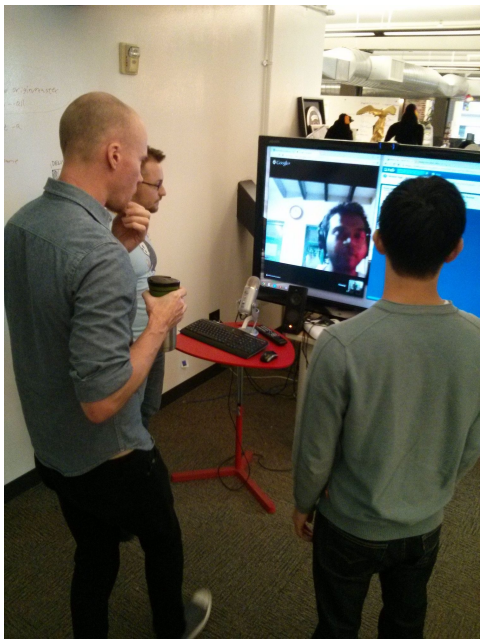
# Software Development

A-Level Computer Science

## Program development life cycle

A **development life cycle** 开发生命周期 is the set of stages from idea to finished, maintained software. It exists to **plan, manage and control** a project —to build the right product, on time, with good quality.



*Software is built by teams who follow a development life cycle to stay coordinated*

**Tabel Simbol-simbol *flowchart* program**

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

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



## Why a life cycle is needed

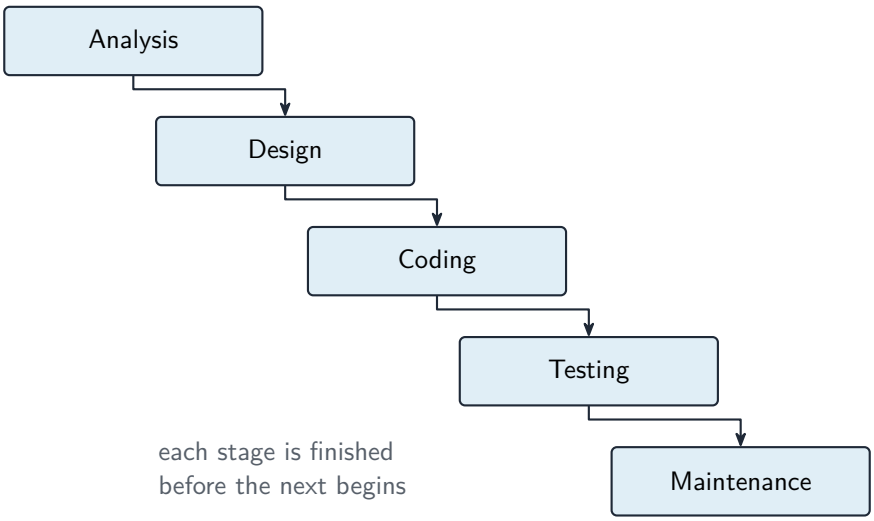
It manages complexity (break a big program into phases), coordinates teams, tracks progress with milestones, builds in testing, records design decisions for later, and manages risk.

## Why there are different ones

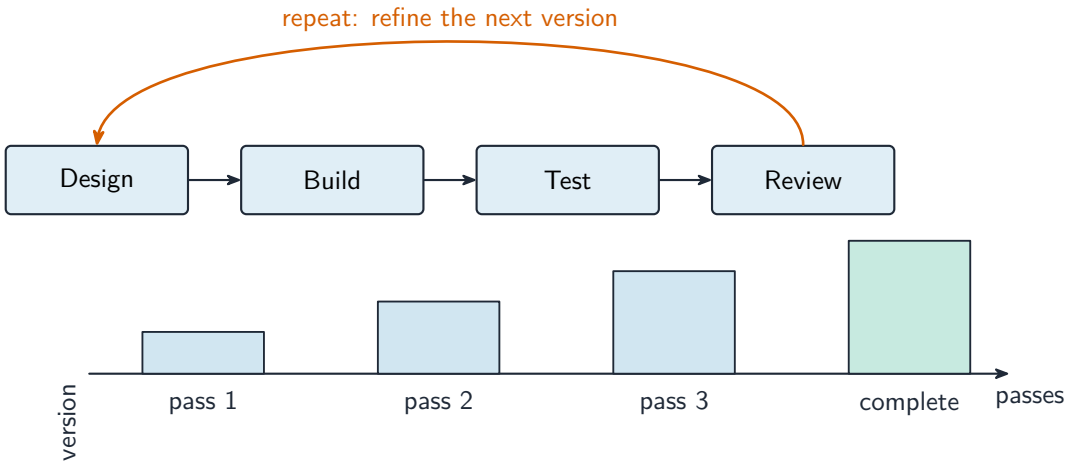
No single life cycle fits every project, so several **development life cycles** exist. The choice depends on the size and complexity, how clear the **requirements** 需求 are at the start, how much change is expected, the risk level, the team, and the deadline.

