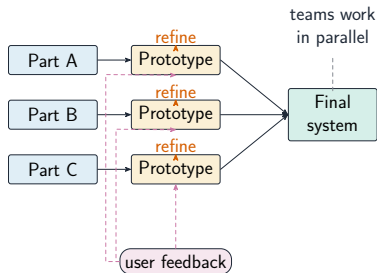
The iterative model

Repeated passes, each producing a partial version that is reviewed and refined. Catches problems earlier and suits requirements **discovered over time** – but is harder to estimate.

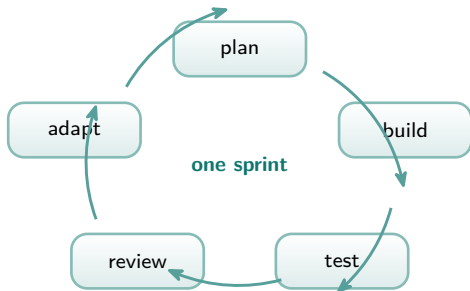


Rapid application development

Heavy use of a **prototype** 原型 and user feedback. Very fast first delivery and good for **changing requirements** – but it depends on users being available, and suits smaller systems.



Agile: short sprints, constant contact



Agile 敏捷 runs short iterations – **sprints** – with constant collaboration and testing throughout.

Flexible and adaptive, but it needs a **committed customer** and a skilled team. No model is best: choose on size, how clear the **requirements** 需求 are, how much change is expected, the risk, the team and the deadline.

2

Structure charts

Sheet 12.2 – what you need to know

You must be able to:

- draw a structure chart with parameters and return values
- read selection (diamond) and iteration (curved arrow)
- write module headers from a chart, and a chart from headers

What the exam asks

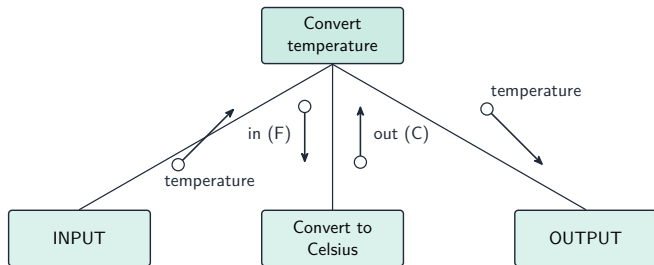
“Draw a structure chart for the five modules, including all parameters and return values.”
[4]

“Explain the meaning of the curved arrow symbol in this structure chart.” [2]

“Write the pseudocode module headers for analyse, update and sort.” [4]

Practise on exercise sheet 12.2, then try the real exam questions on past-paper sheet 12.2.

Design tools: the structure chart



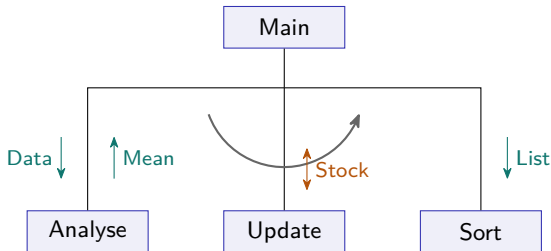
○ → parameter passed along a link (data couple)

A **structure chart** 结构图 shows:

- the **hierarchical decomposition** 分解 into **modules** (subroutines 子程序)
- the **parameters** 参数 passed between them

A design-stage tool: read each procedure's **signature** 签名 straight off it.

From a structure chart to module headers



```
FUNCTION Analyse(Data : ARRAY) RETURNS REAL  
PROCEDURE Update(BYREF Stock : INTEGER)  
PROCEDURE Sort(BYREF List : ARRAY)
```

- a value **returned up** makes a FUNCTION
- a **two-way** arrow: changed and passed back, so BYREF
- the **curved arrow**: the modules below are called **repeatedly** (iteration)
- a filled-circle arrow is a **flag**, a Boolean

3

State-transition diagrams

Sheet 12.2.1 – what you need to know

You must be able to:

- draw a state-transition diagram from a table
- label each arc with input / output, including self-loops
- trace a sequence of inputs

What the exam asks

“Complete the state-transition diagram to represent the information given in the table.”

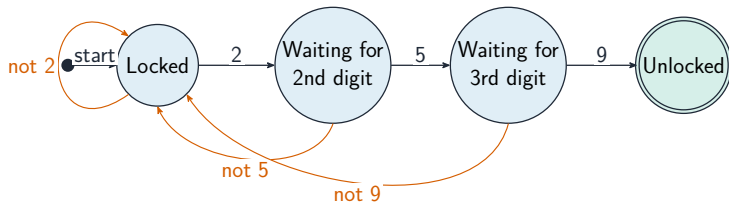
[5]

“Complete the following state-transition diagram for the heating system.” **[3]**

Practise on exercise sheet 12.2.1, then try the real exam questions on past-paper sheet 12.2.1.

