Chapter 15:
Software development

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the purpose of a development life cycle
- show understanding of the need of different development life cycles depending on the program being developed
- describe the principles, benefits and drawbacks of each type of life cycle: waterfall, iterative and rapid application development
- show understanding of the analysis, design, coding, testing and maintenance stages in the program development cycle
- use a structure chart to decompose a problem into sub-tasks
- use a structure chart to 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
- show understanding of ways of exposing and avoiding faults in programs
- locate and identify the different types of errors (syntax errors, logic errors and run-time errors)
- correct identified errors
- show understanding of available testing methods (dry-run, walkthrough, white-box, black-box, integration, alpha, beta, acceptance, stub)
- select appropriate data for a given testing method
- show understanding of the need for a test strategy and test plan and their likely contents
- choose appropriate test data for a test plan (normal, abnormal, extreme/boundary)
- show understanding of the need for continuing maintenance of a system and the differences between each type of maintenance (corrective, adaptive, perfective)
- analyse an existing program and make amendments to enhance functionality.

15.01 Stages in the program development life cycle

Developing a program involves different stages. You solve a problem by designing the solution using Structured English, a flowchart and / or pseudocode (see Chapters 12 and 13). You write the program code and test it.

When large software systems are required to solve big problems, these stages are more formal, especially when more people are involved in the development. Before a solution can be designed, the problem needs to be analysed. When the program works and is being used, issues might arise that require changes. This is known as maintenance.

Analysis

The first step in solving a problem is to investigate the issues and the current system if there is one. The problem needs to be defined clearly and precisely. A 'requirements specification' is drawn up.

The next step is planning a solution. Sometimes there is more than one solution. You need to decide which is the most appropriate.

The third step is to decide how to solve the problem:

- bottom-up: start with a small sub-problem and then build on this
- top-down: stepwise refinement using pseudocode, flowcharts or structure charts.

Design

You have a solution in mind. How do you design the solution in detail? Chapter 12 (Section 12.05) showed that an identifier table is a good starting point. This leads you to thinking about data structures: do you need a 1D array or a 2D array to store data while it is processed? Do you need a file to store data long-term?

Plan your algorithm by drawing a flowchart or writing pseudocode.

Coding

When you have designed your solution, you might need to choose a suitable high-level programming language. If you know more than one programming language, you have to weigh up the pros and cons of each one. Looking at Chapter 14, you need to decide which programming language would best suit the problem you are trying to solve and which language you are most familiar with.

TIP

This stage is often referred to as implementation.

You implement your algorithm by converting your pseudocode into program code. When you start writing programs you might find it takes several attempts before the program compiles. When it finally does, you can execute it. It might 'crash', meaning that it stops working. In this case, you need to debug the code. The program might run and give you some output. This is the Eureka moment: 'It works!!!!'. But does the program do what it was meant to do?

Testing

Only thorough testing can ensure the program really works under all circumstances (see Sections 15.06 and 15.07).

There are several different development methodologies. These include the waterfall, the iterative and the rapid application development model.

Discussion Point:

Do you think that all programs can be totally error-free?

The program development life cycle

The program development life cycle follows the defined stages of analysis, design, coding (implementation), testing and maintenance. When maintenance no longer results in a program fit for purpose, the development starts again, therefore creating a cycle (see Figure 15.01).

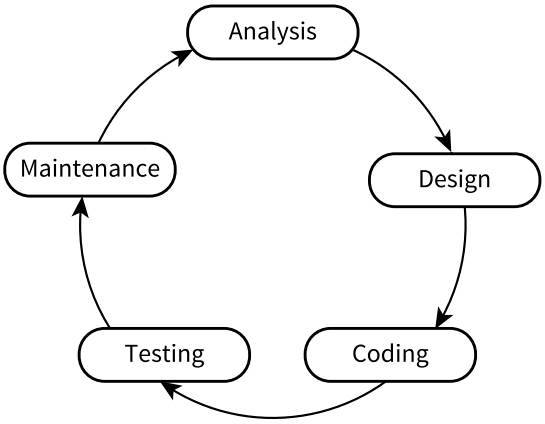


Figure 15.01 The program development life cycle

The waterfall model

Figure 15.02 shows the waterfall model.

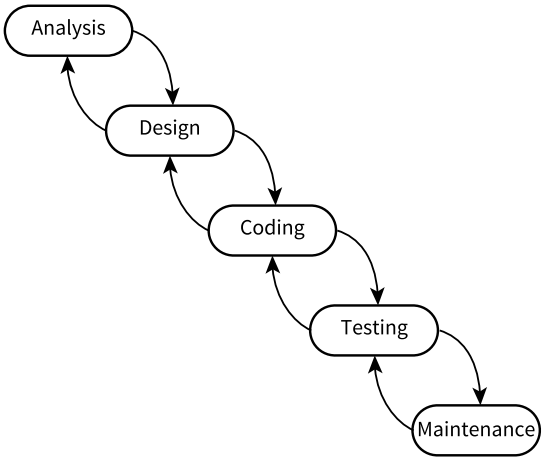


Figure 15.02 The waterfall model

The arrows going down represent the fact that the results from one stage are input into the next stage. The arrows leading back up to an earlier stage reflect the fact that often more work is required at an earlier stage to complete the current stage.

Benefits include the following.

- Simple to understand as the stages are clearly defined.
- Easy to manage due to the fixed stages in the model. Each stage has specific outcomes.
- Stages are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Drawbacks include the following.

- No working software is produced until late during the life cycle.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Cannot accommodate changing requirements.

- It is difficult to measure progress within stages.
- Integration is done at the very end, which doesn't allow identifying potential technical or business issues early.

The iterative model

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development starts with the implementation of a small subset of the program requirements. Repeated (iterative) reviews to identify further requirements eventually result in the complete system.

Benefits include the following.

- There is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development means corrective measures can be taken more quickly.
- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging of a smaller subset of program is easy.
- Risks are identified and resolved during iteration.
- Easier to manage risk – high-risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilised/applied to the next increment.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early, which facilitates customer evaluation and feedback.

Drawbacks include the following.

- Only large software development projects can benefit because it is hard to break a small software system into further small serviceable modules.
- More resources may be required.
- Design issues might arise because not all requirements are gathered at the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.

The Rapid Application Development (RAD) model

RAD is a software development methodology that uses minimal planning. Instead it uses prototyping. A prototype is a working model of part of the solution.

In the RAD model, the modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. There is no detailed preplanning. Changes are made during the development process.