## Common models

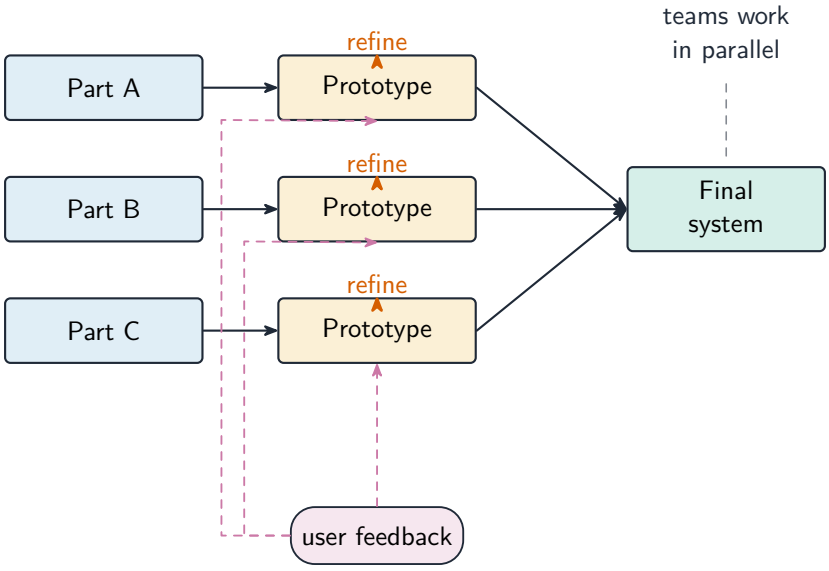
- **Waterfall** 瀑布模型—a linear sequence (Analysis → Design → Coding → Testing → Maintenance), each stage finished before the next. Clear and well-documented; good for **stable requirements**, but poor at coping with mid-project change, and the customer sees nothing working until the end.
- **Iterative model** 迭代模型—repeated passes, each producing a partial version that is reviewed and refined. Catches problems earlier; good when requirements are **discovered over time**, but harder to estimate.
- **Rapid Application Development** 快速应用开发 (RAD) —heavy use of a **prototype** 原型 and user feedback. Very fast first delivery; good for **changing requirements**, but depends on user availability and suits smaller systems.
- **Agile** 敏捷—short iterations ("sprints"), constant collaboration and testing. Flexible and adaptive, but needs a committed customer and a skilled team.




*The waterfall model: each stage is finished before the next begins*




*The iterative model: repeated passes refine the program*




*Rapid application development: teams work on parts in parallel*

### The standard stages

- **analysis** —find **what** the program must do; gather and document requirements.
- **design** —decide **how**: data structures, algorithms, modules, interface, file layouts.
- **coding (implementation** 实现 **)** —write the source code following the design.
- **testing** —run against test data and fix bugs.
- **maintenance** 维护—after release, keep it working and useful.