State-transition diagram

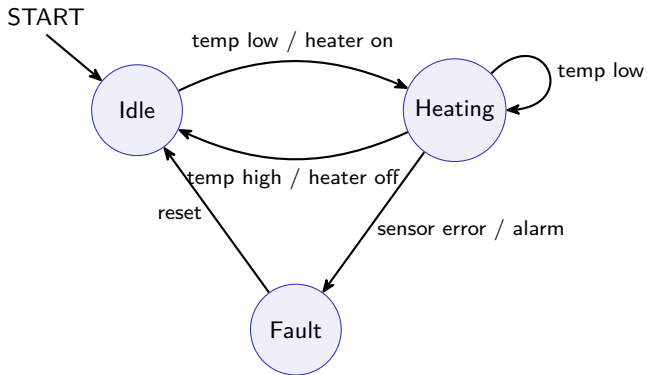
A **state-transition diagram** 状态转换图 shows the **states** 状态 a system can be in and the **events** that move it between them.



states & the events that move between them

Documents the behaviour of a system – vending machines, traffic lights, logins.

State-transition diagrams: notation and tracing



- a **START** arrow points at the first state
- each arc: **input** / **output**; no output, input only
- a **self-loop**: the input keeps the same state

Trace: temp low, temp low, sensor error, reset

Idle → Heating (heater on) →
Heating → Fault (alarm) → Idle.

Table to diagram: one arc per row, from its current state to its next state.

4

Testing and maintenance

Sheet 12.3 – what you need to know

You must be able to:

- name syntax, logic and run-time errors
- choose normal, boundary and erroneous test data
- use the IDE's debugging tools in order; name the kinds of maintenance

What the exam asks

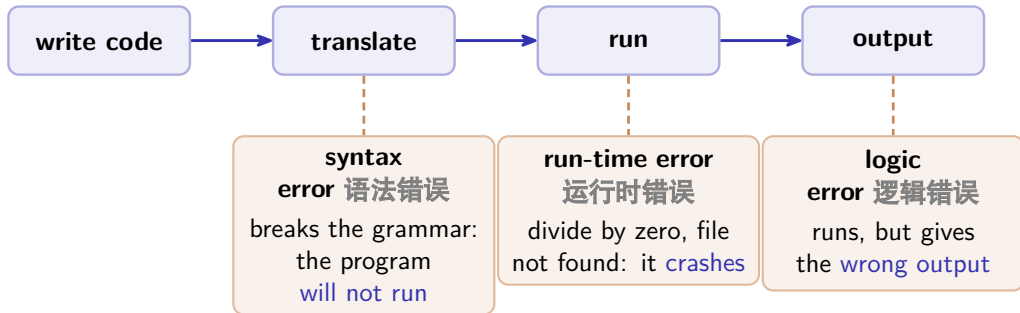
“Describe stepwise refinement.”
[2]

“Identify and describe one type of error that black-box testing could detect.” [2]

“Explain these features including the order that they will be used in to locate the error.” [3]

Practise on exercise sheet 12.3, then try the real exam questions on past-paper sheet 12.3.

Three kinds of error



Only a **logic error** gets past the computer: find it with a **dry run** 手工跟踪 or a trace table.

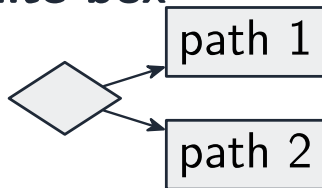
Testing methods

- **black-box testing** 黑盒测试 – from the spec, ignoring the code
- **white-box testing** 白盒测试 – every code path
- a **unit test** 单元测试 checks one routine, with a **stub** 桩程序 standing in for anything it calls; **integration testing** 集成测试 then checks them together
- **alpha** α 测试 (in-house), **beta** β 测试 (real users), **acceptance testing** 验收测试 (the customer signs it off)

black-box

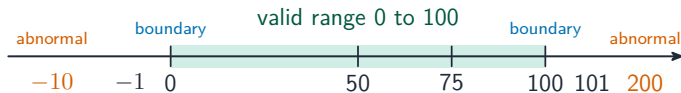


white-box



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 正常数据

50, 75
well inside

extreme 极端数据

0, 100
edges, still accepted

abnormal 异常数据

–10, 200
must be rejected

boundary 边界数据

0/–1, 100/101
both sides of the line

Choosing test data, and the stages of the life cycle

normal data	typical values the program should accept
boundary data 边界数据	the values at the edge of the valid range, and just outside it
extreme data 极端数据	the largest and smallest valid values
abnormal data	values that should be rejected – wrong type, out of range, empty
a stub 桩	a placeholder for a module that does not exist yet, so the structure can be tested top-down
a walkthrough 走查	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.

Worked example: catch the logic error

“Award a pass for marks of 50 or above”:

```
INPUT Mark
IF Mark > 50 THEN
  OUTPUT ``Pass''
ELSE
  OUTPUT ``Fail''
ENDIF
```

the bug: > should be >=

Mark	expected	actual	
75	Pass	Pass	✓
30	Fail	Fail	✓
50	Pass	Fail	X

Only the **boundary value** 50 exposes it – the normal values both pass.