The analysis, design, code and test phases are incorporated into a series of short, iterative development cycles.

Benefits include the following.

- Changing requirements can be accommodated.
- Progress can be measured.
- Productivity increases with fewer people in a short time.
- Reduces development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

Drawbacks include the following.

- Only systems that can be modularised can be built using RAD.
- Requires highly skilled developers/designers.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for projects requiring shorter development times.

15.02 Program design using structure charts

An alternative approach to modular design is to choose the sub-tasks and then construct a **structure chart** to show the interrelations between the modules. Each box of the structure chart represents a module. Each level is a refinement of the level above.

A structure chart also shows the interface between modules, the variables. These variables are referred to as ‘parameters’ (see Section 14.10). A **parameter** supplying a value to a lower-level module is shown as a downwards pointing arrow. A parameter supplying a new value to the module at the next higher level is shown as an upward pointing arrow.

Figure 15.03 shows a structure chart for a module that calculates the average of two numbers. The top-level box is the name of the module, which is refined into the three sub-tasks of Level 1. The input numbers (parameters Number1 and Number2) are passed into the ‘Calculate Average’ sub-task and then the Average parameter is passed into the ‘OUTPUT Average’ sub-task. The arrows show how the parameters are passed between the modules. This parameter passing is known as the ‘interface’.

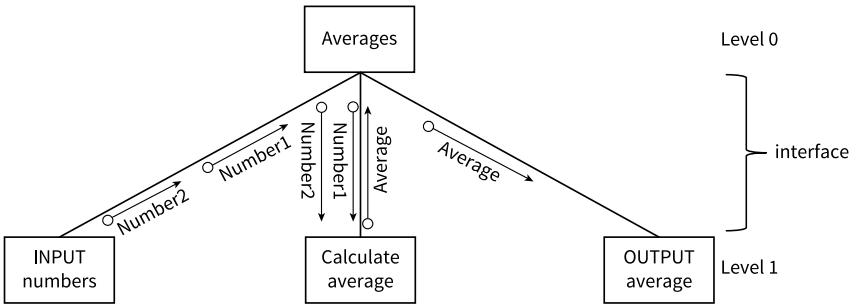


Figure 15.03 Structure chart for a module that calculates the average of two numbers

TASK 15.01

Draw a structure chart for the following module: Input a number of km, output the equivalent number of miles.

Structure charts can also show control information: selection and repetition.

The simple number-guessing game that was introduced in Chapter 12 (Section 12.06) could be modularised and presented as a structure chart, as shown in Figure 15.04.

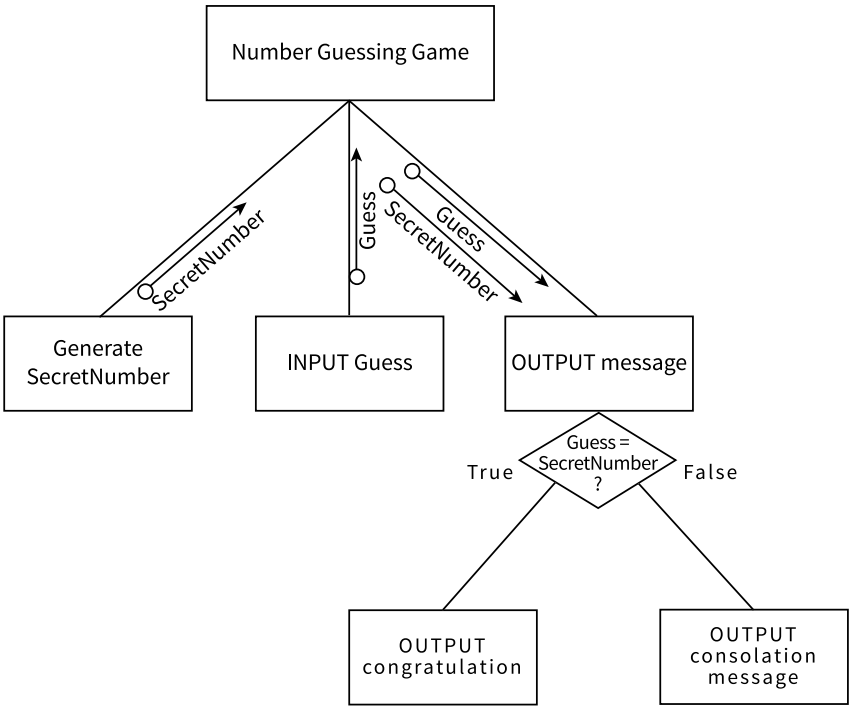


Figure 15.04 Structure chart for number-guessing game with only one guess allowed

The diamond shape shows a condition that is either True or False. So either one branch or the other will be followed.

Figure 15.05 shows the structure chart for the pyramid-drawing program from Worked Example 12.10. The semi-circular arrow represents repetition of the modules below the arrow. The label shows the condition when repetition occurs.

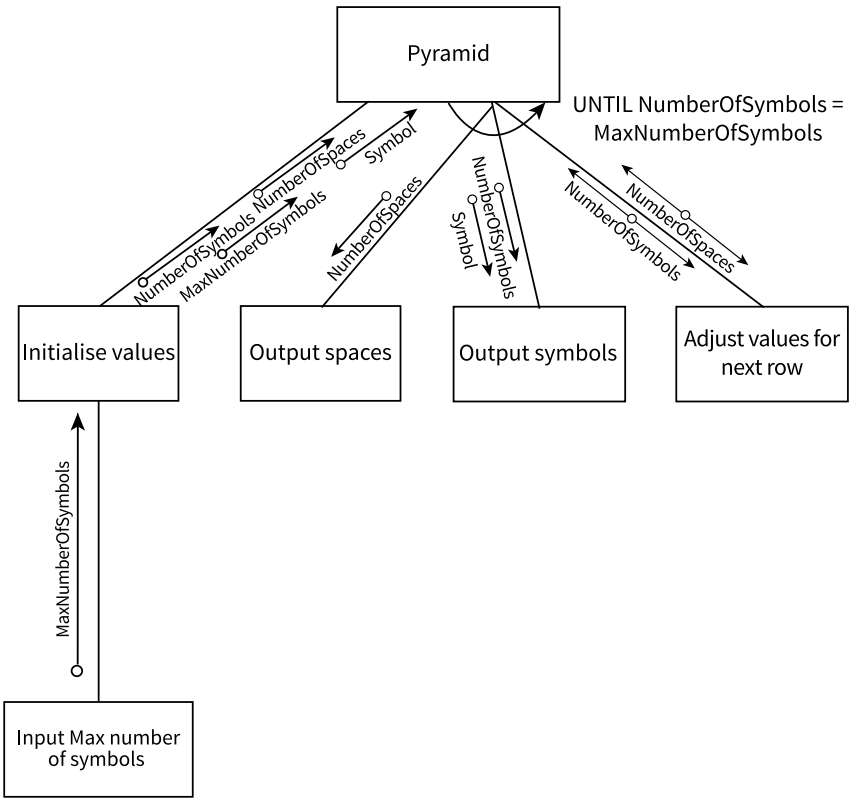


Figure 15.05 Structure chart for pyramid-drawing program

TASK 15.02

Amend the structure chart for the number-guessing game (Figure 15.04) to include repeated guesses until the player guesses the secret number. The output should include the number of

guesses made.

