12 Software Development

12.1 Program Development Life cycle

Candidates should be able to:	Notes and guidance
Show understanding of the purpose of a development life cycle	
Show understanding of the need for different development life cycles depending on the program being developed	Including: waterfall, iterative, rapid application development (RAD)
Describe the principles, benefits and drawbacks of each type of life cycle	
Show understanding of the analysis, design, coding, testing and maintenance stages in the program development life cycle	

12.2 Program Design

Candidates should be able to:	Notes and guidance
Use a structure chart to decompose a problem into sub-tasks and express the parameters passed between the various modules/procedures/functions which are part of the algorithm design	Describe the purpose of a structure chart Construct a structure chart for a given problem Derive equivalent pseudocode from a structure chart
Show understanding of the purpose of state-transition diagrams to document an algorithm	

12.3 Program Testing and Maintenance

Candidates should be able to:	Notes and guidance
Show understanding of ways of exposing and avoiding faults in programs	
Locate and identify the different types of errors	• syntax errors • logic errors • run-time errors
Correct identified errors	
Show understanding of the methods of testing available and select appropriate data for a given method	Including dry run, walkthrough, white-box, black-box, integration, alpha, beta, acceptance, stub
Show understanding of the need for a test strategy and test plan and their likely contents	
Choose appropriate test data for a test plan	Including normal, abnormal and extreme/boundary
Show understanding of the need for continuing maintenance of a system and the differences between each type of maintenance	Including perfective, adaptive, corrective
Analyse an existing program and make amendments to enhance functionality	

Program Development Life cycle

Development models

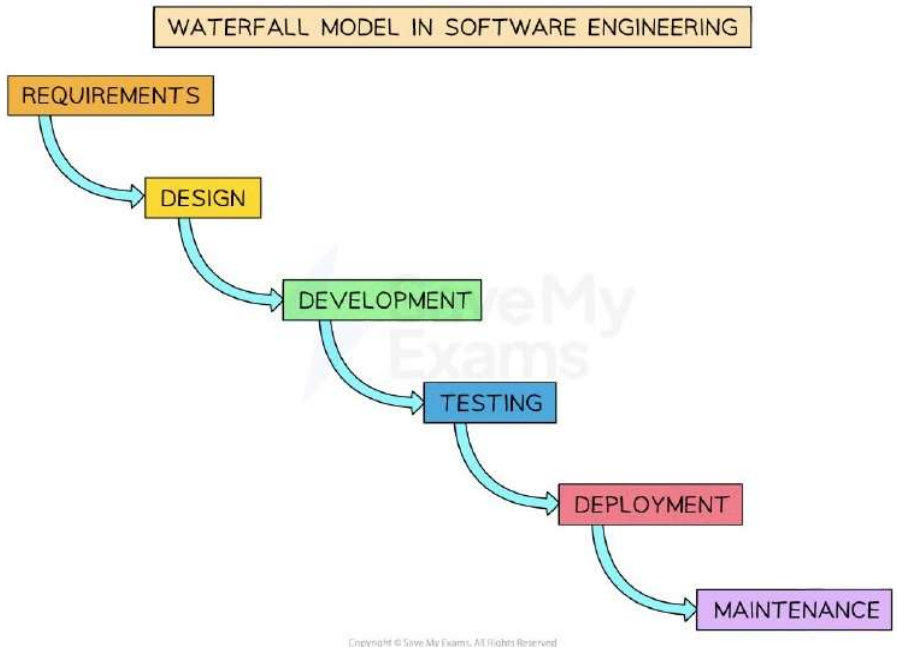
Waterfall

What is the waterfall model?

