

**FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION
TECHNOLOGY: SYSTEMS DEVELOPMENT**

ID 78965 LEVEL 4 – CREDITS 165

LEARNER GUIDE

SAQA: 14918

DESCRIBE THE PRINCIPLES OF COMPUTER PROGRAMMING

Learner Information:

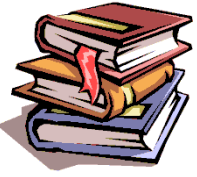


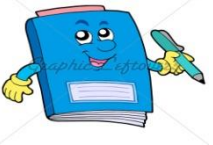

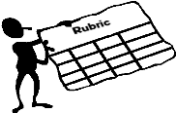
Details	Please Complete this Section
Name & Surname:	
Organisation:	
Unit/Dept:	
Facilitator Name:	
Date Started:	
Date of Completion:	

Copyright

All rights reserved. The copyright of this document, its previous editions and any annexures thereto, is protected and expressly reserved. No part of this document may be reproduced, stored in a retrievable system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission.

Key to Icons

The following icons may be used in this Learner Guide to indicate specific functions:

	This icon means that other books are available for further information on a particular topic/subject.
Books	
	This icon refers to any examples, handouts, checklists, etc...
References	
	This icon represents important information related to a specific topic or section of the guide.
Important	
	This icon helps you to be prepared for the learning to follow or assist you to demonstrate understanding of module content. Shows transference of knowledge and skill.
Activities	
	This icon represents any exercise to be completed on a specific topic at home by you or in a group.
Exercises	
	An important aspect of the assessment process is proof of competence. This can be achieved by observation or a portfolio of evidence should be submitted in this regard.
Tasks/Projects	



Workplace Activities

An important aspect of learning is through workplace experience. Activities with this icon can only be completed once a learner is in the workplace



Tips

This icon indicates practical tips you can adopt in the future.



Notes

This icon represents important notes you must remember as part of the learning process.

Learner Guide Introduction

About the Learner Guide...	<p>This Learner Guide provides a comprehensive overview of the Describe the principles of Computer Programming, and forms part of a series of Learner Guides that have been developed for FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT ID 78965 LEVEL 4 – CREDITS 165. The series of Learner Guides are conceptualized in modular's format and developed FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT ID 78965 LEVEL 4 – CREDITS 165. They are designed to improve the skills and knowledge of learners, and thus enabling them to effectively and efficiently complete specific tasks. Learners are required to attend training workshops as a group or as specified by their organization. These workshops are presented in modules, and conducted by a qualified facilitator.</p>
Purpose	The purpose of this Unit Standard is to Describe the principles of Computer Programming
Outcomes	Describe the principles of Computer Programming
Assessment Criteria	The only way to establish whether a learner is competent and has accomplished the specific outcomes is through an assessment process. Assessment involves collecting and interpreting evidence about the learner's ability to perform a task. This guide may include assessments in the form of activities, assignments, tasks or projects, as well as workplace practical tasks. Learners are required to perform tasks on the job to collect enough and appropriate evidence for their portfolio of evidence, proof signed by their supervisor that the tasks were performed successfully.

To qualify	To qualify and receive credits towards the learning programme, a registered assessor will conduct an evaluation and assessment of the learner's portfolio of evidence and competency
Range of Learning	This describes the situation and circumstance in which competence must be demonstrated and the parameters in which learners operate
Responsibility	<p>The responsibility of learning rest with the learner, so:</p> <ul style="list-style-type: none"> • Be proactive and ask questions, • Seek assistance and help from your facilitators, if required.

Describe the principles of Computer Programming

Learning Unit 1

UNIT STANDARD NUMBER	:	14918
LEVEL ON THE NQF	:	3
CREDITS	:	5
FIELD	:	Physical, Mathematical, Computer and Life Sciences
SUB FIELD	:	Construction Information Technology and Computer Sciences

PURPOSE:	<p>This unit standard is intended:</p> <ul style="list-style-type: none"> <input type="checkbox"/> to provide a conceptual knowledge of the areas covered <input type="checkbox"/> for those entering the workplace in the area of systems development <input type="checkbox"/> as additional knowledge for those wanting to understand the areas covered <p>People credited with this unit standard are able to:</p> <ul style="list-style-type: none"> <input type="checkbox"/> describe problem analysis and program design techniques <input type="checkbox"/> describe different data representations used in computer programs <input type="checkbox"/> describe basic programming principles <input type="checkbox"/> described the principles used in designing a computer program <p>The performance of all elements is to a standard that allows for further learning in this area</p>
LEARNING ASSUMED TO BE IN PLACE:	
<p>Open.</p> <p>The credit value of this unit is based on a person having the prior knowledge and skills to:</p> <ul style="list-style-type: none"> <input type="checkbox"/> Demonstrate an understanding of fundamental mathematics (at least NQF level 3) <input type="checkbox"/> Demonstrate PC competency skills (End User Computing unit standards up to Level 3). 	

SESSION 1.

Describe problem analysis and program design techniques.

Learning Outcomes

- The description provides an appreciation of the steps and techniques of program maintenance.
- The description identifies different problem analysis techniques (at least 2).
- The description identifies different programming design techniques.

The description provides an appreciation of the steps and techniques of program maintenance.

Program development can be described as a seven step process:

1. Understand the problem.
2. Plan the logic of the program.
3. Code the program using a structured high level computer language.
4. Using a compiler, translate the program into a machine language.
5. Test and debug the program.
6. Put the program into production.
7. Maintain and enhance the program.

Planning the logic of the program requires the development of algorithms. An algorithm is a finite, ordered set of unambiguous steps that terminates with a solution to the problem. Human readable representations such as flow charts and pseudo code are typically used to describe the steps of an algorithm and the relationships among the steps. A **flow chart** is a graphical representation of the steps and control structures used in an algorithm. A flow chart does not involve a particular programming language, but rather uses a set of geometric symbols and flow control lines to describe the algorithm. From a flowchart, a programmer can produce the high level code required to compile an executable program. Initially, the standard for describing flow charts only specified the types of shapes and lines used to produce a flow chart. The introduction of

structured programming in the 1960s and 70s brought with it the concept of Structured Flow Charts. In addition to a standard set of symbols, structured flow charts specify conventions for linking the symbols together into a complete flow chart. The structured programming paradigm evolved from the mathematically proven concept that all problems can be solved using only three types of control structures:

Sequence, Decision (or Selection), Iterative (or looping).

The definition of structured flow charts used in this document and software further defines:

3 types of sequential structures: Process, Input/Output, and Subroutine Call

3 types of decision structures: Single Branch, Double Branch, and Case.

4 types of iterative structures: Test at the Top, Test at the Bottom, Counting, and User Controlled Exit.

A comment structure.

The SFC program is designed to aid the programmer in designing and presenting structured flow charts.

The description identifies different problem analysis techniques (at least 2).

Top-down and **bottom-up** are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories and management and organization. In practice, they can be seen as a style of thinking and teaching.

A **top-down** approach (also known as stepwise design or deductive reasoning, and in many cases used as a synonym of *analysis* or *decomposition*) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top

down approach starts with the big picture. It breaks down from there into smaller segments.

A **bottom-up** approach (also known as inductive reasoning,^[1] and in many cases used as a synonym of *synthesis*) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed.

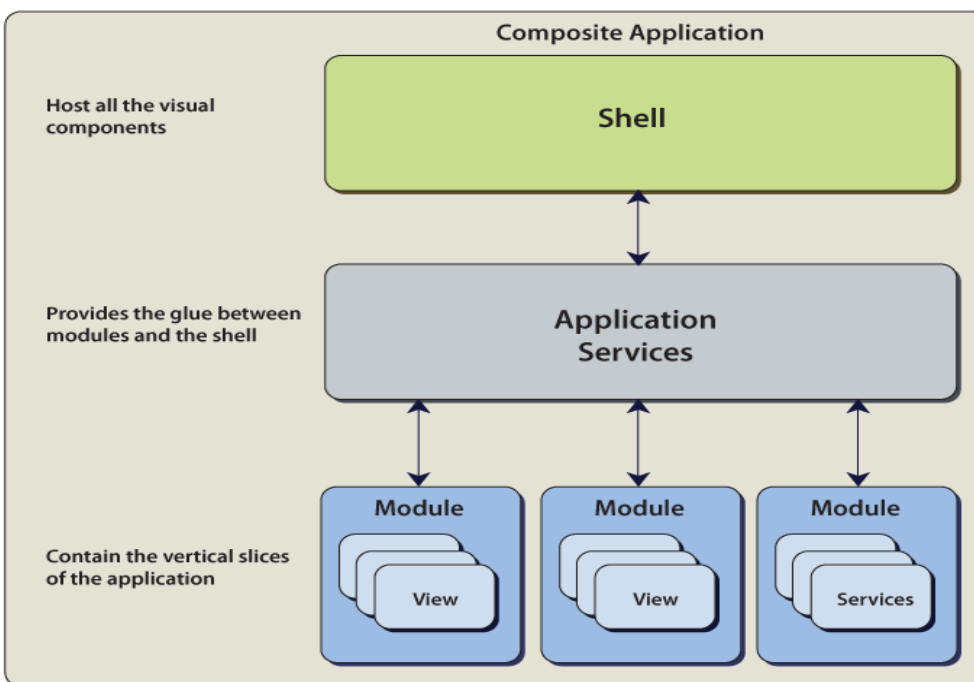
Product design and development

During the design and development of new products, designers and engineers rely on both a bottom-up and top-down approach. The bottom-up approach is being utilized when off-the-shelf or existing components are selected and integrated into the product. An example would include selecting a particular fastener, such as a bolt, and designing the receiving components such that the fastener will fit properly. In a top-down approach, a custom fastener would be designed such that it would fit properly in the receiving components.¹ For perspective, for a product with more restrictive requirements (such as weight, geometry, safety, environment, etc.), such as a space-suit, a more top-down approach is taken and almost everything is custom designed. However, when it's more important to minimize cost and increase component availability, such as with manufacturing equipment, a more bottom-up approach would be taken, and as many off-the-shelf components (bolts, gears, bearings, etc.) would be selected as possible. In the latter case, the receiving housings would be designed around the selected components.