TASK 15.03

Draw a structure chart for the following problem: A user attempts to log on with a user ID. User IDs and passwords are stored in two 1D arrays (lists). The algorithm searches the list of user IDs and looks up the password in the password list. The user is given three chances to input the correct password. If the correct password is entered, a suitable message is output. If the third attempt is incorrect, a warning message is output.

Structure charts help programmers to visualise how modules are interrelated and how they interface with each other. When looking at a larger problem this becomes even more important. Figure 15.06 shows a structure chart for the Connect 4 program (Task 13.06). It uses the following symbols:

- An arrow with a solid round end ●⟶ shows that the value transferred is a flag (a Boolean value)
- A double-headed arrow ⟷ shows that the variable value is updated within the module.

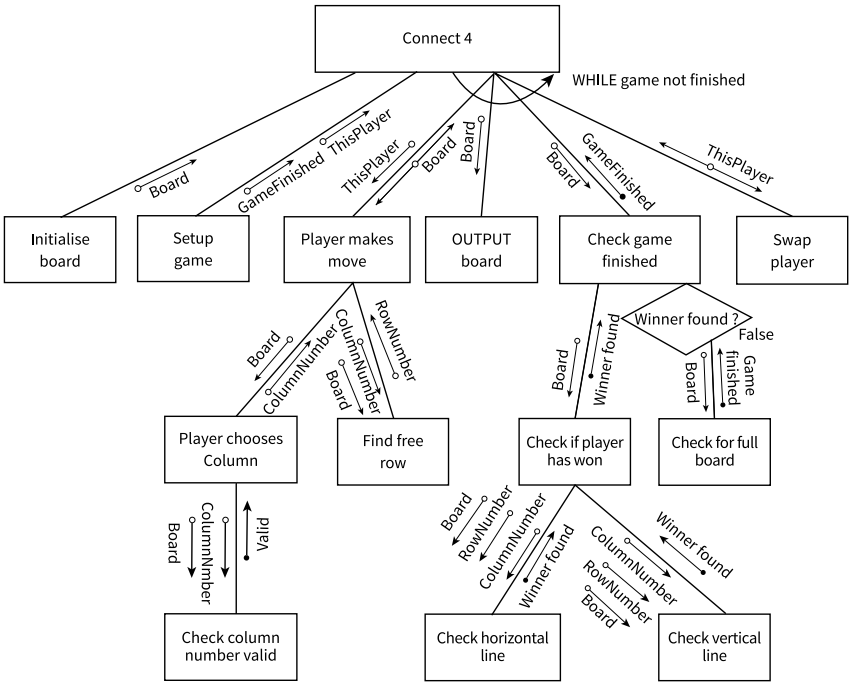


Figure 15.06 Structure chart for the Connect 4 game

15.03 Deriving pseudocode from a structure chart

Let's look at the pyramid problem again (Figure 15.05). In Worked Example 12.10, a modular solution was created without using a structure chart and all variables were global. Now we are going to use local variables and parameters. The reason for using local variables and parameters is that modules are then self-contained and any changes to variables do not have accidental effects on a variable value elsewhere.