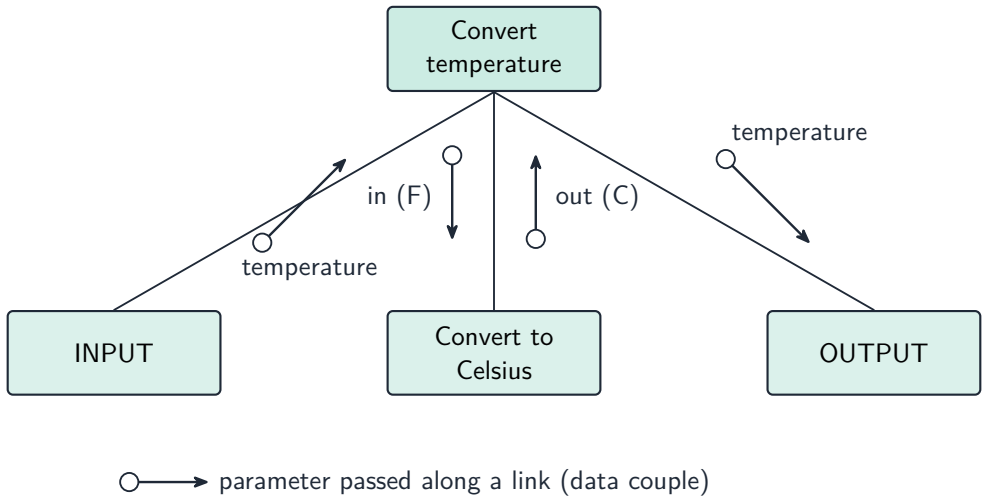
## Program design tools

### Structure chart

A **structure chart** 结构图 shows the **hierarchical decomposition** 分解 of a program into modules (**subroutines** 子程序) and the **parameters** 参数 passed between them. Each module is a rectangle; lines link caller (above) to callee (below); small arrows show data going down and results coming back up. The design can then be turned into equivalent **pseudocode** 伪代码.

```
                CalculatePay
            /        |         \
GetEmployee  CalculateBonus  CalculateTax
Returns:     Takes: sales    Takes: gross
employeeID   Returns: bonus  Returns: tax
```

It is a design-stage tool, and you can read the procedure signatures off it.



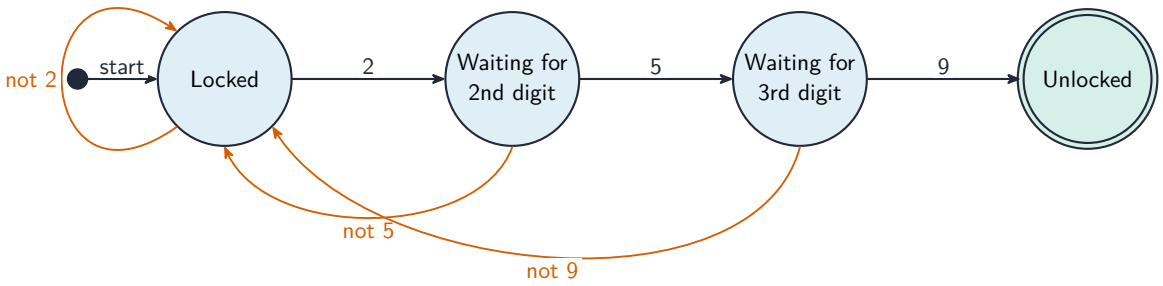

*A structure chart: modules with the parameters passed between them*

## 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 —good for vending machines, traffic lights, user interfaces. **State-transition diagrams** are used to document the behaviour of an algorithm or system. Each state is a circle; each transition is an arrow labelled with the event.

```
   coin inserted                item selected
[Idle]        → [Awaiting selection]      → [Dispensing]
```

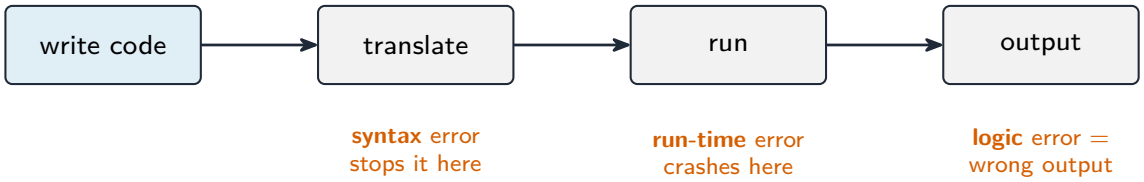
It makes missing transitions easy to spot ("what if a second coin is inserted while awaiting selection?").