Modularity

Modularity is designing a system that is divided into a set of functional units (named modules) that can be composed into a larger application. A module represents a set of related concerns. It can include a collection of related components, such as features, views, or business logic, and pieces of infrastructure, such as services for logging or authenticating users. Modules are independent of one another but can communicate with each other in a loosely coupled fashion. A composite application exhibits modularity. For example, consider an online banking program. The user can access a variety of functions, such as transferring money between accounts, paying bills, and updating personal information from a single user interface (UI). However, behind the scenes, each of these functions is a discrete module. These modules communicate with each other and with back-end systems such as database servers. Application services integrate components within the different modules and handle the communication with the user. The user sees an integrated view that looks like a single application.

Figure 1 illustrates a design of a composite application with multiple modules.



Module composition

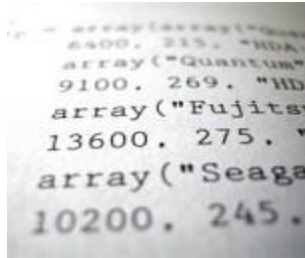
Why Choose a Modular Design?

The following scenarios describe why you might want to choose a modular design for your application:

- **Simplified modules.** Properly defined modules have a high internal cohesion and loose coupling between modules. The coupling between the modules should be through well-defined interfaces.
- **Developing and/or deploying modules independently.** Modules can be developed, tested, and/or deployed on independent schedules when modules are developed in a loosely coupled way. By doing this, you can do the following:
 - You can independently version modules.
 - You can develop and test modules in isolation.
 - You can have modules developed by different teams.
- **Loading modules from different locations.** A Windows Presentation Foundation (WPF) application might retrieve modules from the Web, from the file system and/or from a database. A Silverlight application might load modules from different XAP files. However, most of the time, the modules come from one location; for example, there is a specific folder that contains the modules or they are in the same XAP file.
- **Minimizing download time.** When the application is not on the user's local computer, you want to minimize the time required to download the modules. To minimize the download time, only download modules that are required to start-up the application. The rest are loaded and initialized in the background or when they are required.
- **Minimizing application start-up time.** To get part of the application running as fast as possible, only load and initialize the module(s) that are required to start the application.
- **Loading modules based on rules.** This allows you to only load modules that are applicable for a specific role. An application might retrieve from a service the list of modules to load.

The description identifies different programming design techniques.

What Is Pseudocode?



```
array("Quantum", 9100, 269, "HD")
array("Fujitsu", 13600, 275, "HD")
array("Seagate", 10200, 245, "HD")
```

Pseudocode and flowcharts are both popular ways of representing algorithms. Pseudocode has been chosen as the primary method of representing an algorithm because it is easy to read and write, and allows the programmer to concentrate on the logic of the problem. Pseudocode is really structured English. It is English that has been formalised and abbreviated to look like the high-level computer languages. Pseudocode provides another way to represent program logic. The approach is to write the program in a language similar to the one used for implementation but with an easily understood shorthand for complex expressions. The idea is to represent the logic but to ignore many of the syntactic requirements of the programming language. A pseudocode design description is thus a mixture of human language and programming constructs. Instead of the complete logic for well-known functions, simple statements are used. Similarly, the program flow can be represented with traditional conditional statements, using written expressions to describe the logic. While there are no generally accepted pseudocode notation standards, it is a good idea to use language constructs that are similar to those in the high-level language being used. When you do, the pseudocode will look familiar at implementation time and will provide a framework for implementation. Because pseudocode does not require the programming language details, it is generally easier and quicker to produce than a completed source program. While there is no standard pseudocode, once you understand one version, it is easy to understand other versions. Many programmers use pseudocode. Abstract versions of it are used to express abstract algorithms. More language specific versions of it are used to express design for programs in those languages. Pseudocodes supposed to be clear enough that a human being can execute it "by hand," without using a computer and with little or no knowledge of programming languages. Pseudocode is used to represent the algorithm. The common characteristics of pseudocode:

- Statements are written in simple English
- Each instructions is written on a separate line
- Keywords and indentation are used to signify particular control structures
- Each set of instructions is written from top to bottom, with only one entry and one exit
- Groups of statements may be formed into modules, and the module given a name

Pseudocode is an English-like nonstandard language that lets you state your solution with more precision than you can in plain English but with less precision than is required when using a formal programming language. Pseudocode permits you to focus on the program logic without having to be concerned just yet about the precise syntax of a particular programming language. However, pseudocode is not executable on the computer.

Documenting the Program

Documenting is an ongoing, necessary process, although, as many programmers are, you may be eager to pursue more exciting computer-centered activities. Documentation is a written detailed description of the programming cycle and specific facts about the program. Typical program documentation materials include the origin and nature of the problem, a brief narrative description of the program, logic tools such as flowcharts and pseudocode, data-record descriptions, program listings, and testing results. Comments in the program itself are also considered an essential part of documentation. Many programmers document as they code. In a broader sense, program documentation can be part of the documentation for an entire system. The wise programmer continues to document the program throughout its design, development, and testing. Documentation is needed to supplement human memory and to help organize program planning. Also, documentation is critical to communicate with others who have an interest in the program, especially other programmers who may be part of a programming team. And, since turnover is high in the computer industry, written documentation is needed so that those who come after you can make any necessary modifications in the program or track down any errors that you missed.

SESSION 2.

Describe different data representations used in computer programs.

Learning Outcomes

- The description distinguishes between different numeric data types (at least 3).
- The description identifies different logical data types.
- The description distinguishes between different internal representations of data types.
- The description identifies different logical operators.

The description distinguishes between different numeric data types (at least 3).

Binary, Decimal, Hexadecimal, Octal.

Numerical Notations: Hexadecimal, Decimal, Octal, and Binary

Since most programming languages accept literal constants in the same decimal notation that we all learned as children, it is not uncommon for a programmer to master many of the basic programming techniques and concepts before they ever feel the need to use an alternate notation. Even assembly languages can be used without leaving the comfort of decimal notation. While decimal notation is appropriate for mathematical calculations, it conceals much of the underlying structure implicit in the way data is stored and handled internally by the computer. For certain tasks, representing data in hexadecimal or binary notation can provide deeper insights into exactly what a program is doing. They can also help illuminate some of the limitations of numerical data types.

Getting out of our own way

Binary, hexadecimal, and octal notations are not difficult to learn. In fact, you already know the most fundamental concepts required to use the notations. They are all inspired by the same underlying principle that defines the decimal system, which you mastered long before you learned the mathematics required to even casually describe

it. The most difficult obstacle that many face is the unquestioned assumption that a number is identical with its representation. To handle data flexibly, we have to realize that when we write a combination of symbols, such as 32, we are using merely a representation of a number. There is nothing intrinsic about the number thirty-two that demands that it be represented by those specific symbols. None of its arithmetic or algebraic properties depend upon the visual symbols 3 and 2. The decimal notation developed because it was found to be convenient for manipulating the symbols used to represent numbers and helped provide insight into their mathematical relationships. It is, however, just one member of an infinite family of similar notation systems. To help us move past this obstacle, we'll begin by taking a closer look at how decimal notation is used. This discussion is not intended to be a rigorous mathematical inquiry, and you may find other descriptions that use an entirely different approach. The goal here is merely to highlight the salient features of the decimal system that will help us understand hexadecimal, octal, and binary notations. For simplicity, we are going to focus only on unsigned whole integer values, and disregard negative values and decimal points.

The Decimal System

Modern decimal notation is a positional system. To understand what this means, let's review the basic task of counting. To count aloud, we merely recite a list of words in a specific sequence.

zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven, twelve, thirteen, fourteen.....

When we want to write this sequence, rather than spell out the spoken representation, we resort to a compact notation using only ten unique symbols.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

At this point, we have used all ten of our symbols, so to provide a unique representation of the next highest integer, we need more than one digit. We can rewrite our sequence as two digit values like this:

00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23....

It may help to envision a mechanical counter like an old style odometer display. Each position consists of a wheel with the numerals 0 through 9 printed in order on the outer

circumference. As each wheel rolls over from 9 to 0, the wheel to the left is advanced one digit. This step is often called the carry, as in the expression "carry the one".

With two digits, we can continue with this pattern until we reach 99. After that we will need a third digit.

000, 001, 002, 003, 004, 005, 006, 007, 008, 009, 010 ... 098, 099, 100, 101, 102 ... 998, 999

Each digit occupies a position. The position at the furthest right, is position 0. The next position to the left is position 1, and so on through each position.