- The Waterfall Model is a **sequential software development process** divided into **distinct phases**
- Each phase **must be completed** before the next one begins

Steps

1. **Requirement Gathering and Analysis:** All possible system requirements to be developed are captured and documented clearly
2. **System Design:** The requirements are translated into a design. Architects and designers define the overall architecture and identify the main components
3. **Implementation:** The actual code is written in this phase based on the design documents, turning the system design into a functional program
4. **Integration and Testing:** All the components and modules are integrated and tested to ensure that the entire system works as expected
5. **Deployment:** The product is released to the market or handed over to the client. It may involve installation, customization, and training
6. **Maintenance:** Post-release, the system needs regular maintenance to fix bugs, improve performance, or add new features



The Waterfall Model in Software Engineering

Benefits and drawbacks

Benefits	Drawbacks
Simple and linear – easy to understand and follow	**Inflexible** – difficult to make changes once development begins
Clear stages and milestones – easy to track progress	**Expensive to fix late problems** – issues found late are harder to resolve
Ideal for well-defined projects – works best when requirements are fixed	**Long development cycle** – each stage must be completed before moving on

Suitability

- The Waterfall Model is most suitable for projects where **requirements are well understood** and **unlikely to change**
- It works well when **high quality and compliance are essential**, and there is a clear understanding of the project's goals and constraints

Iterative (Agile)

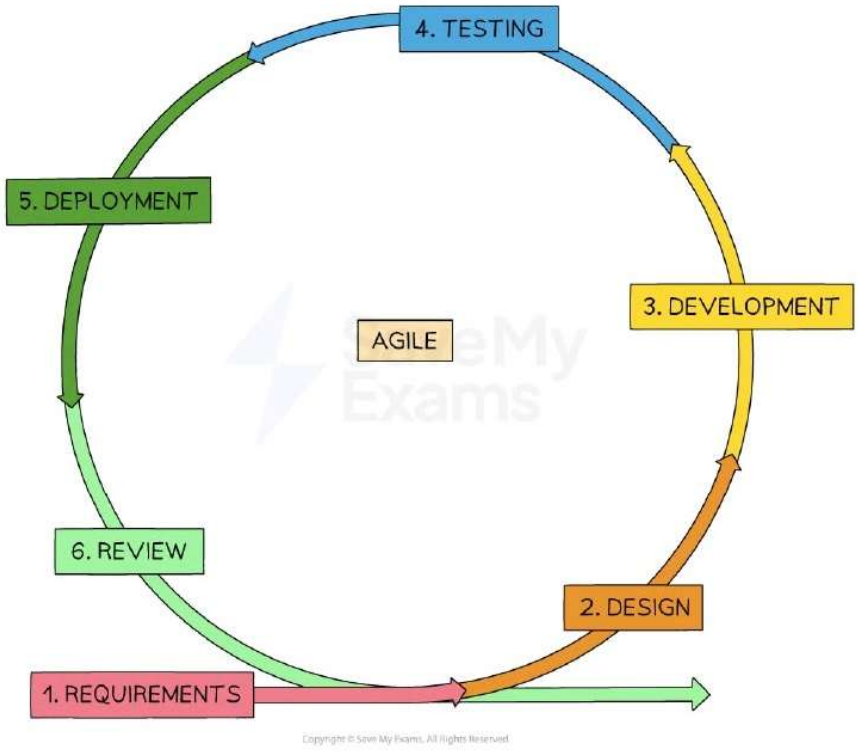
What is the iterative model?

- The iterative model s a **type of Agile software development methodology** that promotes **adaptability** and high customer involvement

Steps

1. **Identify user stories and requirements**
    1. Work closely with stakeholders to gather functional and non-functional requirements
    2. Requirements are often written as **user stories** (e.g. *As a user, I want to...*)
2. **Plan the sprint (Sprint Planning)**
    1. Break down requirements into **tasks**
    2. Choose a set of tasks (features) for the **current sprint** (a short time-boxed development period, usually 1–4 weeks)
    3. Define the **sprint goal**
3. **Design the solution**
    1. Decide how the selected features will be built
    2. Focus is on **simple and adaptable design**, not heavy upfront documentation

4. **Develop the features**
    1. Write code for the selected tasks in the sprint
    2. Developers often work in pairs or small teams (e.g. **pair programming**)
5. **Test continuously**
    1. Perform **unit testing**, **integration testing**, and **acceptance testing** during the sprint
    2. Testing is **ongoing**, not saved for the end
6. **Review progress (Sprint Review)**
    1. Demo the working software to stakeholders
    2. Collect feedback and identify changes or improvements
7. **Reflect on process (Sprint Retrospective)**
    1. The team reflects on what went well and what needs improving in the next sprint
    2. Focus is on **team performance and process optimisation**
8. **Release** (may happen after every sprint or set of sprints)
    1. Deploy working software to users or staging environment
9. **Repeat**
    1. Move to the next sprint with updated priorities and feedback