The top-level module, Pyramid, calls four modules. When a module is called, we supply the parameters in parentheses after the module identifier. This gives the following pseudocode:

```
MODULE Pyramid
    CALL SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    REPEAT
        CALL OutputSpaces(NumberOfSpaces)
        CALL OutputSymbols(NumberOfSymbols, Symbol)
        CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    UNTIL NumberOfSymbols > MaxNumberOfSymbols
ENDMODULE

PROCEDURE SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    INPUT Symbol
    CALL InputMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE

PROCEDURE InputMaxNumberOfSymbols(MaxNumberOfSymbols)
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE

PROCEDURE OutputSpaces(NumberOfSpaces)
    FOR Count ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count
ENDPROCEDURE

PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
    FOR Count← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT Newline // move to the next line
ENDPROCEDURE

PROCEDURE AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    NumberOfSpaces ← NumberOfSpaces – 1
    NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

Note that a structure chart does not give details about how parameters are passed: by reference or by value.

TASK 15.04

1. Write pseudocode to implement the structure chart from Figure 12.03 (for the average of two numbers).
2. Write pseudocode to implement the structure chart from Figure 12.04 (for the number-guessing game).
3. Amend the pseudocode from Worked Example 13.05 to implement the interface shown in the structure chart from Figure 15.06.

Discussion Point:

The full rules of Connect 4 are that a diagonal of four tokens also is a winning line. Where in Figure

15.06 should the module to check for a diagonal be added? What parameters are required for this module? Does this additional module require further stepwise refinement?

15.04 Program design using state-transition diagrams

A computer system can be seen as a **finite state machine (FSM)**. An FSM has a start state. An input to the FSM produces a transformation from one state to another state.

The information about the states of an FSM can be presented in a **state-transition table**.

Table 15.01 shows an example FSM represented as a state-transition table.

- If the FSM is in state S1, an input of a causes no change of state.
- If the FSM is in state S1, an input of b transforms S1 to S2.
- If the FSM is in state S2, an input of b causes no change of state.
- If the FSM is in state S2, an input of a transforms S2 to S1.

A **state-transition diagram** can be used to describe the behaviour of an FSM. Figure 15.07 shows the start state as S1 (denoted by ●⟶). If the FSM has a final state (also known as the halting state), this is shown by a double-circled state (S1 in the example).

		current state	
		S1	**S2**
input	**a**	S1	S1
	b	S2	S2

Table 15.01 State-transition table

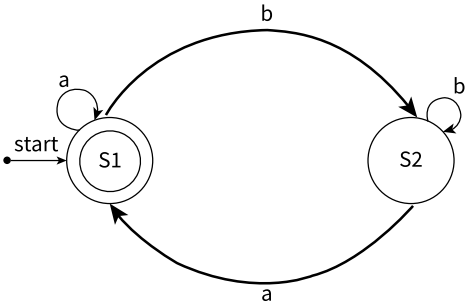


Figure 15.07 State-transition diagram

If an input causes an output this is shown by a vertical bar (as in Figure 15.08). For example, if the current state is S1, an input of b produces output c and transforms the FSM to state S2.

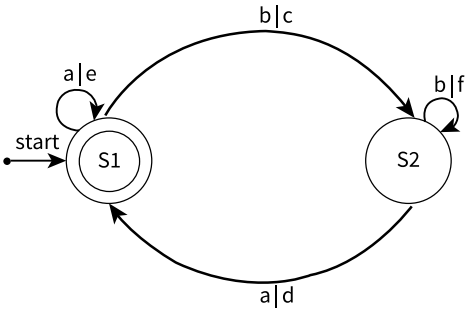


Figure 15.08 State-transition diagram with outputs

A Finite State Machine with outputs is also known as a Mealy Machine.

WORKED EXAMPLE 15.01

Creating a state-transition diagram for an intruder detection system

A program is required that simulates the behaviour of an intruder detection system.

Description of the system: The system has a battery power supply. The system is activated when the start button is pressed. Pressing the start button when the system is active has no effect. To de-activate the system, the operator must enter a PIN. The system goes into alert mode when a sensor is activated. The system will stay in alert mode for two minutes. If the system has not been de-activated within two minutes an alarm bell will ring.

We can complete a state-transition table (Table 15.02) using the information from the system description.

Current state	Event	Next state
System inactive	Press start button	System active
System active	Enter PIN	System inactive
System active	Activate sensor	Alert mode
System active	Press start button	System active
Alert mode	Enter PIN	System inactive
Alert mode	2 minutes pass	Alarm bell ringing
Alert mode	Press start button	Alert mode
Alarm bell ringing	Enter PIN	System inactive
Alarm bell ringing	Press start button	Alarm bell ringing

Table 15.02 State-transition table for intruder detection simulation

The start state is ‘System inactive’. We can draw a state-transition diagram (Figure 15.09) from the information in Table 15.02.

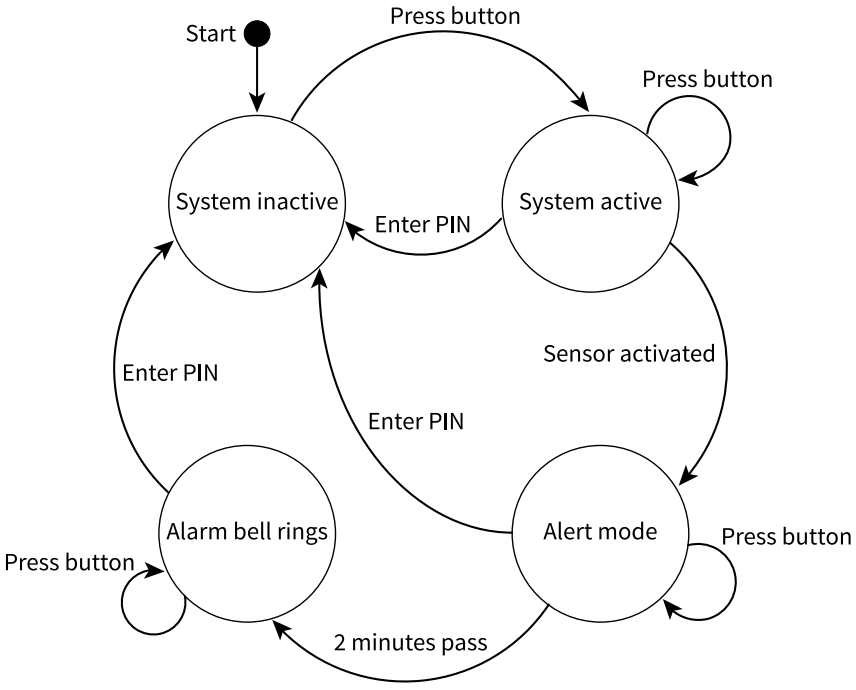


Figure 15.09 State-transition diagram for intruder detection system

WORKED EXAMPLE 15.02

Creating a state-transition diagram for a two’s complement FSM

A finite state machine has been designed that will take as input a positive binary integer, one bit at

a time, starting with the least significant bit. The FSM converts the binary integer into the two's complement negative equivalent. The method to be used is as follows.

1 Output the bits input up to and including the first 1.

2 Output the other bits following this scheme:

2.1 For each 1, output a 0.

2.2 For each 0, output a 1.

This information is represented in the state-transition table shown in Table 15.03.

Current state	S1	S1	S2	S2
Input bit	0	1	0	1
Next state	S1	S2	S2	S2
Output bit	0	1	1	0

Table 15.03 State-transition table with outputs

This method can be represented as the state-transition diagram in Figure 15.10.

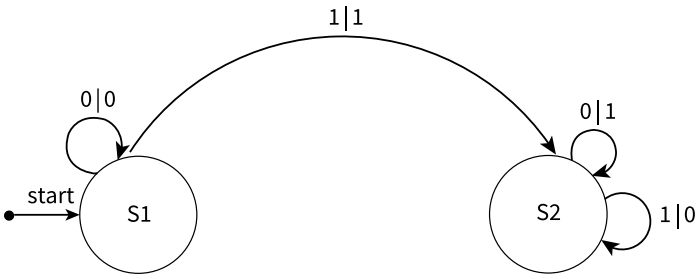


Figure 15.10 State-transition diagram for a two's complement FSM

TASK 15.05

Write a program that simulates the intruder detection system in Worked Example 15.01.

Question 15.01

What is the output from the FSM represented by the state-transition diagram in Figure 15.10, when the input is `0101`?

Extension Question 15.01

Does the FSM in Figure 15.10 work for converting a negative binary number into its positive equivalent?

15.05 Types of error

Why errors occur and how to find them

Software may not perform as expected for a number of reasons, such as:

- the programmer has made a coding mistake
- the requirement specification was not drawn up correctly
- the software designer has made a design error
- the user interface is poorly designed, and the user makes mistakes
- computer hardware experiences failure.

How are errors found? The end user might report an error. This is not good for the reputation of the software developer. Testing software before it is released for general use is essential. Research has shown that the earlier an error can be found, the cheaper it is to fix it. It is very important that software is tested throughout its development.

The purpose of testing is to discover errors. Edsger Dijkstra, a famous Dutch computer scientist, said 'Program testing can be used to show the presence of bugs, but never to show their absence!'.

Finding syntax errors is easy. The compiler/interpreter will find them for you and usually gives you a hint as to what is wrong.

Depending on your development environment editor, some syntax errors may be flagged up by your editor, so you can correct these as you go along. A **syntax error** is a 'grammatical' error, in which a program statement does not follow the rules of the high-level language constructs.

Some syntax errors might only become apparent when you are using an interpreter or compiler to translate your program. Interpreters and compilers work differently (see Chapter 8, Section 8.05, and Chapter 20, Section 20.06). When a program compiles successfully, you know there will be no syntax errors remaining.

This is not the case with interpreted programs. Only statements that are about to be executed will be syntax checked. So, if your program has not been thoroughly tested, it might even have syntax errors remaining.

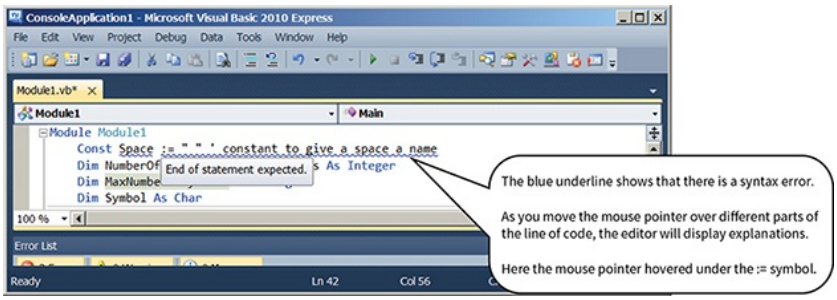


Figure 15.11 Syntax error in a Visual Basic program

Figure 15.11 gives an example of how a compiler flags a syntax error. The compiler stops when it first notices a syntax error. The error is often on the previous line. The compiler can't tell until it gets to the next line of code and finds an unexpected keyword.

Much more difficult to find are **logic errors** and **run-time errors**. A run-time error occurs when program execution comes to an unexpected halt or 'crash' or it goes into an infinite loop and 'freezes'.

Both of these types of error can only be found by careful testing. The danger of such errors is that they may only show up under certain circumstances. If a program crashes every time it is executed, it is obvious there is an error. If the program is used frequently and appears to work until a certain set of data causes a malfunction, that is much more difficult to discover without perhaps serious consequences.

15.06 Testing methods

Stub testing

When you develop a user interface, you might wish to test it before you have implemented all the facilities. You can write a ‘stub’ for each procedure (see Figure 15.12). The procedure body only contains an output statement to acknowledge that the call was made. Each option the user chooses in the main program will call the relevant procedure.

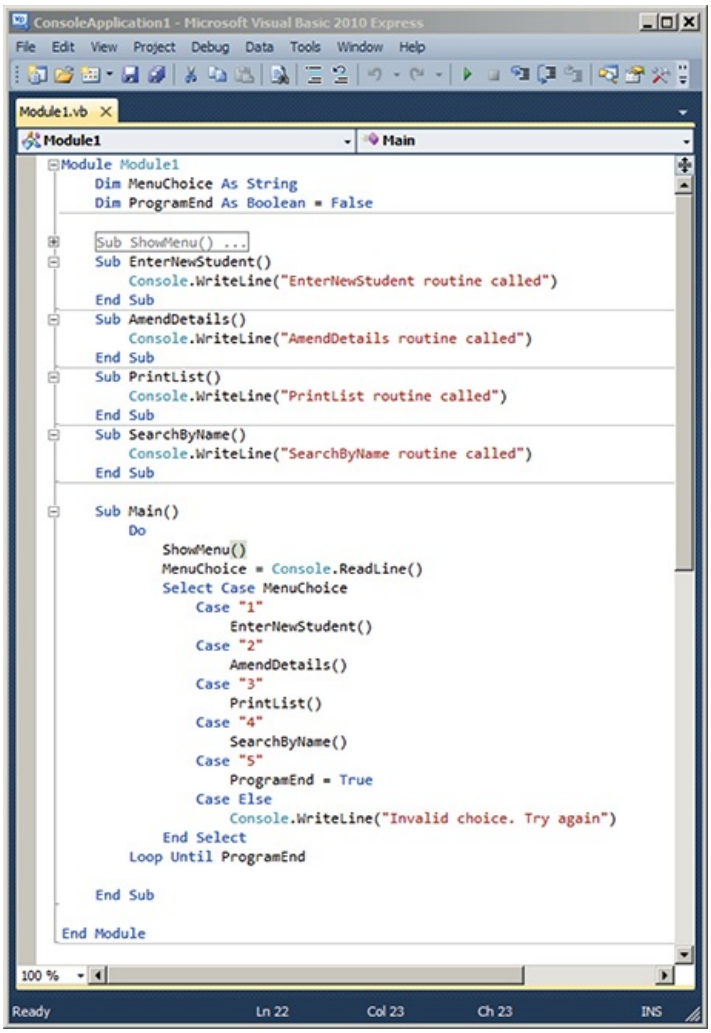


Figure 15.12 VB.NET stub testing

Black-box testing

As the programmer, you can see your program code and your testing will involve knowledge of the code (see white-box testing).

As part of thorough testing, a program should also be tested by other people, who do not see the program code and don’t know how the solution was coded.

Such program testers will look at the program specification to see what the program is meant to do, devise **test data** and work out expected results. Test data usually consists of normal data values, extreme/boundary data values and erroneous/abnormal data values.

The tester then runs the program with the test data and records their results. This method of testing is called **black-box testing** because the tester can’t see inside the program code: the program is a ‘black box’.

Where the actual results don’t match the expected results, a problem exists. The programmer needs to find the reason for this discrepancy before correcting the program (see Section 15.08). Once black-box testing has established that there is an error, debugging software or dry-running have to be used to find

the lines of code that need correcting.

White-box testing

How can we check that code works correctly? We choose suitable test data that checks every path through the code. This is called **white-box testing**.

WORKED EXAMPLE 15.03

White-box testing of pseudocode

This is the pseudocode from Worked Example 12.02 in Chapter 12:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN            // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE            // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

To test it, we need four sets of numbers with the following characteristics.

- The first number is the largest.
- The first number is larger than the second number; the third number is the largest.
- The second number is the largest.
- The second number is larger than the first number; the third number is the largest.

Note that it does not matter what exact values are chosen as test data. The important point is that the values differ in such a way that each part of the nested `IF` statement is checked. Table 15.04 lists four sets of test data and the results from them. The parts of the algorithm not entered for a particular set of data are greyed out. This makes it easier to see that each part has been checked after all four tests have been done.

Line of algorithm	Test 1	Test 2	Test 3	Test 4
`INPUT Number1`	15	12	12	8
`INPUT Number2`	12	8	15	12
`INPUT Number3`	8	15	8	15
`IF Number1 > Number2`	TRUE	TRUE	FALSE	FALSE
`THEN` `  IF Number1 > Number3`	TRUE	FALSE		
`  THEN`	Output 15			

OUTPUT Number1				

ELSE
OUTPUT Number3
ENDIF
		Output 15		

ELSE IF Number2 > Number3			TRUE	FALSE
THEN OUTPUT Number2			Output 15	

ELSE
OUTPUT Number3
ENDIF
ENDIF
				Output 15

Table 15.04 Testing the validity of the nested IF statement

Dry-running an algorithm

A good way of checking that an algorithm works as intended is to **dry-run** the algorithm using a **trace table** and different test data. This is also known as a **walk through**.

The idea is to write down the current contents of all variables and conditional values at each step of the algorithm.

WORKED EXAMPLE 15.04

Tracing an algorithm

Here is the algorithm of the number-guessing game:

```
SecretNumber ← 34
INPUT "Guess a number: " Guess
NumberOfGuesses ← 1
REPEAT
    IF Guess = SecretNumber
      THEN
        OUTPUT "You took ", NumberOfGuesses, " guesses”
      ELSE
        IF Guess > SecretNumber
          THEN
            INPUT "Guess a smaller number: " Guess
          ELSE
            INPUT "Guess a larger number: " Guess
        ENDIF
         NumberOfGuesses ← NumberOfGuesses + 1
    ENDIF
UNTIL Guess = SecretNumber
```

To test the algorithm, construct a trace table (Table 15.05) with one column for each variable used in the algorithm and also for the condition `Guess > SecretNumber`

Now carefully look at each step of the algorithm and record what happens. Note that we do not tend to write down values that don’t change. Here `SecretNumber` does not change after the initial assignment, so the column is left blank in subsequent rows.

SecretNumber	Guess	NumberOfGuesses	Guess > SecretNumber	Message
34	5	1	FALSE	...larger...
	55	2	TRUE	...smaller...

	30	3	FALSE	...larger...
	42	4	TRUE	...smaller...
	36	5	TRUE	...smaller...
	33	6	FALSE	...larger...
	34	7		... 7 guesses

Table 15.05 Trace table for number-guessing game

We only make an entry in a cell when an assignment occurs. Values remain in variables until they are overwritten. So a blank cell means that the value from the previous entry remains.

It is important to start filling in a new row in the trace table for each iteration (each time round the loop).

TIP

When learning to complete trace tables and to ensure you follow every line of code in the correct sequence, you can number the lines of the algorithm and add a column for the line numbers in your trace table (see Worked Example 15.05 Trace Table 15.06).

WORKED EXAMPLE 15.05

Tracing an algorithm

To test the improved algorithm of Worked Example 13.03 (bubble sort), dry-run the algorithm by completing the trace table (Table 15.06).