The following table shows the position of the digits in the decimal number 2993.

Position	4	3	2	1	0
Digit	0	2	9	9	3

The number, or value, represented by the sequence of digits 2993, is the sum of the value of each digit, and the value of each digit is determined by the position in which it appears.

So, with the sequence 2993 we have:

Position	Digit	Value
0	3	$3 * 1 = 3$
1	9	$9 * 10 = 90$
2	9	$9 * 100 = 900$
3	2	$2 * 1000 = 2000$

$$3 + 90 + 900 + 2000 = 2993$$

You should notice that each position indicates a power of ten. In position 2, the digit was multiplied by 10^2 . In position 3, the digit was multiplied by 10^3 .

Also, notice that the maximum value that can be represented by a given number of digits is one less than a power of ten.

Number of digits	Maximum Value
1	$10^1 - 1 = 9$
2	$10^2 - 1 = 99$
3	$10^3 - 1 = 999$
...	
10	$10^{10} - 1 = 9999999999$

In general, the maximum value that can be represented in the decimal notation when n digits are used is $10^n - 1$. Since the lowest value is 0, this means that n digits can represent 10^n unique values. The key to making this system work is the use of the symbol 0 as a place holder in any position that does not contribute to the value of the number represented. Without this place holder, expressing a number such as 202 would be very difficult. Normally, in the decimal notation, we don't indicate the digits on the left of a number when they are zero, so instead of 0021, we usually simply write 21. Conceptually, though, you can imagine an infinite list of zeros preceding any number written in decimal notation.

Binary

The binary notation works exactly the same way as the decimal notation, but instead of ten symbols, only two symbols are used: 0 and 1. Whereas decimal is a base 10 positional notation, binary is a base 2 positional notation.

A digit in binary, is usually referred to as a bit.

With one bit, we have two possible values:

0, 1

So, in binary, with one bit, we can count from zero to one. To count higher requires more bits. With two bits, we can count as high as three.

Decimal:	0, 1, 2, 3
Binary:	00, 01, 10, 11

The same pattern applies here as it did in the decimal notation. When a digit in one position rolls over from the highest numeral (1) to the lowest (0), the digit to the left increases. To avoid any confusion it should be explicitly stated here that in binary notation, 10 represents the number two. It does not represent the number ten. For years, our minds have associated the combination of symbols 10 with the concept "ten". This association is so strong that it can be difficult to temporarily put aside. It may help to realize that the symbols 0 and 1 are merely a convention. If we like, we could adopt any other two symbols, such as T and H. Let's say T represents zero, and H represents one. We can rewrite the above sequence this way:

Decimal:	0, 1, 2, 3
----------	------------

Binary:	TT, TH, HT, HH
---------	----------------

Unfortunately, when we want to express data in a binary notation that our compiler or interpreter will understand, we must return to the conventional symbols 0 and 1. In general, with n bits, we can represent the values 0 to $2^n - 1$. The total number of unique values we can represent when we have n bits, including 0, is 2^n . To convert from the binary notation to the decimal notation, we can use the same arithmetic that we used to determine the value of a number in decimal notation.

In binary, the value of a bit is multiplied by 2^{position} . Again, the right most position is position 0.

So the decimal notation equivalent of the binary 1101 is:

Position	Bit	Value
0	1	$1 * 2^0 = 1$
1	0	$0 * 2^1 = 0$
2	1	$1 * 2^2 = 4$
3	1	$1 * 2^3 = 8$

$$1 + 4 + 8 = 13$$

You should notice that value of each bit is either zero or an integral power of two. This is true because each bit may hold only one of two states: 0 or 1. Given an eight bit binary number, the resulting value will be the sum of one or more of the decimal values: 1, 2, 4, 8, 16, 32, 64, and 128. Given a four bit value, that list is reduced to: 1, 2, 4, and 8.

Here is an example using the binary value 1001.

$$(8 * 1) + (4 * 0) + (2 * 0) + (1 * 1) = 8 + 1 = 9$$

So the decimal equivalent of the binary number 1001 is 9.

The key concept is that these are equivalent representations. They both signify the same number.

Why use the binary notation?

The binary notation is used to closely represent the way information is physically implemented by a computer. Since an individual bit can only be in one of two states, designing machinery or circuitry to manipulate bits is well understood and these designs can be aggregated to perform operations on collections of bits. In fact, while a

computer's memory can be visualized as a very long sequence of individual bits, most microprocessors are not designed to perform operations on one bit at a time. Instead, the processor's architecture defines the least number of contiguous bits which it may be addressed and manipulated during one machine instruction. This least number of bits that a processor may manipulate at one time is called a BYTE. In the past, the size of a BYTE varied from one design to another, but over time, most designs have settled on a BYTE size of 8 bits. This is the BYTE used by processors that implement the Intel x86 architecture, which is the architecture on which most modern PC processors are based. The original version of the x86 architecture provided means to work with a single BYTE and with a pair of bytes called a WORD. A WORD, being composed of two BYTES, is a 16 bit wide field. With extensions to the architecture, the ability to work directly upon 32 bit double words (DWORD) was added, and on some machines 64 bit quadruple words (QWORD) may be handled by a single instruction.

This gives us the following common primitive data types:

BYTE	8 bits
WORD	16 bits
DWORD	32 bits
QWORD	64 bits

At the source code level, binary notation is rarely used to represent numerical data. Decimal notation is much more intuitive, and almost all programming languages support decimal notation directly in source code. Binary notation is usually only used to explicitly indicate the specific bits in a data field. The interpretation of such data is, of course, dependant on the program, but common usages include option flags, data masks, bit patterns, packed data, etc. When binary notation is used, the value of all bits in the data field are usually explicitly indicated, including leading zeros. So, to specify the contents of a single BYTE in binary, all eight bits would be included in the notation. For example, to indicate that only bit 0 and bit 4 should be "on", it might be written like this: 00010001. The exact syntax required to specify binary notation depends on the programming language used. Ansi C, in fact, does not specify any method for giving literal constants in binary notation.

Hexadecimal

Using binary notation can be cumbersome and prone to typing mistakes. If we wanted to specify a 16 bit value with all bits off except bit 9 in binary notation, it would look something like this (in FreeBASIC syntax):

```
&b0000000100000000
```

If you can verify that the 1 is in the 9th position in less than ten seconds, you deserve a pat on the back.

The Hexadecimal notation provides a more compact form, but it retains a visual similarity to the underlying machine representation of a value.

Hexadecimal is a base 16 positional notation. It uses sixteen symbols.

Hexadecimal digit	Binary Equivalent	Decimal Equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Hexadecimal notation is often abbreviated to Hex. Strictly speaking, Hex notation would refer to a radix 6 system, but the use of such a system is so rare that you can

safely assume that Hex stands for Hexadecimal. With one hexadecimal digit, we can represent the values 0 to F (In decimal: 0 to 15. In binary: 0000 to 1111.)

Again, this is a positional notation, so the value of a hexadecimal digit is 16^{position} .

So 0B12, in decimal notation would be:

Position	Digit	Value
0	2	$2 * 16^0 = 2$
1	1	$1 * 16^1 = 16$
2	B	$11 * 16^2 = 2816$
3	0	$0 * 16^3 = 0$

$$2 + 16 + 2816 + 0 = 2833$$

In practice, converting hexadecimal notation to decimal notation by hand is rare. The utility of hexadecimal notation consists in the compact manner that it can express an equivalent value in binary notation. Since sixteen is the fourth power of two, every hexadecimal digit corresponds to at most four bits. In other words, since four bits can only represent the values 0 to 15, exactly one hexadecimal digit is sufficient to represent a four bit binary value. An eight bit BYTE requires no more than two hexadecimal digits. A sixteen bit WORD requires no more than four hexadecimal digits. This pattern holds for any bit length that is a multiple of four. Since four bits is half the bit length of a BYTE, four bits is often called a NIBBLE. So we can say that one hexadecimal digit represents one NIBBLE.

Converting a four bit binary value to decimal is a relatively simple activity. With a little practice it can often be done mentally. This makes converting binary to hexadecimal trivial. Here is a thirty-two bit value generated by flipping a coin with consecutive bits shown in groups of four with a illustration of a conversion to hexadecimal. When performed often, this becomes second nature, and the intermediate decimal notation can usually be skipped.

Coin Tosses:	HHTT	THTH	HHTH	THHT	HHHT	THHT	HTHH	HHTT
Binary:	1100	0101	1101	0110	1110	0110	1011	1100
Decimal equivalent of each nibble:	12	5	13	6	14	6	13	12

Hexadecimal:	C5D6E6DC
--------------	----------

To convert this value to decimal by hand would be cumbersome, and the result depends on whether or not we are using signed or unsigned integer values. In any event, the decimal equivalent is not simply the decimal equivalent of each NIBBLE concatenated together. C5D6E6DC is **not** equivalent to 1251361461312. Using a calculator to perform the conversion to an unsigned integer, we find that C5D6E6DC is equivalent to decimal 3319195356. Every two hexadecimal digits align neatly on byte boundaries. This makes the memory layout of hexadecimal data visually obvious. However, when viewed this way, the "endian-ness" of the system must be considered. On the x86 architecture, multi-byte values are stored in memory in the little endian format, which means that less significant bytes are stored at lower addresses, so the value C5D6E6DC would appear in memory when listed by byte address like this: DC, E6, D6, C5.