Benefits and drawbacks

Benefits	Drawbacks
Highly adaptable – responds quickly to changing requirements	**Requires experienced team members** – may be hard to manage without expertise
Frequent communication – promotes constant collaboration	**Risk of burnout** – intense collaboration can tire the team
Focus on quality – encourages good design and continuous testing	**May lack documentation** – flexibility can reduce written records
Customer collaboration – ensures the product meets real needs	**Scope creep** – changing goals may lead to uncontrolled growth

Suitability

- The iterative model is most suitable for **small to medium-sized projects** where **requirements can change** and customer involvement is high

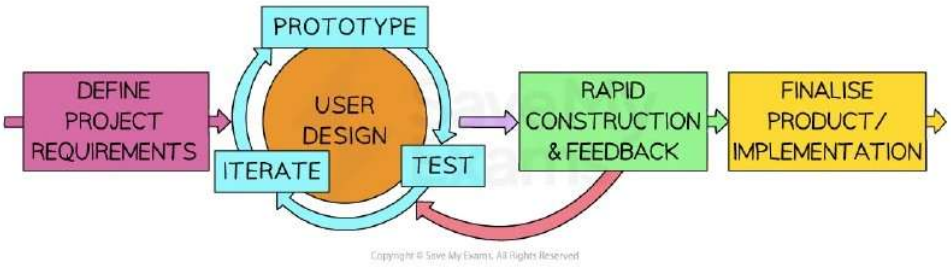
Rapid Application Development (RAD)

What is RAD?

- Rapid Application Development (**RAD**) is a software development methodology that **emphasises fast and iterative development**

Steps

1. **Requirement planning:** Gather general system requirements, define constraints and assumptions
2. **User design and prototyping:** Collaborate with users to develop prototypes, ensuring alignment with user needs
3. **Construction or iterative development:** Build the system incrementally, with continuous user feedback and adaptation
4. **Cutover or deployment:** Transition the product into the live environment, including user training, support, and documentation
5. **Maintenance and updates:** Continue to adapt and improve the system based on user feedback and needs



Rapid Application Development (RAD) Model of Software Development

Benefits and drawbacks

Benefits	Drawbacks
Speed – fast development and delivery at relatively low cost	**Requires strong team collaboration** – skilled and cohesive teams are essential
User involvement – client feedback shapes the system throughout	**Potential quality issues** – speed may reduce testing and documentation
Flexibility – adapts quickly to changing requirements	**Not ideal for small projects** – may be too complex for simple systems
Incremental development – builds in small, testable parts	**Scope creep risk** – flexibility can lead to ever-expanding requirements

Suitability

- Rapid Application Development is most suitable for projects where **rapid delivery is required** and where requirements can be developed and **refined on the go**