```
01 MaxIndex ← 7
02 n ← MaxIndex - 1
03 REPEAT
04     NoMoreSwaps ← TRUE
05     FOR j ← 1 TO n
06         IF MyList[j] > MyList[j + 1]
07           THEN
08             Temp ← MyList[j]
09             MyList[j] ← MyList[j + 1]
10             MyList[j + 1] ← Temp
11             NoMoreSwaps ← FALSE
12         ENDIF
13     NEXT j
14     n ← n – 1
15 UNTIL NoMoreSwaps = TRUE
```

Line Numbers	Max Index	n	No-MoreSwaps	j	MyList[j] > MyList[j + 1]	Temp	MyList						
							[1]	[2]	[3]	[4]	[5]	[6]	[7]
01, 02	7	6					5	34	98	7	41	19	25
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 07, 08, 09, 10				3	TRUE	98			7	98			
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				4	TRUE	98				41	98		
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				5	TRUE	98					19	98	
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				6	TRUE	98						25	98
11, 12			FALSE										
13, 14, 15		5											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 07, 08, 09, 10				2	TRUE	34		7	34				
11, 12			FALSE										
13, 05, 06, 12				3	FALSE								
13, 05, 06, 07, 08, 09, 10				4	TRUE	41				19	41		
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				5	TRUE	41					25	41	
11, 12			FALSE										
13, 14, 15		4											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 07, 08, 09, 10				3	TRUE	34			19	34			
11, 12			FALSE										
13, 05, 06, 07, 08, 09, 10				4	TRUE	34				25	34		
11, 12			FALSE										
13, 14, 15		3											
03, 04, 05, 06, 12			TRUE	1	FALSE								
13, 05, 06, 12				2	FALSE								
13, 05, 06, 12				3	FALSE								
13, 14, 15		2											

Table 15.06 Trace table for improved bubble sort algorithm

TASK 15.06

Design a trace table for the following algorithm:

```
FUNCTION ConvertFromHex(HexString : STRING) RETURNS INTEGER
    DECLARE ValueSoFar, HexValue, HexLength, i : INTEGER
    DECLARE HexDigit : CHAR
    ValueSoFar ← 0
    HexLength ← Length(HexString)
    FOR i ← 1 TO HexLength
        HexDigit ← HexString[i]
        CASE OF HexDigit
          'A': HexValue ← 10
          'B': HexValue ← 11
          'C': HexValue ← 12
          'D': HexValue ← 13
          'E': HexValue ← 14
          'F': HexValue ← 15
          OTHERWISE HexValue ← StringToInt(HexDigit)
        ENDCASE
        ValueSoFar ← ValueSoFar * 16 + HexValue
    NEXT i
    RETURN ValueSoFar
ENDFUNCTION
```

Dry-run the function call `ConvertFromHex('A5')` by completing the trace table.

These testing methods are used early on in software development, for example when individual modules are written. Sometimes programmers themselves use these testing methods. In larger software development organisations, separate software testers will be employed.

Discussion Point:

Do you think that a program tester will find errors the programmer did not know about? You can try out the idea by letting your friends test a program that you think works perfectly.

Software often consists of many modules, sometimes written by different programmers. Each individual module might have passed all the tests, but when modules are joined together into one program, it is vital that the whole program is tested. This is known as **integration testing**. Integration testing is usually done incrementally. This means that a module at a time is added and further testing is carried out before the next module is added.

Software will be tested in-house by software testers before being released to customers. This type of testing is called **alpha testing**.

Bespoke software (written for a specific customer) will then be released to the customer. The customer will check that it meets their requirements and works as expected. This stage is referred to as **acceptance testing**. It is generally part of the hand-over process. On successful acceptance testing, the customer will sign off the software.

When software is not bespoke but produced for general sale, there is no specific customer to perform acceptance testing and sign off the software. So, after alpha testing, a version is released to a limited audience of potential users, known as ‘beta testers’. These beta testers will use the software and test it in their own environments. This early release version is called a beta version and the chosen users perform **beta testing**. During beta testing, the users will feed back to the software house any problems they have found, so that the software house can correct any reported faults.

15.07 Test strategy, test plans and test data

During the design stage of a software project, a suitable testing strategy must be worked out to ensure rigorous testing of the software from the very beginning. Consideration should be given to which testing methods are appropriate for the project in question. A carefully designed test plan has to be produced.

It is important to recognise that large programs cannot be exhaustively tested but it is important that systematic testing finds as many errors as possible. We therefore need a test plan. In the first instance, an outline plan is designed, for example:

- flow of control: does the user get appropriate choices and does the chosen option go to the correct module?
- validation of input: has all data been entered into the system correctly?
- do loops and decisions perform correctly?
- is data saved into the correct files?
- does the system produce the correct results?

This outline test plan needs to be made into a detailed test plan.

How can we carry out these tests? We need to select data that will allow us to see whether it is handled correctly. This type of data is called ‘test data’. It differs from real, live data because it is specifically chosen with a view of testing different possibilities. We distinguish between different types of test data, listed in Table 15.07.

Type of test data	Explanation
Normal (valid)	Typical data values that are valid
Abnormal (erroneous)	Data values that the system should not accept
Boundary (extreme)	Data values that are at a boundary or an extreme end of the range of normal data; test data should include values just within the boundary (that is, valid data) and just outside the boundary (that is, invalid data)

Table 15.07 Types of test data

WORKED EXAMPLE 15.06

Designing test data

Look at the Pyramid Problem (code shown in Section 14.13). This is a simple program, but we can use it to illustrate how to choose test data. There are just two user inputs: the number of symbols that make up the base and the symbol that is to be used to construct the pyramid. Let’s consider just the test data for the number of symbols (Table 15.08).

Type of test data	Example test values	Explanation
Normal (valid)	7	7 is an odd integer, so should be accepted. Any odd positive integer would be suitable as test data. However, it should be bigger than 1 to check that the pyramid is correctly formed. More than one different value to test would be a good idea.
Abnormal (erroneous)		Any number that is not a positive odd integer. This will require several tests to ensure that the following types of data are not accepted:

	–7	• negative integer
	8	• even integer
	7.5	• real number
	'*'	• non-numeric input. You should not take shortcuts and choose one negative even integer or one negative real number and think you can test two things at the same time. You will not know whether the test fails for just one reason or both.
Boundary (extreme)	1	What is a boundary value? The smallest possible pyramid is a single symbol. So the value 1 is just within the boundary. Sometimes choosing test data throws up some interesting questions that need to be considered when designing the solution:
	0	• Should 0 be accepted? Is 0 an even number? Is it outside the boundary because a pyramid of 0 symbols is not really a pyramid? • Is there just one boundary? Should the program reject numbers that are too large?
	79 81	The output would not look like a pyramid if there were a wrap-around. So the program really should check how many symbols fit onto one line and not allow the user to input a number greater than this. If the number of characters across the screen is 80, then 79 would be just within the boundary but 81 would be outside the boundary, and should not be accepted. Note that by testing with values within the boundary you are also testing normal data, albeit at the extreme ends of the normal range.

Table 15.08 Test data for the pyramid problem

TASK 15.07

Look at the programs you wrote in Chapter 14.

1. Design test data for the number-guessing game (Task 14.09.2).
2. Design test data for the Connect 4 game (Task 14.11).

How to prevent errors

The best way to write a program that works correctly is to prevent errors in the first place. How can we minimise the errors in a program? A major cause of errors is poor requirements analysis. When designing a solution it is very important that we understand the problem and what the user of the system wants or needs. We should use:

- tried and tested design techniques such as structured programming or object-oriented design
- conventions such as identifier tables, data structures and standard algorithms
- tried and tested modules or objects from program libraries.

15.08 Corrective maintenance

Maintaining programs is not like maintaining a mechanical device. It doesn't need lubricating and parts don't wear out. **Corrective maintenance** of a program refers to the work required when a program is not working correctly due to a logic error or because of a run-time error. Sometimes program errors don't become apparent for a long time because it is only under very rare circumstances that there is an unexpected result or the program crashes. These circumstances might arise because part of the program is not used often or because the data on an occasion includes extreme values. Earlier corrective maintenance may also introduce other errors.

When a problem is reported, the programmer needs to find out what is causing the bug. To find a bug, a programmer either uses program debugging software or a trace table (see Section 15.06).

TASK 15.08

1. Design a trace table for the following algorithm:

```
INPUT BinaryString
StringLength ← Length(BinaryString)
FOR i ← 1 TO StringLength
    Bit ← BinaryString[i]
    BitValue ← IntegerValue(Bit) // convert string to integer
    DenaryValue ← DenaryValue + 2 + BitValue
NEXT i
```

2. Dry-run the algorithm using '101' as the input. Complete the trace table.
3. The result should be 5. Can you find the error in the code and correct it?

15.09 Adaptive maintenance

Programs often get changed to make them perform functions they were not originally designed to do.

For example, the Connect 4 game introduced in Chapter 13 (Worked Example 13.03) allows two players, O and X, to play against each other. An amended version would be for one player to be the computer. This would mean a single player could try and win against the computer.

Adaptive maintenance is the action of making amendments to a program to enhance functionality or in response to specification changes.

TASK 15.09

Design the algorithm to simulate the computer playing the part of Player X in Connect 4.

15.10 Perfective maintenance

The program runs satisfactorily. However, there is still room for improvement. For example, the program may run faster if the file handling is changed from sequential access to direct access.

TASK 15.10

Analyse the pseudocode below and make amendments to enhance maintainability.