Hexadecimal notation also helps highlight certain prominent bit patterns.

0	all bits clear	0000
F	all bits set	1111
even	low bit clear	xxx0
odd	low bit set	xxx1
<=7	high bit clear	0xxx
>=8	high bit set	1xxx

The last two patterns, 0xxx and 1xxx, are useful when examining signed integer data because the highest bit in the most significant BYTE is used as the "sign bit" and determines whether or not the value will be regarded as a negative number.

Octal

The octal notation is a base 8 positional notation. Valid digits are 0,1,2,3,4,5,6,7.

Note particularly, that the symbol "8" is not an octal digit.

This notation was much more common before 8 bits became the de facto standard size of a BYTE. The interesting characteristic of the octal notation is that any combination of three bits requires at most one octal digit. Following the same method

used above, we can determine that the highest unsigned value that n octal digits can represent is $8^n - 1$.

Numerical Notation Syntax

The lexical conventions for specifying different numerical notations differs from one programming language to another. Some languages support more than one convention. Some languages do not support certain notations at all. This table demonstrates the representation of the ASCII character code for a blank space in a number of different languages.

Language	Hexadecimal	Decimal	Octal	Binary
Ansi C	0x20	32	040	N/A
FreeBASIC	&h20	32	&o40	&b00100000
Gas (GNU assembler)	0x20	32	040	0b00100000
Nasm (Netwide Assembler)	20h	32d	40o	00100000b
	20x	32t	40q	00100000y
	0x20	0t32	0q40	0y00100000
	0h20	0d32	0o40	0b00100000
	\$20	32		
ECMAScript (Javascript)	0x20	32	n/a	n/a
Python	0x20	32	0o40	0b00100000
			040	

Displaying values in different notations:

The description identifies different logical data types.

Numeric Data

Numeric data simply means **numbers**. But, just to complicate things for you, numbers come in a variety of different **types**...

Integers

An integer is a **whole number** - it has **no decimal or fractional parts**. Integers can be either **positive** or **negative**.

Examples

- 12
- 45
- 1274
- 1000000
- -3
- -5735

123

Real Numbers

Any number that you could place on a number line is a real number. Real numbers include **whole numbers** (integers) and **numbers with decimal/fractional parts**. Real numbers can be **positive** or **negative**.

Examples

- 1
- 1.4534
- 946.5
- -0.0003
- 3.142

3.1

Some computer software used strange names for real data.

You might see this data type referred to as '**single**', '**double**' or '**float**'.

Currency

Currency refers to **real** numbers that are **formatted** in a specific way. Usually currency is shown with a **currency symbol** and (usually) **two decimal places**.

Examples

- £12.45
- -£0.01
- €999.00
- \$5500

A large, stylized, red 3D-effect font representing the value £99.

Percentage

Percentage refers to **fractional real** numbers that are formatted in a specific way - **out of 100**, with a **percent symbol**. So, the real value **0.5** would be shown as **50%**, the value **0.01** would be shown as **1%** and the number **1.25** would be shown as **125%**

Examples

- 100%
- 25%
- 1200%
- -5%

A large, stylized, red 3D-effect font representing the value 8%.

Inside the computer the 50% is stored as a **real** number: 0.5, But when it is displayed it is shown **formatted** as a percentage

Alphanumeric (Text) Data

Alphanumeric (often simply called 'text') data refers to data made up of **letters** (alphabet) and **numbers** (numeric). Usually **symbols** (\$%^+@, etc.) and spaces are also allowed.

Examples

- DOG
- "A little mouse"
- ABC123



Text data is often input to a computer with **speech marks** (". . ") around it:

"MONKEY"

These tell the computer that this is text **data** and not some special command.

Date and Time Data

Date (and time) data is usually **formatted** in a specific way. The format depends upon the **setup** of the computer, the software in use and the user's **preferences**.

Date Examples

- 25/10/2007
- 12 Mar 2008
- 10-06-08

Time Examples

- 11am
- 15:00
- 3:00pm
- 17:05:45

23/09/78 8:49am

With inputting dates particular care has to be taken if the data contains **American** style dates and the computer is setup to expect **international** style dates (or vice-versa)...

The date 06/09/08 refers to 6th September 2008 in the international system, but would be 9th June 2008 in America! Check your computer's settings.

Boolean (Logical) Data

Boolean data is sometimes called 'logical' data (or in some software, 'yes/no' data).

Boolean data can only have two values: **TRUE** or **FALSE**

Examples

- TRUE
- FALSE
- ON
- OFF
- YES
- NO

TRUE FALSE

Note that TRUE and FALSE can also be shown as **YES / NO**, **ON / OFF**, or even graphically as **tick boxes**(ticked / unticked)

Selecting Data Types

When we are presented with data to be input into a computer system, we must analyse it and select **appropriate data types** for each value... e.g. For the following data, we might use the data types shown:

Data Name

- Name
- Height
- Date of Birth
- Phone No.
- Pay Rate
- Tax Rate

Data Type

- **Text**
- **Real**
- **Date**
- **Alphanumeric**
- **Currency**
- **Percentage**

The description distinguishes between different internal representations of data types.

IDL DATA TYPES

1. DIGITS, BITS, BYTES, AND WORDS

We have gotten to the place where you need to know a little about the internal workings of computers. Specifically, how a computer stores numbers and characters.

Humans think of numbers expressed in **powers-of-ten**, or decimal numbers. This means that there are 10 digits (0 → 9), and you begin counting with these digits. When you reach the highest number expressible by a single digit, you use two digits and generate the next series of numbers, 10 → 99 and so on with more digits. This can be expressed mathematically - note the following when we use 5198 as an example.

$$5198 = 10^3 * 5 + 10^2 * 1 + 10^1 * 9 + 10^0 * 8$$

Fundamentally, all computer information is stored in the form of *binary numbers*, meaning **powers-of-two**. How many digits? Two! They are 0 and 1. The highest

number expressible by a single digit is 1. The two-digit numbers range from 10 to 11 and so on with more digits, but what decimal numbers do these binary numbers represent? Let's look at an example using 1101. Notice that we use the base of 2 instead of 10.

$$1101 \rightarrow 2^3 * 1 + 2^2 * 1 + 2^1 * 0 + 2^0 * 1 = 8 + 4 + 0 + 1 = 13$$

But wait a minute! The word "digit" is a misnomer - it implies something about 10 fingers. Hence, it's the word **bit** that is appropriate. Each binary "digit" is really a **bit**. So the binary number 1101 is a 4-bit number. What decimal number does the binary number 1001 equal? For convenience, computers and their programmers group the bits into groups of eight. Each group of 8 bits is called a **byte**. Consider, then, the binary number 11111111; it's the maximum-sized number that can be stored in a byte. What is this number? Finally, computers group the bytes into **words**. The oldest PC's dealt with 8-bit words - one byte. The Pentiums and Sparcs deal with 32-bit words - four bytes. What's the largest number you can store in a 4-byte word? And how about negative numbers? We'll learn answers to these questions below.

Below we describe how IDL (and everybody else) gets around this apparent upper limit on numbers. They do this by defining different data types. We don't cover all data types below - specifically, we omit Complex (yes, complex numbers!), Hexadecimal, and Octal data types, which you can look up if you are interested. Please refer to §2.2 in your text book, *"Practical IDL Programming"* for more information.

2. INTEGER DATA TYPES IN IDL

Integer data types store the numbers just like you'd expect. IDL supports integers of four different lengths: 1, 2, 4, and 8 bytes. The shorter the word, the less memory required; the longer the word, the larger the numbers can be. Different requirements require different compromises.

2.1. 1 byte: The Byte Data Type

The **byte** data type is a single byte long and always positive. Therefore, its values run 0 to 255. Images are always represented in bytes. The *data* might not be in bytes, but the numbers that the computer sends to the video processor card are always bytes. Video screens require lots of memory and really quick processing speed, so bytes are ideal. You generate an array using **bytarr()** for zeroed array or **bindgen()** for

index array; you can generate a single byte variable by saying **x=3b** for example. If a byte number exceeds 255 during a calculation, then it will "wrap around"; for example, 256 wraps to 0, 257 to 1, etc.

2.2. 2 byte: Integers and Unsigned Integers

With 2 bytes, numbers that are always positive are called **Unsigned Integers**. They can range from $0 \rightarrow 256^2-1$, or $0 \rightarrow 65535$. You generate an array using **uintarr()** for zeroed array or **uindgen()** for index array. How do you think unsigned integers wrap around? Normally you want the possibility of negative numbers and you use **Integers**. The total number of positive integer values is $256^2 / 2 = 32768$. One possible value is, of course, zero. So the number of negative and positive values differ by one. The choice is to favor negative numbers, so **integers** cover the range $-32768 \rightarrow 32767$. You generate an array using **intarr()** or **indgen()**. What happens with wrap around? What if **x = 5**, **y = 30000** and **z = x * y**? Check it out!

2.3. 4 bytes: Long Integers and Unsigned Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 becomes 256^4 . What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ulonarr()** or **ulindgen()** and **lonarr()** or **lindgen()**.

2.4. 8 bytes: 64-bit Long Integers and Unsigned 64-bit Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 become 256^8 . What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ulon64arr()** or **ul64indgen()** and **lon64arr()** or **l64indgen()**.