Program Design

- Structure charts
- State-Transition Diagrams

Structure charts

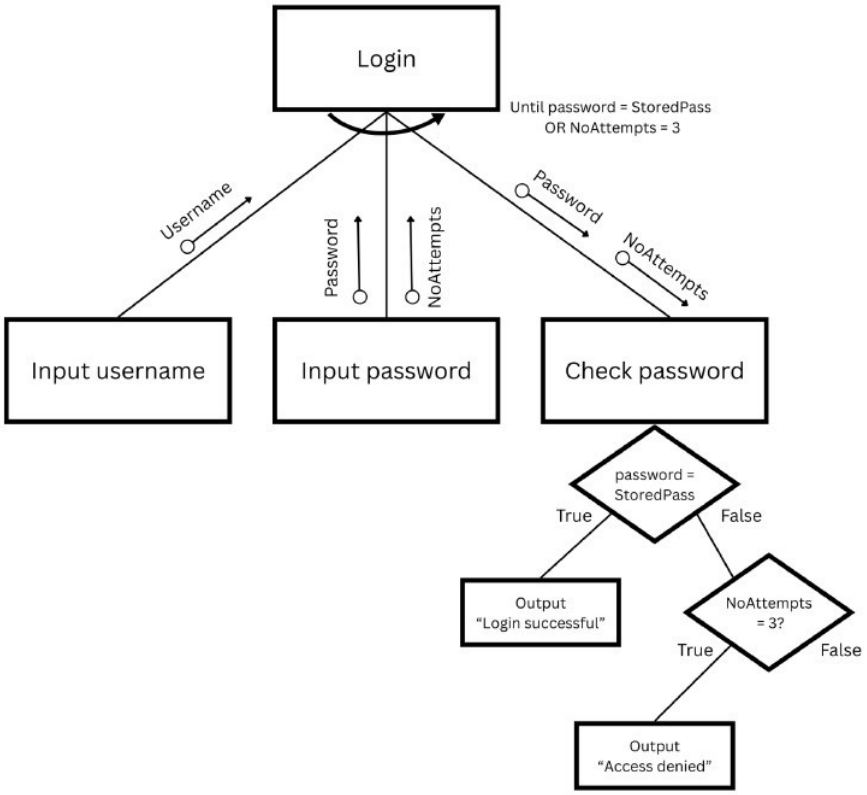
Structure charts

What is a structure chart?

- A structure chart is a **modelling tool** used in the **design stage** of program development
- It helps to **decompose a problem** into smaller, manageable **sub-tasks**, representing each as a **module**
- Structure charts focus on **what the program does**, not how it does it

Example

- A program that asks a user to enter their username and password
- Checks if the password matches the stored password and uses a counter to track how many times it is checked
- If the password matches, a successful message is displayed, else after three failed attempts a 'denied' message is displayed
- A structure chart might look like:



Key features

Feature	Explanation
Top-down design	The chart starts with the **main program** and breaks it into sub-modules
Decomposition	Each module represents a **specific task** or **function**
Module boxes	Each module is shown as a **box**
Vertical lines	Show **control flow** – which module calls which
Parameter arrows	**Arrows** pointing into modules show **data being passed in or returned**

Stepwise refinement	Each level adds **more detail** to the task above
Diamond shape	Shows a **condition** that could be **true or false**
Semi-circular arrow	Indicates **repetition**

Pseudocode from structure chart

- The first step is to create an **identifier table**

Identifier	Data type	Description
Username	STRING	Stores the username entered by the user
Password	STRING	Stores the password entered by the user
StoredPass	STRING	The correct password stored in the system for comparison
NoAttempts	INTEGER	Counts the number of failed login attempts
LoginSuccess	BOOLEAN	Indicates whether the login was successful

- Next, identify is any **functions** or **procedures** could be used

Module	Type	Purpose
GetCredentials	**PROCEDURE**	Asks the user to input username and password
CheckPassword	**FUNCTION**	Compares input password with stored password and returns TRUE/FALSE
ShowAccessMessage	**PROCEDURE**	Displays success or failure message

- Any finally, write the pseudocode

```
DECLARE Username : STRING
DECLARE Password : STRING
DECLARE StoredPass : STRING
DECLARE NoAttempts : INTEGER
DECLARE LoginSuccess : BOOLEAN

StoredPass ← "admin123"
NoAttempts ← 0
LoginSuccess ← FALSE
```

```
PROCEDURE GetCredentials()
   OUTPUT "Enter username: "
   INPUT Username
   OUTPUT "Enter password: "
   INPUT Password
ENDPROCEDURE

FUNCTION CheckPassword(P : STRING) RETURNS BOOLEAN
   IF P = StoredPass THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   ENDIF
ENDFUNCTION

PROCEDURE ShowAccessMessage(Success : BOOLEAN)
   IF Success = TRUE THEN
      OUTPUT "Login successful"
   ELSE
      OUTPUT "Access denied"
   ENDIF
ENDPROCEDURE

WHILE NoAttempts < 3 AND LoginSuccess = FALSE
   CALL GetCredentials()
   LoginSuccess ← CheckPassword(Password)

   IF LoginSuccess = FALSE THEN
      OUTPUT "Incorrect password"
      NoAttempts ← NoAttempts + 1
   ENDIF
ENDWHILE

CALL ShowAccessMessage(LoginSuccess)
```

State-Transition Diagrams

State-Transition Diagrams

What are state-transition diagrams?