A logic error runs without crashing – compare **expected vs actual** output, and always test the exact boundary.

Worked example – four kinds of test data

A field accepts an exam mark from 0 to 100. Give test data of each kind, with its expected result.

- **Normal:** 50 – accepted, a typical value inside the range.
- **Abnormal:** -10, 200, äbč – rejected, out of range or the wrong type.
- **Extreme:** 0 and 100 – the largest and smallest values still *accepted*.
- **Boundary:** the pairs either side of each edge – -1 rejected beside 0 accepted.

Every value must carry its **expected result**, or the test plan proves nothing. An extreme value sits inside and is accepted; a boundary pair sits either side of the edge.

Black-box testing, the IDE in order, and refinement

“Identify and describe one type of error that black-box testing could detect.” [2]

A **logic error**: for given inputs the outputs are not the expected ones, though the program runs.

“Describe stepwise refinement.” [2]

Break the problem into **smaller steps**, then break each step down again until each can be written as **one line** of code.

“Explain these features, including the order they are used in, to locate the error.” [3]

- 1 set a **breakpoint** where the error may start: the program pauses there
- 2 **single-step** one statement at a time
- 3 watch the **variables** (watch window) change after each step, to see where a value goes wrong

Perfective maintenance: improving a working program, for example making it faster.

Maintenance and amending code

perfective

improve
features
or speed

adaptive

keep it
working in a
changed
environment

corrective

fix bugs found
in use

- **perfective maintenance** 完善性维护 – improve it
- **adaptive maintenance** 适应性维护 – a new environment
- **corrective maintenance** 纠正性维护 – fix bugs

Amending a program: read it, make the **smallest** change, update callers, then

regression test 回归测试:
check the new behaviour *and*
that you broke nothing old.

The three kinds of maintenance, and two design tools

corrective	fixing faults found after release
adaptive	changing the program to suit a new environment – a new OS, a new law, a new data format
perfective 完善性维护	improving performance or features even though it works
which dominates	maintenance is most of a program's total cost, and perfective is the largest share of it
a structure chart 结构图	shows the hierarchical decomposition 分解 of a program into modules, with the data passed between them
a state-transition diagram	makes missing transitions easy to spot – “what if a second coin is inserted while awaiting selection?”

That question about the second coin is the point of the diagram. It surfaces the cases nobody thought of, which is where most real faults live.

5

Test strategy and plans

Sheet 12.3.1 – what you need to know

You must be able to:

- tell a test strategy from a test plan
- write test-plan rows with expected results
- say how to expose a fault and how to avoid one

What the exam asks

“Identify this additional information and explain why it is needed.” [3]

“Explain why the compiler does not identify this error.” [1]

“Complete the following state-transition diagram for the heating system.” [3]

Practise on exercise sheet 12.3.1, then try the real exam questions on past-paper sheet 12.3.1.

Test strategy vs test plan

test strategy 测试策略

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

test plan 测试计划

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.

Exposing faults, avoiding them, and a walkthrough

expose a fault

dry run /
walkthrough

black-box tests with
expected results

white-box tests of
every path

avoid a fault

modular design, tested
modules

meaningful names and
comments

library routines already
tested

“Explain why the compiler does not identify this error.” [1]

It is a **logic** error: the code obeys the language’s rules, so it compiles.

A walkthrough: what is needed, and why

The **algorithm** (pseudocode), **test data** with **expected results**, and a **trace table**, so each step can be followed and the result compared.

A test-plan row

test number · purpose · test data · type
(normal, boundary, erroneous) · **expected result** · actual result

6

Recap

Exam tips

- Compare development models (waterfall, iterative, RAD) and know the stages of the **program development life cycle**.
- Distinguish **syntax, logic and run-time** errors and how each is found.
- Choose **test data** of three kinds – normal, boundary and erroneous – and give an example of each for the stated range.
- Distinguish the types of **maintenance** (corrective, adaptive, perfective).

Topic 12 – key takeaways

1

Life cycle

waterfall vs iterative/RAD/agile

2

Errors

syntax, run-time, logic

3

Testing

black/white box; normal, extreme, abnormal data

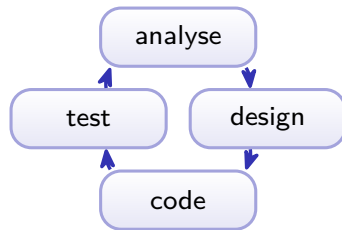
4

Maintenance

perfective, adaptive, corrective;
regression test

Check yourself

- 1 Which error gives wrong output but does not crash?
- 2 Which testing uses only the specification?
- 3 Test data 0 and 100 for a 0–100 field – which kind?
- 4 What is regression testing?



Answers 1. a logic error 2. black-box 3. extreme 4. re-test old features still work