3. FLOATING DATA TYPES IN IDL

The problem with integer data types is that you can't represent anything other than integral numbers - no fractions! Moreover, if you divide two integer numbers and the result should be fractional, it won't be; instead, it will be rounded down (e.g., $5/3$ is calculated as 1). To get around this, the *floating* data type uses some of the bits to store an *exponents*, which may be positive or negative. You throw away some of the precision of the integer representation in favor of being able to represent a much wider range of numbers.

3.1. 4 bytes: Floats

"**Floating point**" means floating decimal point - it can wash all around. With Floats, the exponent can range from about -38 -> +38 and there is about 6 digits of precision. You generate an array using **fltarr()** or **findgen()** and a single variable by including a decimal point (e.g., **x = 3.**) or using exponential notation (e.g., **x = 3e5**).

3.2. 8 bytes: Double-Precision

Like Float, but the exponent can range from about -307 -> +307 and there is about 16 digits of precision. You generate an array using **dblarr()** or **dindgen()** and a single variable by writing something like **x = 3d** or **x = 3d5**.

4. STRINGS

Strings store characters - letters, symbols, and numbers (but numbers as *characters* - you can't calculate with strings!) A string constant such as **hello** consists of five letters. It takes 5 bytes to store this constant - one byte for each character. There are 256 possible characters for each of the bytes; with 2*26 letters (smalls and caps) and 10 digits, this leaves 104 other possibilities, which are used for things like semicolon, period and etc. You can generate an array of strings with **strarr()** or **sindgen()** and a single string using ' ' like this: **x = 'Hi there!!!'**.

5. STRUCTURES

Structures are a special data type that allows variables of different types and sizes to be packaged into one entity. This is different from an array, where every element must be the same data type. There are two kinds of structures in IDL: an *anonymous structure* (a package of arbitrary variables) and a *named structure* (a package of variables that conform to a template created by the user). Structures are used

The description identifies different logical operators.

And, Or, Not.

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the **&&** and **|** operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.

Logical Operators

Common Lisp provides three operators on Boolean values: `and`, `or`, and `not`. Of these, `and` and `or` are also control structures because their arguments are evaluated conditionally. The function `not` necessarily examines its single argument, and so is a simple function.

The logical operators are described in the following table:

Operator	Usage	Description
Logical AND (<code>&&</code>)	<code>expr1 && expr2</code>	Returns <code>expr1</code> if it can be converted to false; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code>&&</code> returns true if both operands are true; otherwise, returns false.
Logical OR (<code> </code>)	<code>expr1 expr2</code>	Returns <code>expr1</code> if it can be converted to true; otherwise, returns <code>expr2</code> . Thus, when used with Boolean values, <code> </code> returns true if either operand is true; if both are false, returns false.
Logical NOT (<code>!</code>)	<code>!expr</code>	Returns false if its single operand can be converted to true; otherwise, returns true.

Fortran has five **LOGICAL** operators that can only be used with expressions whose results are logical values (i.e., **.TRUE.** or **.FALSE.**). All **LOGICAL** operators have priorities lower than *arithmetic* and *relational* operators. Therefore, if an expression involving arithmetic, relational and logical operators, the arithmetic operators are evaluated first, followed by the relational operators, followed by the logical operators.

These five logical operators are

- **.NOT.** : logical **not**
- **.AND.** : logical **and**
- **.OR.** : logical **or**
- **.EQV.** : logical **equivalence**
- **.NEQV.** : logical **not equivalence**

SESSION 3.

Describe the basic principles of Computer Programming.

Learning Outcomes

- 1. The description identifies different algorithmic structures of programming languages.
- 2. The description identifies good program documentation principles (at least 3).
- 3. The description identifies programming quality assurance (QA) principles.
- 4. The description distinguishes between validation and verification.
- 5. The description explains the relationship between files, records and fields.

The description identifies different algorithmic structures of programming languages.

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s) for the target machines to produce *output* from given *input* (perhaps null).

The sequence structure

We have been using the *sequence* structure since early in the course. Basically we can describe the sequence structure using the pseudocode shown in Figure 1.

Pseudocode

Pseudocode is an outline of a program, written in a form that can easily be converted into real programming statements.

Figure 1. The sequence structure in pseudocode.

```
Enter
Perform one or more actions in sequence
Exit
```

Thus, the general requirement for the sequence structure is that one or more actions may be performed in sequence after entry and before exit. There may not be any branches or loops between the entry and the exit.

All actions must be taken in sequence.

The action elements themselves may be structures

However, it is important to note that one or more of the action elements may themselves be sequence, selection, or loop structures. If each of the structures that make up the sequence has only one entry point and one exit point, each such structure can be viewed as a single action element in a sequence of actions. Obviously, the sequence structure is the simplest of the three.

The selection structure

The *selection* or *decision* structure can be described as shown in the pseudocode **The selection structure in pseudocode.**

Enter

 Test a condition for true or false

 On true

 Take one or more actions in sequence

 On false

 Take none, one, or more actions in sequence

Exit

Test a condition for true or false

Once again, there is only one entry point and one exit point. The first thing that happens following entry is that some condition is tested for true or false. If the condition is true, one or more actions are taken in sequence and control exits the structure. If the condition is false, **none**, one or more different actions are taken in sequence and control exits the structure. (*Note the inclusion of the word none here.*)

The action elements may themselves be structures

Once again, each of the action elements in the sequence may be another sequence, selection, or loop structure. Eventually all of the actions for a chosen branch will be completed in sequence and control will exit the structure.

Sometimes no action is required on false

It is often the case that no action is required when the test returns false. In that case, control simply exits the structure without performing any actions.

The loop structure

The *loop* or *iteration* structure can be described as shown in the pseudocode

The loop structure in pseudocode.

Enter

Test a condition for true or false

Exit on false

On true

Perform one or more actions in sequence.

Go back and test the condition again

As before, there is only one entry point and one exit point.

Perform the test and exit on false

The first thing that happens following entry is that a condition is tested for true or false.

If the test returns false, control simply exits the structure without taking any action at all.

Perform some actions and repeat the test on true

If the test returns true:

- One or more actions are performed in sequence.
- The condition is tested again.

During each iteration, if the test returns false, control exits the structure. If the test returns true, the entire cycle is repeated.

Each action element may be another structure

Each of the action elements may be implemented by another sequence, selection, or loop structure. Eventually all of the actions will be completed and the condition will be tested again.

Need to avoid infinite loops

Generally speaking, unless something is done in one of the actions to cause the test to eventually return false, control will never exit the loop.

In this case, the program will be caught in what is commonly referred to as an *infinite loop*.

An introduction to algorithms and pseudocode

● What is an algorithm?

- Lists the steps involved in accomplishing a task (like a recipe)
- Defined in programming terms as 'a set of detailed and ordered instructions developed to describe the processes necessary to produce the desired output from a given input'

The description identifies good program documentation principles (at least 3).

Internal documentation

The variable declaration comments are one part of good internal documentation. Internal documentation is the set of comments that are included within the code to help clarify algorithms. Some students take internal documentation to mean that they should comment each line of code! This is obviously an example of overdoing a good idea. Any programmer knows how to increment a value in a variable; there's no reason to explain trivial operations such as that. The value of some good internal documentation should be clear by looking at the latest version of our sample program. Even with the good code organization and variable names, the function of this program is still not obvious.

Here's a list of items that should be included in your internal documentation:

1. "Block comments" (comments that are several lines long) should be placed at the head of every subprogram. These will include the subprogram name; the purpose of the subprogram; and a list of all parameters, including direction of information transfer (into this routine, out from the routine back to the calling routine, or both), and their purposes.
2. Meaningful variable names. In a nod to tradition, simple loop variables may have single letter variable names, but all others should be meaningful. Never use nonstandard abbreviations.
3. Each variable and constant must have a brief comment next to its declaration that explains its purpose. This applies to all variables, as well as to fields of struct declarations.
4. Complex sections of code and any other parts of the program that need some explanation should have comments just ahead of them or embedded in them.

A critical point: Documentation and internal documentation in particular, should be written and included in the program as the code is being written. Students tend to get in the habit of writing the code and then tossing in some documentation only if they have time before the due date. This makes documenting seem even more boring and tedious that it already is, and students who rush the documentation at the last minute usually do a very mediocre job. Documentation should be written when the code is being written, and should be typed in as the code is typed in. To demonstrate some of these points, here's yet another version of our program, this time containing some acceptable internal documentation:`#include <stdio.h>`

The trick with internal documentation is to make it easy to find while at the same time ensuring that it's not making the code hard to read. Block comments can be partially boxed (as shown) to separate them from the code. The use of the '*' at the start of each line of the shorter clarifying comments in the code serves a similar purpose. There's no one right way to do this, but it does need to be done. Experiment with some styles and pick one you like. One piece of advice: Don't fall in love with the "complete box" style. Lots of students like to completely enclose the block comments within a box. This

looks great, but the right-hand wall of the box is very hard to keep lined up as you make adjustments and additions to the comments. The "three wall" style shown above is much easier to deal with and looks almost as good.