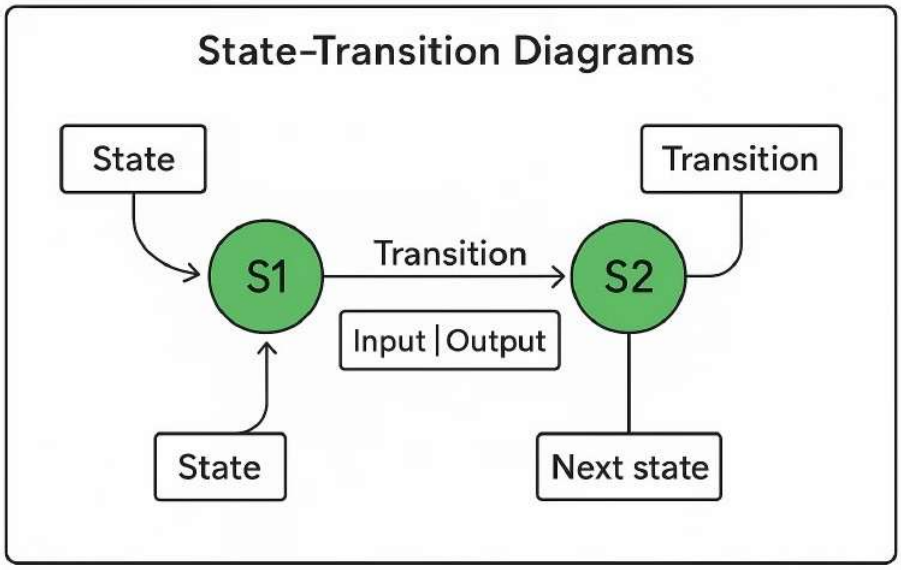
- A state-transition diagram shows **how a system moves between different states** based on inputs
- Each **state** represents a condition or situation of the system
- **Transitions** are arrows that show movement between states when certain inputs occur
- Transitions may also produce **outputs**, shown as labels on the arrows
- These diagrams model **finite-state machines (FSMs)**, which are widely used in computing

Key elements

Element	Description
State	Circle labelled with a name (e.g. S1, S2) showing a situation the system can be in
Initial state	Starting point, sometimes marked with an arrow pointing to it
Transition	Arrow from one state to another, labelled with input and output
Input	Event or data that causes a transition (e.g. Button-Y pressed)
Output	Action or signal produced by a transition (e.g. Output-A)
Next state	The state the system moves into after the transition

How to read labels on transitions

- Labels are often written as **Input | Output**
- If no output is produced, "none" may be written
- Example: Button-Y | Output-B means pressing Button-Y in the current state produces Output-B and causes a transition to the next state
- The diagram below shows the key features of a state-transition diagram:



- States are circles labelled S1, S2, etc
- Transitions are arrows between states, labelled with Input | Output
- The initial state is marked with an incoming arrow

Completing transition tables

- Exam questions often ask you to convert a diagram into a **table**
- The table normally has columns for **Current state, Input, Output, Next state**
- Work through the diagram row by row, following the arrows for each possible input

Example table structure:

Current state	Input	Output	Next state
S1	Button-Y	Output-A	S2
S1	Button-Z	Output-B	S3

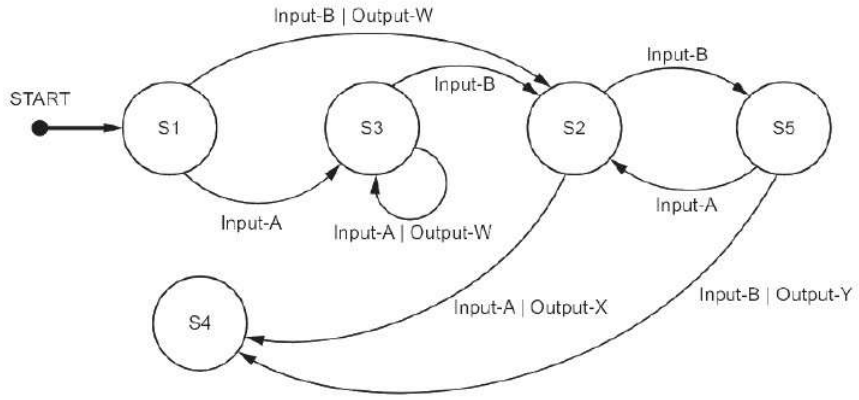
Examiner Tips and Tricks

- Always identify the **initial state** before tracing
- Highlight or mark inputs and outputs on your diagram to avoid missing one
- For tables, keep the order consistent: Input → Output → Next state
- Use arrows systematically, avoid guessing paths