```
FUNCTION GetPositiveNumber
    DECLARE n : INTEGER
    OUTPUT "Enter a positive number: "
    INPUT n
    RETURN n
ENDFUNCTION
// main program
REPEAT
    Number1 ← GetPositiveNumber
    IF Number1 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number1 > 0
REPEAT
    Number2 ← GetPositiveNumber
    IF Number2 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number2 > 0
```

Reflection Point:

Have you used dry-running for programs you have written? You can check your trace table if you add output statements at key points in your program. You can then compare the program output with the contents of your trace table.

Summary

- The stages of the program development cycle consist of analysis, design, coding, testing and maintenance.
- Structure charts are graphical representations of the modular structure of solutions.
- A structure chart shows the interface between modules: parameters passed between calling module and the module being called.
- Structure charts show selection, where a module is called only under certain conditions.
- Structure charts show repetition, where modules are called repeatedly.
- A state transition diagram is another way of documenting an algorithm.
- Testing strategies include stub testing, black-box testing, white-box testing, integration testing, alpha and beta testing, and acceptance testing.
- Locating and correcting logic errors and run-time errors can be done by dry-running an algorithm or using a trace table.
- Corrective maintenance means fixing bugs that have come to light during use of the program.
- Adaptive maintenance involves altering an algorithm and data structure in response to required

changes.

- **Perfective maintenance** means enhancing performance or maintainability.

Exam-style Questions

1 Consider this code for a function:

```
FUNCTION Binary(Number : INTEGER) RETURNS STRING
    DECLARE BinaryString : STRING
    DECLARE PlaceValue : INTEGER
    BinaryString ← ''  // empty string
    PlaceValue ← 8
    REPEAT
        IF Number >= PlaceValue
          THEN
            BinaryString ← BinaryString & '1' // concatenates two strings
            Number ← Number – PlaceValue
          ELSE
            BinaryString ← BinaryString & '0'
        ENDIF
        PlaceValue ← PlaceValue DIV 2
    UNTIL Number = 0
    RETURN BinaryString
ENDFUNCTION
```

a Dry-run the function call `Binary(11)` by completing the given trace table.

Number	BinaryString	PlaceValue	Number >= PlaceValue
11	''	8	

What is the return value? [5]

b **i** Now dry-run the function call `Binary(10)` by completing the given trace table.

Number	BinaryString	PlaceValue	Number >= PlaceValue
10	''	8	

What is the return value? [3]

ii The algorithm is supposed to convert a denary integer into the equivalent binary number, stored as a string of 0s and 1s. Explain the result of each dry-run and what needs changing in the given algorithm. [3]

2 A procedure to output a row in a tally chart has been written using pseudocode:

```
PROCEDURE OutputTallyRow(NumberToDraw : INTEGER)
    IF Count > 0
      THEN
        FOR Count ← 1 TO NumberToDraw
            IF (Count MOD 5) = 0
              THEN
                OUTPUT('\') // every 5th bar slants the other way
              ELSE
```

```
                OUTPUT('/')
            ENDIF
        NEXT Count
    ENDIF
    OUTPUT NewLine // move to next row
ENDPROCEDURE
```

Suggest suitable test data that will test the procedure adequately. Justify your choices in each case.

3 A random number generator is to be tested to see whether all numbers within the range 1 to 20 are generated equally frequently. The structured English version of the algorithm is [9]

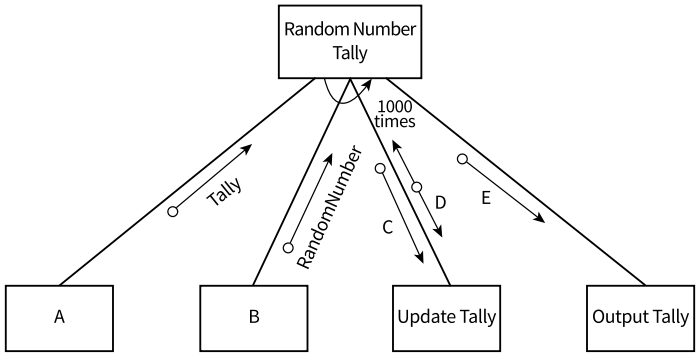
```
Initialise a tally for the numbers 1 to 20
Repeatedly generate numbers in range 1 to 20
For each number generated, increment the relevant count
Calculate how often each number should be generated (expected frequency)
Output expected frequency
Output the list of numbers as a table with actual frequency
```

The identifiers required are:

Identifier	Data Type	Explanation
`Tally`	`Array[1 : 20] OF INTEGER`	1D array to store the count of how many times each number has been generated
`RandomNumber`	`INTEGER`	The random number generated
`NumberOfTests`	`INTEGER`	The number of times a random number is to be generated (1000 in this example)
`ExpectedFrequency`	`INTEGER`	The number of times any one number would be generated if all numbers are generated equally frequently (1000/20 in this example)

a Complete the structure chart below by naming the labels A to E. [5]

b Develop pseudocode from the structure chart. [12]

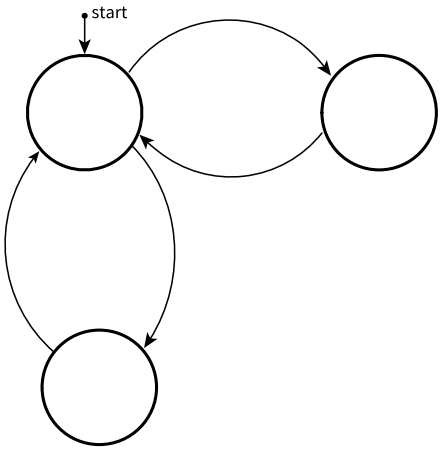


4 A car park has a barrier at the exit. The starting position of the barrier is lowered. When a car wants to exit the car park, the driver has to insert a coin into a coin slot at the barrier. The barrier raises and allows the car to drive out of the car park. After the car has passed through the barrier, the barrier lowers. In case of emergency, a member of staff can open the barrier using a remote control. The barrier will remain open until the remote control is used again to lower the barrier.

The barrier has three states: lowered, raised and open. The transition from one state to another is as shown in the state-transition table:

Current state	Event	Next state
Barrier lowered	Coin inserted	Barrier raised
Barrier lowered	Open remotely	Barrier open
Barrier open	Close remotely	Barrier lowered
Barrier raised	Car has exited	Barrier lowered

Complete the state-transition diagram for the barrier:



[7]