External documentation

In a professional programmer's shop, large projects are documented in great detail, not only with comments in the code but with descriptions that are maintained separately from the code. In such an environment, programmers are often asked to fix problems in code that they didn't write. Many times, the author of the code isn't even with the company any longer. The documentation may be all the programmer has as reference material to help him or her make the necessary modifications. External documentation doesn't deal with details of the code. Instead, it serves as a general description of the project, including such information as what the code does, who wrote it and when, which common algorithms it uses, upon which other programs or libraries it is dependent, which systems it was designed to work with, what form and source of input it requires, the format of the output it produces, etc. Often the external documentation will include structure charts of the outline of the program that were produced when the program was being designed. All of this information is necessary to help other programmers understand the program. One seemingly innocent change in a program can have unpredictable consequences on other parts of the system. Good documentation can help prevent such problems.

In most programming classes, it is impractical for instructors to require large amounts of external documentation for programs that are only a few hundred lines long. Instead, it is common for instructors to require that a small amount of external documentation be included at the top of the program in the form of a large block comment. This condensed version should include at least the following pieces of information:

1. Your name, the course name, assignment name/number, instructor's name, and due date.
2. Description of the problem the program was written to solve.

3. Approach used to solve the problem. This should always include a brief description of the major algorithms used, or their names if they are common algorithms.
4. The program's operational requirements: Which language system you used, special compilation information, where the input can be located on disk, etc.
5. Required features of the assignment that you were not able to include.
6. Known bugs should be reported here as well. If a particular feature does not work correctly, it is in your best interest to be honest and complete about your program's shortcomings.

The final version of the program is given at the end of this document. Look it over carefully. Do you understand what the program does? More importantly for this discussion, do you understand how it does it? If the indentation, identifier names, and documentation helped, then they were well worth the time it took the programmer to put them in. Hopefully, you'll now see the value of putting such documentation in your programs as well. Take the time to ask yourself if you think the design of the comments is a good one; are the comments easy to find and to read? Do they distract from the code excessively? Are there too many of them to suit you, or too few? By asking and answering questions such as these, you will begin to develop a style of your own. When you see documentation styles that you like, consider adopting them into your own style. Soon you'll have one you like, and as a result you'll be more likely to use it.

There are plenty of decisions that were made in the design and documentation of this program that can be questioned and improved on. As you gain more experience in programming, consider revisiting this program and trying to rewrite it from scratch. Perhaps you can think of a better way to generate the times, for example. There isn't a program in existence that can't be improved, and this one is certainly no exception.

Core principles of good documentation

- *Less is more*: People simply do not like to read this kind of stuff and so leave out anything they don't need to know, such as theoretical discussions of why something is the way it is. Also, at least in the first release of docs, use the 80/20 rule and only

document the main use cases rather than the exceptions. Think of this as paying rent.

- *Pictures really are worth a 1000 words:* Screenshots and even better screencasts are much better ways of explaining action sequences than text. Any time you're putting a numbered list in the text, consider doing a short screencast instead or also.
- *Fix the software:* If you have a tough time explaining a feature or capability for the documentation then consider rewriting the software. The best documentation is software that needs none, the ultimate less being more.
- *Steal or copy:* Look at the documentation for software you use or is competitive or you respect and see what you can reuse from that, whether it's formatting, how they use graphics/screencasts or method of explaining difficult concepts.

The description identifies programming quality assurance (QA) principles.

Quality Information Systems: Vital to Total Quality Management

Heightened global competition has made it imperative for organizations to deliver products - goods and services - of consistently high quality. The principles of total quality management (TQM) recognize that consistent product quality results from designing and executing business processes so as to remove error and waste.

Product quality is the result of :

- a. Customer focus
- b. Introduction and continuous improvement of business processes and product-development processes that reduce variations
- c. Creation of a company-wide quality culture through motivation and training of all its members
- d. Continuous measurement and analysis of the accomplished results.

Quality information systems are vital to total quality management, because:

1. Business processes of the firm depend on information systems and, therefore, their quality depends to a large degree on IS quality

2. Information systems enable most projects of business process design. This IS enabled streamlining of processes gives fewer opportunities for error, thus leading to higher quality of the processes' outputs.

3. Information systems are a necessary component of the feedback loop in managing an enterprise. IS are necessary to continually gauge any deviations from the expected norms in the firm's performance and thus help reduce variance in performance.

International Standards Organization (ISO) 9000 - many companies, particularly in the manufacturing sectors, comply with the ISO 9000 group of quality standards. Such compliance is mandatory for those selling any of a broad range of products to the countries of the European Union. Some other countries also require this certification. The standards aim to ensure quality of products by certifying quality assurance during business processes, such as product design, manufacturing, delivery, and service support. Extensive quality-oriented information processing is a prerequisite for a certification of compliance.

Software Quality

There are many attributes of software quality. These include:

1. Effectiveness
- 2 Usability
3. Efficiency
4. Reliability
5. Maintainability
6. Understandability
7. Modifiability
8. Testability

Effectiveness Refers to the satisfaction of the user and organizational requirements as established during an analysis of these requirements, possibly using prototyping.

Usability The ease with which the intended users can use the system, depend on the proper user-system interface.

Efficient operation is reflected mainly in how economically hardware resources are used to satisfy the given effectiveness requirements

Reliability Refers to the probability that the information system will operate correctly; that is, according to specifications over a period of time. It may also be defined as the mean time between failures. Software reliability is rooted in its freedom from defects. If a system must run on different hardware or systems software platforms, portability should be included as a desired attribute.

Maintainability Refers to ease of understanding, modifying, and testing.

Understandability Is achieved by readable and well-commented system code and by documentation, which includes the requirements specifications, system documentation, user manuals, and, sometimes special maintenance documentation.

Modifiability Means that it is relatively easy to identify and change any part of the system that requires maintenance without affecting its other parts.

Testability Is the ease with which we can demonstrate that a modification resulted in a quality system.

Applying Total Quality Management to Information Systems

The following are the principal aspects of TQM - oriented quality assurance for information systems:

1. Customer focus is achieved by involving end users in the IS development process, particularly during its early stages when the requirements for the system are being defined. Systems prototyping and joint application development (JAD) are the principal techniques applied to this end. Joint Application Development (JAD) is an organizational technique for conducting meetings between the prospective users of an information system and its developers.
2. The life-cycle oriented systems development, with the inclusion of prototyping, is a process that lends itself to control, measurement, and continuous improvement. Support with CASE tools helps to ensure product quality.

3. Software development and maintenance teams are the primary human element in ensuring software quality.

4. The quality measurement program can assist in consistent striving for higher quality levels. Such a program rests on the foundation of software metrics. Software matrices include techniques for measuring the attributes of software and techniques for measuring the attributes of software development process.

The description distinguishes between validation and verification.

Verification and Validation

Before getting into the various forms and strategies of testing we must understand the process of **verifying** and **validating** the software code. Verification and validation is the generic name given to checking processes which ensure that the software **conforms to its specification** and meets **the needs of the customer**.

The system should be verified and validated at each stage of the software development process using documents produced in earlier stages. Verification and validation thus starts with **requirements reviews** and continues through **design and code reviews** to **product testing**.

Verification and validation are sometimes confused, but they are different activities. The difference between the two can be summarised as follows:

Validation: Are we building the right product?

Verification: Are we building the product right?

Verification involves checking that the program conforms to its specification.

Validation involves checking that the program as implemented meets the expectations of the customer. Requirements validation techniques, such as **prototyping**, help in this respect. However, flaws and deficiencies in the requirements can sometimes be discovered only when the system implementation is complete.

To satisfy the objectives of the verification and validation process, both **static and dynamic techniques** of system checking and analysis should be used. **Static techniques** are concerned with the analysis and checking of system representations such as the requirements document, design diagrams and the program source code. **Dynamic techniques** or tests involve exercising an implementation.

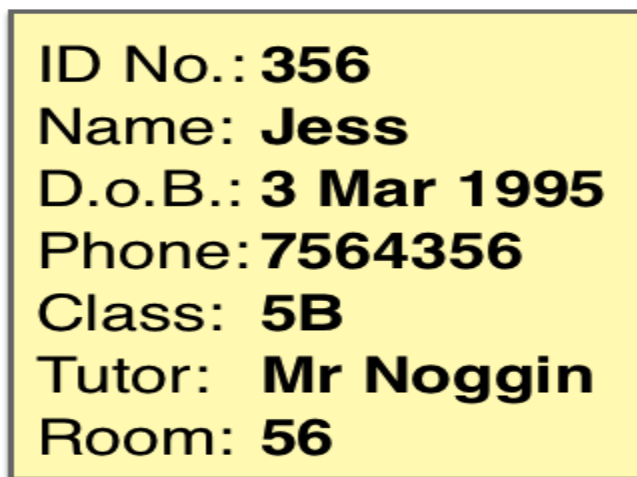
Static techniques include program inspections, analysis and formal verification. Some theorists have suggested these techniques should completely replace dynamic techniques in the verification and validation process and that testing is not necessary, this is not a useful point of view and could be 'considered harmful'. Static techniques can only check the correspondence between a program and its specification (verification). They cannot demonstrate that the software is operationally useful.

Although static verification techniques are becoming more widely used, **program testing** is still the predominant verification and validation technique. Testing involves exercising the program using data like the real data processed by the program. The existence of program defects or inadequacies is inferred from **unexpected system outputs**. Testing may be carried out during the implementation phase to verify that the software behaves as intended by its designer. This later testing phase checks conformance with the requirements and assesses the reliability of the system.

The description explains the relationship between files, records and fields.

What is a Record?