- Remember: more than one sequence can sometimes reach the same state, but exams often ask for the **minimum changes**

Worked Example

Part of a program is represented by the following state-transition diagram



A. Complete the table to show the inputs, outputs and next states.

Assume that the current state for each row is given by the 'Next state' on the previous row. For example, the first Input-A is made when in state S1.

If there is no output for a given transition, then the output cell should contain 'none'.

The first two rows have been completed.

Input	Output	Next state
		S1
Input-A	none	S3
	Output-W	
	none	
Input-B		
Input-A		
		S4

[5]

B. Identify the input sequence that will cause the minimum number of state changes in the transition from S1 to S4

[1]

Answers

A.

Input	Output	Next state
		S1
Input-A	none	S3
Input-A	Output-W	S3
Input-B	none	S2
Input-B	none	S5
Input-A	none	S2
Input-A	Output-X	S4

- One mark per row 3 to 7 [5 marks]

B. Input-B, Input-A [1 mark]

Program Testing & Maintenance

✳ Faults & errors ✳ Testing methods ✳ Maintenance

Faults & errors

Error types

What are the common types of error?

- Designing algorithms is a skill that must be developed and when designing algorithms, **mistakes and issues will occur**
- **Trace tables** can also help to find any kind of error in a program or algorithm
- There are three main categories of errors that when designing algorithms a programmer must be able to identify & fix, they are:
  - **Syntax errors**
  - **Logic errors**
  - **Runtime errors**

Syntax errors

What is a syntax error?

- A syntax error is **an error that breaks the grammatical rules of a programming language** and **stops it from running**
- Examples of syntax errors are:
  - **Typos and spelling errors**
  - **Missing or extra brackets or quotes**
  - **Misplaced or missing semicolons**
  - **Invalid variable or function names**
  - **Incorrect use of operators**
  - **Incorrectly nested loops & blocks of code**

Pseudocode – with syntax errors

```
FUNCTION generate_username(first_name STRING, last_name STRING) RETURNS STRING
   DECLARE username : STRING
   username = SUBSTRING(first_name, 1, 1) + SUBSTRING(last_name, 1, 1) + SUBSTRING(last_name, 1, 3)
   RETURN username
ENDFUNCTION

PROCEDURE main()
   DECLARE first_name : STRING
   DECLARE last_name : STRING
   DECLARE username : STRING
```

```
    OUTPUT "Enter your first name: "
    INPUT first_name

    OUTPUT "Enter your last name: "
    INPUT last_name

    username ← generate_username[first_name, last_name]

    OUTPUT "Here's a suggested username:" + username
ENDPROCEDURE

// Main program
CALL Main
```

Pseudocode – without syntax errors

```
FUNCTION generate_username(first_name : STRING, last_name : STRING) RETURNS STRING
    DECLARE username : STRING
    username ← SUBSTRING(first_name, 1, 1) + SUBSTRING(last_name, 1, 1) + SUBSTRING(last_name, 1, 3)
    RETURN username
ENDFUNCTION

PROCEDURE main()
    DECLARE first_name : STRING
    DECLARE last_name : STRING
    DECLARE username : STRING

    OUTPUT "Enter your first name: "
    INPUT first_name

    OUTPUT "Enter your last name: "
    INPUT last_name

    username ← generate_username(first_name, last_name)

    OUTPUT "Here's a suggested username: ", username
ENDPROCEDURE

// Main program
CALL main()
```

Syntax errors

- FUNCTION generate_username(first_name STRING...)
  - **Missing colon** : between parameter name and type
  - first_name : STRING and last_name : STRING
- username = SUBSTRING(...)
  - **Uses =** instead of correct assignment operator ←
  - Should be username ← ...
- username ← generate_username[first_name, last_name]

- **Incorrect use of square brackets []** instead of parentheses () for function call
- Should be `generate_username(first_name, last_name)`

- `CALL Main`
  - **Incorrect capitalisation and missing parentheses** — procedure name is case-sensitive
  - Should be `CALL main()`

Logic errors

What is a logic error?