*A state-transition diagram for a door lock with code 259*

# Errors

- **syntax error** 语法错误—breaks the language's grammar (missing bracket, misspelled keyword). Caught at translation time; the program won't run until fixed.
- **run-time error** 运行时错误—happens while running (divide by zero, file not found, array index out of range). The program crashes or raises an exception; fix by adding checks.
- **logic error** 逻辑错误—the program runs but gives **wrong results** (using `+` for `-`, an off-by-one loop, conditions in the wrong order). The hardest to find; the only sign is wrong output, so use careful testing and tracing.




*When each error shows up: syntax at translation, run-time during the run, logic in the output*

# Testing methods

- **dry run** 手工跟踪—trace the code on paper, writing each variable's value in a table.
- **walkthrough** 走查—a team review of the code.
- **white-box testing** 白盒测试—designed from the code's internal structure, covering every statement, branch and loop.
- **black-box testing** 黑盒测试—designed from the specification only: feed inputs, check outputs.
- **integration testing** 集成测试—combine modules and test the interfaces between them.
- **alpha testing** $\alpha$ 测试—by the developers/in-house before release; **beta testing** $\beta$ 测试—by a limited group of real users in their own environment.
- **acceptance testing** 验收测试—by the customer, to decide if the product is fit for purpose.
- **stub** 桩—a placeholder for a module that does not exist yet, so the structure can be tested top-down.


*Black-box tests the specification; white-box tests the code paths*

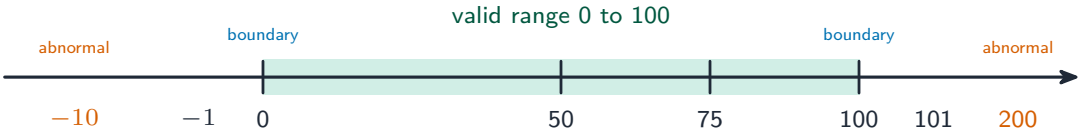
# Test strategy and test plan

A **test strategy** 测试策略 is the **high-level approach** —which kinds of testing, who does them, when, and the criteria to move on. A **test plan** 测试计划 is the **detailed list of tests** —each with input data, expected output, and a column for the actual output.

## Choosing test data

For each field or condition, include three kinds:

- **normal data** 正常数据—typical values inside the valid range (for marks 0–100: `50`, `75`).
- **abnormal data** 异常数据—values that should be rejected (`-10`, `200`, `"abc"`).
- **extreme data** 极端数据—the largest and smallest values still **accepted** (`0` and `100`).
- **boundary data** 边界数据—values at the edges, where off-by-one errors hide (each accepted extreme and the rejected value just outside it: `0`/`-1`, `100`/`101`).




normal (50, 75) inside; extremes (0, 100) the accepted edges; abnormal (−1, 101, −10, 200) rejected

*Test data for a 0–100 field: normal inside, extremes at the boundaries, abnormal outside*

**Worked example.** 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`, `"abc"` - all rejected, being out of range or the wrong data type. **Extreme**: `0` and `100` - the largest and smallest values that are still **accepted**. **Boundary**: the pairs straddling each edge - `-1` rejected alongside `0` accepted, and `100` accepted alongside `101` rejected. Every value must carry its **expected result**, or the test plan proves nothing. Extreme and boundary are the pair most often confused: an extreme value sits **inside** and is accepted, while a boundary test is always a **pair** either side of the edge - which is exactly where off-by-one errors hide.

# Maintenance

Most of a program's lifetime cost is in maintenance. Three kinds:


*Three kinds of maintenance: perfective, adaptive and corrective*

- **perfective maintenance** 完善性维护—improving performance or features even though it works (a faster query, a new option).
- **adaptive maintenance** 适应性维护—keeping it working in a changing environment (a new OS, a new API, a legal change).
- **corrective maintenance** 纠正性维护—fixing bugs found in use.

A program may need all three throughout its life.

# Amending an existing program

When asked to add a feature or fix a bug:

1. **read the existing code** until you understand the algorithm and data flow.
2. **find where the change goes** —which subroutine, which lines.
3. **make the change as small as possible** —don't rewrite working code.
4. **update related parts** —every caller of a changed parameter list, every routine using a changed data structure.
5. **test** the new behaviour **and** the old (**regression testing** 回归测试—check you broke nothing).
6. **document** the change.

Clear comments, meaningful names, decomposed subroutines and a structure chart make a program much easier to amend —which is why the design tools matter even after the first release.

# 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).