The **set of data** associated with a **single object or person** is known as a **record**. In the example of our students, the data associated with each student is a record. Here is Jess's record...

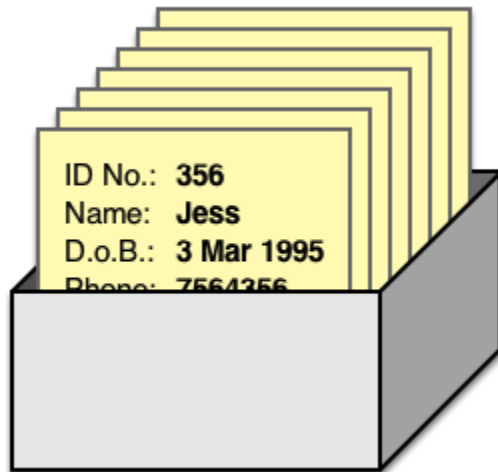


ID No.: 356
Name: Jess
D.o.B.: 3 Mar 1995
Phone: 7564356
Class: 5B
Tutor: Mr Noggin
Room: 56

Each student has their own record just like Jess's but with different data. The **data** in each record is **different**, but each **record** has the **same structure**. (each one has a

name, d.o.b., phone, etc.) We say that each record contains the same **fields**. A database is a collection of records. You can imagine a single record being a card with the details of one person/object written on it.

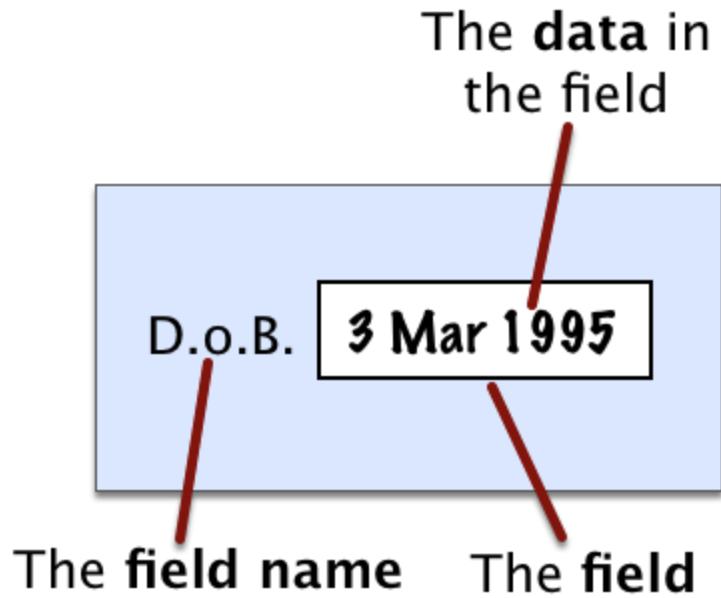
A database would be a boxful of these record cards...



This is exactly how a lot of old, manual databases used to look. If you went to a public library 30 years ago, and you wanted to find a specific book, you would have to look through boxes of index cards until you found the details of your book.

What is a Field, and What is a Field Name?

You'll see that each of our student's records contain the same items. These items are known as **fields**. Each field has a **field name** (e.g. 'Date of Birth') Each field will contain **different data** in each of the records (e.g. in Jess's record, the Phone field contains 7564356, but in Sita's record the Phone field contains 8565634 - same field, different data values) It can be a bit confusing - what's the difference between the field, the field name, and the data in the field?! Imagine that you were manually filling in a record card for Jess. The card would have various labels and boxes to write in...



- The **field** is the **box** that you would write in
- The **field name** is the **label** next to the box
- The **data** is what you would **write** in the box

SESSION 4.

Describe the principles used in designing a computer program.

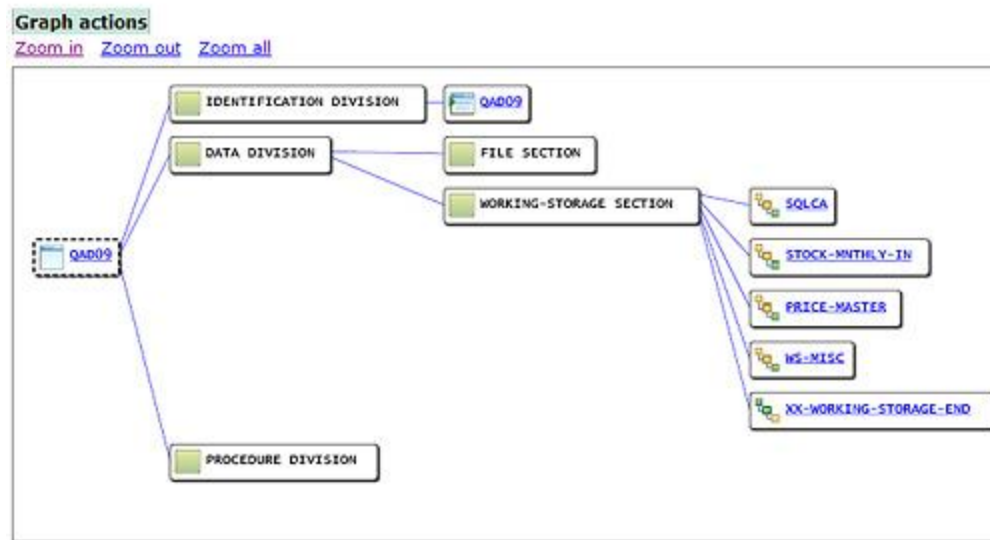
Learning Outcomes

- The description identifies methods of specifying problems.
- The description explains techniques used to research problems in terms of inputs and outputs.
- The description includes an evaluation of the viability of developing computer programs to solve problems and it identifies the issues in assessing the viability.
- The description explains the features of a computer program that could solve a given problem.

The description identifies methods of specifying problems.

A **Structure Chart** (SC) in software engineering and organizational theory is a chart which shows the breakdown of a system to its lowest manageable levels. They are used in structured programming to arrange program modules into a tree. Each module is represented by a box, which contains the module's name. The tree structure visualizes the relationships between modules. A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. The lines represent the connection and or ownership between activities and subactivities as they are used in organization

Program structure diagram



Top-down and **bottom-up** are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories (systemic), and management and organization. In practice, they can be seen as a style of thinking and teaching.

A **top-down** approach (also known as stepwise design or deductive reasoning,^[1] and in many cases used as a synonym of *analysis* or *decomposition*) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.[[]

A **bottom-up** approach (also known as inductive reasoning, and in many cases used as a synonym of *synthesis*) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-

up processing is a type of information processing based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

Decision trees, decision tables

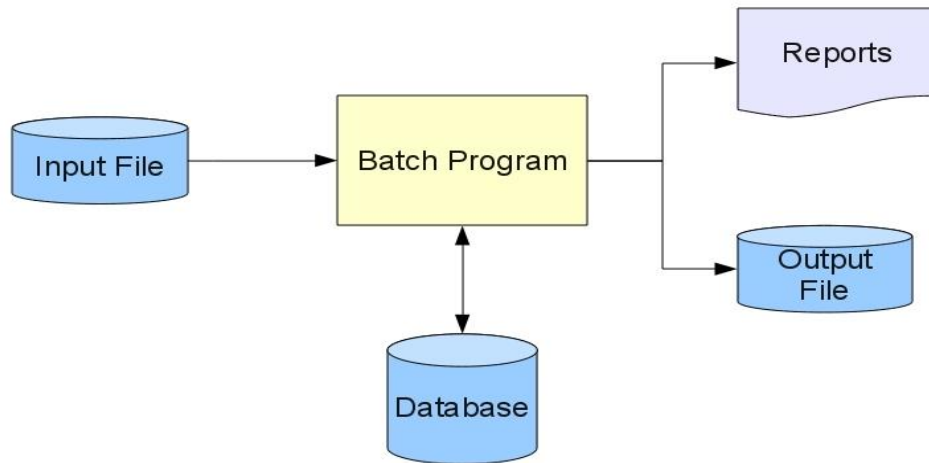
First it takes advantage of the sequential structure of decision tree branches so that the order of checking conditions and executing actions is immediately noticeable. Second, Conditions and actions of decision trees are found on some branches but not on others which contrasts with decision tables, in which they are all part of the same table. Those conditions and actions that are critical are connected directly to other conditions and actions, whereas those conditions that do not matter are absent. In other words it does not have to be symmetrical. Third, compared with decision tables, decision trees are more readily understood by others in the organization. Consequently, they are more appropriate as a communication tool. Unbalanced Decision Tables are a compromise between Decision Tables and Decision Trees. Decision Trees themselves can become quite complex with enough conditions and actions. Unbalanced Decision Tables provide either a prioritized list of conditions that lead to a set of actions, or a list of conditions that lead to a set of actions. The result is often more concise than either traditional Decision Tables or Decision Trees.

The description explains the features of a computer program that could solve a given problem.