- A logic error is where **incorrect code is used that causes the program to run**, but **produces an incorrect output or result**
- Logic errors can be difficult to identify by the person who wrote the program, so one method of finding them is to use '**Trace Tables**'
- Examples of logic errors are:
  - **Incorrect use of operators (< and >)**
  - **Logical operator confusion (AND for OR)**
  - **Looping one extra time**
  - **Indexing arrays incorrectly (arrays indexing starts from 0)**
  - **Using variables before they are assigned**
  - **Infinite loops**

Pseudocode

```
FUNCTION calculate_area(length : REAL, width : REAL) RETURNS REAL
   // Checks for valid dimensions and calculates area
   IF length > 0 OR width > 0 THEN
      DECLARE area : REAL
      area ← width - length
      RETURN area
   ELSE
      OUTPUT "Length and width must be positive values."
   ENDIF
ENDFUNCTION

PROCEDURE main()
   DECLARE length : REAL
   DECLARE width : REAL
   DECLARE area : REAL

   OUTPUT "Enter the length of the rectangle: "
   INPUT length
```

```
    OUTPUT "Enter the width of the rectangle: "
    INPUT width

    area ← calculate_area(length, width)

    OUTPUT "The area of the rectangle is approximately ", area, " square units."
ENDPROCEDURE

// Main program
CALL main()
```

Logic errors

Line	Error	Correction
IF length > 0 OR width > 0 THEN	The check allows one value to be negative, which is not valid	Should be IF length > 0 AND width > 0 THEN
area ← width - length	Incorrect calculation – subtracts instead of multiplying to find the area	Should be area ← length * width

Runtime errors

What is a runtime error?

- A runtime error is where **an error causes a program to crash**
- Examples of runtime errors are:
  - **Dividing a number by 0**
  - **Index out of the range of an array**
  - **Unable to read or write a drive**

Pseudocode

```
FUNCTION calculate_area(length : REAL, width : REAL) RETURNS REAL
    // Calculates area of a rectangle after input validation
    IF length < 0 OR width < 0 THEN
        OUTPUT "Length and width must be positive values."
    ENDIF

    DECLARE area : REAL
    area ← length * width
    RETURN area
ENDFUNCTION

PROCEDURE main()
    DECLARE length : REAL
    DECLARE width : REAL
    DECLARE area : REAL
```

```
    OUTPUT "Enter the length of the rectangle: "
    INPUT length

    OUTPUT "Enter the width of the rectangle: "
    INPUT width

    area ← calculate_area(length, width)

    OUTPUT "The area of the rectangle is approximately ", area, " square units."
ENDPROCEDURE

// Main program
CALL main()
```

Runtime error

Line	Error	Correction
area ← calculate_area(length)	Only one parameter is passed instead of two – will crash at runtime	Should be area ← calculate_area(length, width)

Worked Example

Each pseudocode statement in the following table may contain an error due to the incorrect use of the function or operator.

Describe the error in each case, or write 'NO ERROR' if the statement contains no error.

You can assume that none of the variables referenced are of an incorrect type. **[5]**

Statement	Error
Result ← 2 & 4	
SubString ← MID("pseudocode", 4, 1)	
IF x = 3 OR 4 THEN	
Result ← Status AND INT(x/2)	
Message ← "Done" + LENGTH(MyString)	

Answer

Statement	Error
Result ← 2 & 4	Should be arithmetic

	operator (not &) **OR** 2 and 4 should be CHAR / STRING
SubString ← MID("pseudocode", 4, 1)	NO ERROR
IF x = 3 OR 4 THEN	Not Boolean values / incorrect operator **OR** Condition incorrect
Result ← Status AND INT(x/2)	INT(X/2) doesn't evaluate to a Boolean value / incorrect operator
Message ← "Done" + LENGTH(MyString)	Can't add string to number / "Done" is not a number

Testing methods

Testing methods

What are the different methods of testing?

- Testing takes place during the **testing stage** of the **program development life cycle**, once the program has been written and compiled
  - **Syntax errors** are usually detected during compilation
  - Testing focuses on checking for **logic errors**, **unexpected behaviour**, and **operational integrity**
  - A detailed **test plan**, outlining test cases and expected results, should be created during the **analysis stage**

Sample test plan

- Scenario: A program accepts a user's age between 0 and 120 inclusive

Test No.	Description	Input	Expected output	Type of test data	Actual outcome
1	Valid age within accepted range	25	"Age accepted"	Normal	
2	Input is not a number	"twenty"	"Invalid input: enter a number"	Abnormal	
3	Age above the maximum allowed	130	"Age out of range"	Extreme	
4	Age exactly at the upper boundary	120	"Age accepted"	Boundary	
5	Age exactly at the lower boundary	0	"Age accepted"	Boundary	
6	Age just below lower boundary	-1	"Age out of range"	Extreme	
7	Blank input	(blank)	"Please enter your age"	Abnormal	

Common methods of testing

Testing method	Description
Dry run	**Manually trace through** the code (e.g. on paper) to predict output and track variables
Walkthrough	Step-by-step review of the code **with others** to identify issues early
White-box testing	Tests the **internal logic** and code structure; the tester knows how the program works
Black-box testing	Tests the **inputs and expected outputs** without knowing the internal workings
Integration testing	Checks that different modules or components **work correctly together**
Alpha testing	Performed **in-house** by the developers during early testing
Beta testing	Carried out by **external users** in real-world environments
Acceptance testing	Final check to ensure the program meets the original **client requirements**
Stub testing	Uses temporary modules (stubs) to **simulate missing components** during early testing

Choosing suitable test data

What is suitable test data?

- Suitable test data is **specially chosen to test the functionality** of a program or design
- Developers or test-users would pick **a selection of test data** from the following categories
  - **Normal**
  - **Abnormal**
  - **Extreme**
  - **Boundary**
- The results would be **compared to the expected results** to check if the algorithm/program works as intended
- The results would be **stored in the test plan**
- Each category is explained within the context of a simple pseudocode program below

Pseudocode

```
DECLARE name : STRING
DECLARE age : INTEGER

OUTPUT "What is your name? "
INPUT name

OUTPUT "How old are you? "
INPUT age

IF age >= 12 AND age <= 18 THEN
   OUTPUT "Welcome, " + name + "! Your age is accepted."
ELSE
   OUTPUT "Sorry, " + name + ". Your age is not accepted."
ENDIF
```

Normal data

- Normal test data is data that **should be accepted** in the program
- An example would be a user entering their age as 16 into the age field of the program

Abnormal data

- Abnormal test data is data that is the **wrong data type**
- An example would be a user entering their age as "F" into the age field of the program

Extreme data

- Extreme test data is the **maximum** and **minimum** values of normal data that are **accepted** by the system
- An example would be a user entering their age as 18 or 12 into the age field of the program

Boundary data

- Boundary test data is similar to extreme data except that the values **on either side of** the **maximum** and **minimum** values are tested
- The largest and smallest **unacceptable values**
- An example would be a user entering their age as 11 or 19 into the age field of the program

Selecting suitable test data

Type of Test	Input	Expected Output
Normal	14	Accepted
Normal	16	Accepted

Extreme	12	Accepted
Extreme	18	Accepted
Abnormal	H	Rejected
Abnormal	@	Rejected
Boundary	11	Rejected
Boundary	19	Rejected

Maintenance

Types of maintenance

What is program maintenance?

- Program maintenance is the process of updating or improving a program **after it has been delivered** to the user
- Unlike physical equipment, programs **don't wear out**, but they may need to be changed due to **errors**, **changing requirements**, or **new technology**

Why maintenance is needed

- To **fix errors that were missed** during testing
- To **improve performance** based on user feedback
- To **adapt the software** to new uses or platforms

Types of program maintenance

Type	Purpose	Example
Corrective	Fixes bugs or errors found during real-world use	Fixing a bug that causes the program to crash when special characters are entered
Perfective	Improves performance or adds small enhancements	Replacing a loading screen with a progress bar to give better user feedback
Adaptive	Modifies the program to support new environments or requirements	Modifying the program to work on a tablet instead of just a desktop computer