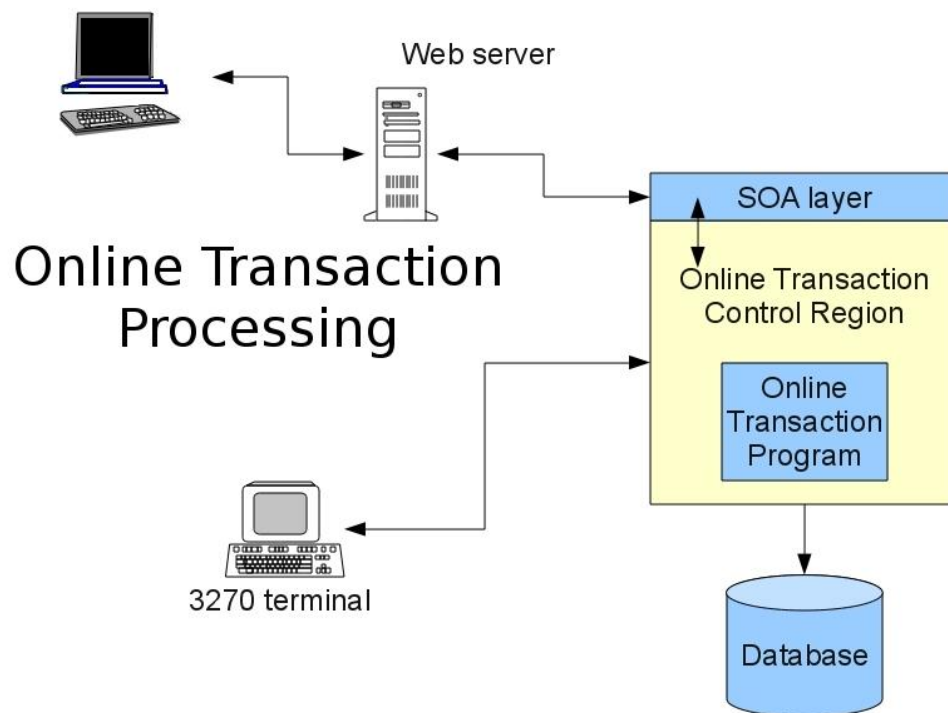
Batch processing is execution of a series of programs ("jobs") on a computer without manual intervention. Jobs are set up so they can be run to completion without manual intervention. So, all input data are preselected through scripts, command-line parameters, or job control language. This is in contrast to "online" or interactive programs which prompt the user for such input. A program takes a set of data files as input, processes the data, and produces a set of output data files. This operating environment is termed as "batch processing" because the input data are collected into batches of files and are processed in batches by the program.

Batch processing has these benefits:

- It can shift the time of job processing to when the computing resources are less busy.
- It avoids idling the computing resources with minute-by-minute manual intervention and supervision.
- By keeping high overall rate of utilization, it amortizes the computer, especially an expensive one.
- It allows the system to use different priorities for batch and interactive work.



Batch Processing



Online Transaction Processing

The **mouse** pointing device sits on your work surface and is moved with your hand. In older mice, a ball in the bottom of the mouse rolls on the surface as you move the mouse, and internal rollers sense the ball movement and transmit the information to the computer via the cord of the mouse.

The newer **optical mouse** does not use a rolling ball, but instead uses a light and a small optical sensor to detect the motion of the mouse by tracking a tiny image of the desk surface. Optical mice avoid the problem of a dirty mouse ball, which causes regular mice to roll unsmoothly if the mouse ball and internal rollers are not cleaned frequently.

A **cordless** or **wireless mouse** communicates with the computer via radio waves (often using **BlueTooth** hardware and protocol) so that a cord is not needed (but such mice need internal batteries).

A mouse also includes one or more buttons (and possibly a scroll wheel) to allow users to interact with the GUI. The traditional PC mouse has two buttons, while the traditional Macintosh mouse has one button. On either type of computer you can also use mice with three or more buttons and a small scroll wheel (which can also usually be clicked like a button).

Touch pad

Most laptop computers today have a **touch pad** pointing device. You move the on-screen cursor by sliding your finger along the surface of the touch pad. The buttons are located below the pad, but most touch pads allow you to perform "mouse clicks" by tapping on the pad



Two-button mouse with scroll wheel



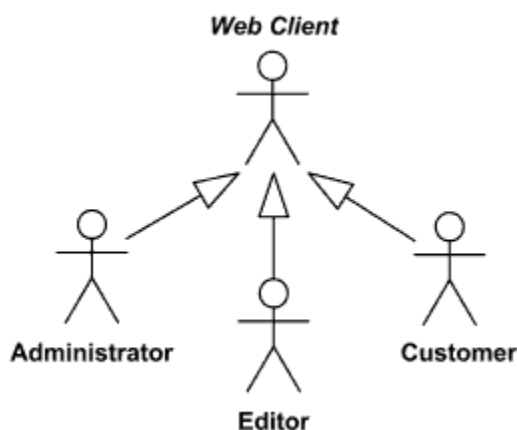
Wireless Macintosh mouse

itself.

Touch pads have the advantage over mice that they take up much less room to use. They have the advantage over trackballs (which were used on early laptops) that there are no moving parts to get dirty and result in jumpy cursor control.

The description explains techniques used to research problems in terms of inputs and outputs.

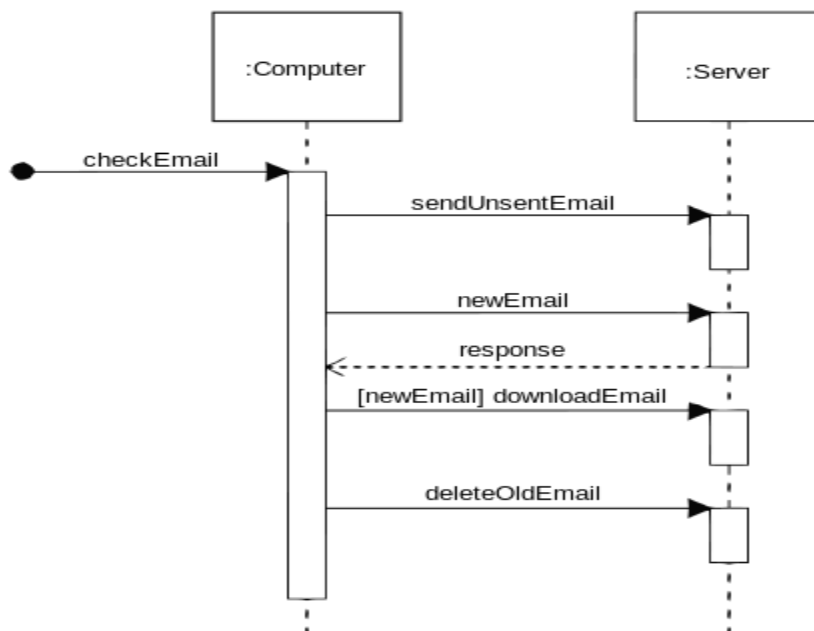
An **actor** in the Unified Modeling Language (UML) "specifies a role played by a user or any other system that interacts with the subject. "An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is external to the subject. "Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., "role") of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances. UML 2 does not permit associations between Actors. The use of generalization/specialization relationship between actors is useful in modeling overlapping behaviours between actors and does not violate this constraint since a generalization relation is not a type of association. Actors interact with use cases.



So the following are the places where use case diagrams are used:

- Requirement analysis and high level design.
- Model the context of a system.
- Reverse engineering.
- Forward engineering.

A **sequence diagram** is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called **event diagrams**, **event scenarios**, and **timing diagrams**. A sequence diagram shows, as parallel vertical lines (*lifelines*), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner.



The description includes an evaluation of the viability of developing computer programs to solve problems and it identifies the issues in assessing the viability.

Testing the Program

Some experts insist that a well-designed program can be written correctly the first time. In fact, they assert that there are mathematical ways to prove that a program is correct. However, the imperfections of the world are still with us, so most programmers get used to the idea that their newly written programs probably have a few errors. This is a bit discouraging at first, since programmers tend to be precise, careful, detail-oriented people who take pride in their work. Still, there are many opportunities to introduce mistakes into programs, and you, just as those who have gone before you, will probably find several of them. Eventually, after coding the program, you must prepare to test it on the computer. This step involves these phases:

Desk-checking. This phase, similar to proofreading, is sometimes avoided by the programmer who is looking for a shortcut and is eager to run the program on the computer once it is written. However, with careful desk-checking you may discover several errors and possibly save yourself time in the long run. In desk-checking you simply sit down and mentally trace, or check, the logic of the program to attempt to ensure that it is error-free and workable. Many organizations take this phase a step further with a walkthrough, a process in which a group of programmers-your peers-review your program and offer suggestions in a collegial way.

Translating. A translator is a program that (1) checks the syntax of your program to make sure the programming language was used correctly, giving you all the syntax-error messages, called diagnostics, and (2) then translates your program into a form the computer can understand. A by-product of the process is that the translator tells you if you have improperly used the programming language in some way. These types of mistakes are called syntax errors. The translator produces descriptive error messages. For instance, if in FORTRAN you mistakenly write `N=2 *(I+J))`-which has two closing parentheses instead of one-you will get a message that says, "UNMATCHED PARENTHESES." (Different translators may provide different wording for error messages.) Programs are most commonly translated by a compiler. A compiler translates your entire program at one time. The translation involves your original program, called a

source module, which is transformed by a compiler into an object module. Prewritten programs from a system library may be added during the link/load phase, which results in a load module. The load module can then be executed by the computer.

Debugging. A term used extensively in programming, debugging means detecting, locating, and correcting bugs (mistakes), usually by running the program. These bugs are logic errors, such as telling a computer to repeat an operation but not telling it how to stop repeating. In this phase you run the program using test data that you devise. You must plan the test data carefully to make sure you test every part of the program.