|  |
| --- |
| **FURTHER EDUCATION AND TRAINING CERTIFICATE: INFORMATION TECHNOLOGY: SYSTEMS DEVELOPMENT**  **ID 78965 LEVEL 4 – CREDITS 165** |
| **LEARNER GUIDE**  **SAQA: 14915**  **DESIGN A COMPUTER PROGRAM ACCORDING TO GIVEN SPECIFICATIONS** |

**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:

|  |  |
| --- | --- |
| **http://www.duluth.lib.mn.us/Images/BookStack.gif**  **Books** | **This icon means that other books are available for further information on a particular topic/subject.** |
| http://www.rpsrelocation.com/_borders/checklist.jpg  **References** | **This icon refers to any examples, handouts, checklists, etc…** |
| http://www.school-portal.co.uk/GroupDownloadAttachment.asp?GroupId=21353&AttachmentID=1300079**Important** | **This icon represents important information related to a specific topic or section of the guide.** |
| **http://cloud.graphicleftovers.com/11976/item34004/Cartoon-exercise-book.jpgActivities** | **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.** |
| http://3.bp.blogspot.com/_0EodaYtqevU/TMun5XOj03I/AAAAAAAAAIU/lzrnWelQjgc/s1600/group-discussion.jpg**Exercises** | **This icon represents any exercise to be completed on a specific topic at home by you or in a group.** |
| **http://edtech.kennesaw.edu/intech/images/rubric.gif**  **Tasks/Projects** | **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.** |
| **http://tell.fll.purdue.edu/JapanProj/FLClipart/Adjectives/busy.gifWorkplace 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** |
| http://blog.mindjet.com/wp-content/uploads/2010/01/helpful_tips_image.jpg**Tips** | **This icon indicates practical tips you can adopt in the future.** |
| http://school.discoveryeducation.com/clipart/images/read.gif**Notes** | **This icon represents important notes you must remember as part of the learning process.** |

# Learner Guide Introduction

|  |  |
| --- | --- |
| **About the Learner Guide…** | This Learner Guide provides a comprehensive overview of the **Design a computer program according to given specifications,**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. |
| **Purpose** | The purpose of this Unit Standard is to **Design a computer program according to given specifications** |
| **Outcomes** | **Design a computer program according to given specifications** |
| **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** | The responsibility of learning rest with the learner, so:   * Be proactive and ask questions, * Seek assistance and help from your facilitators, if required. |

Learning Unit1

**UNIT STANDARD NUMBER :** 14915

**Design a computer program according to given specifications**

**LEVEL ON THE NQF :** 4

**CREDITS :** 8

**FIELD :** Physical, Mathematical, Computer and Life Sciences

**SUB FIELD :**  Construction Information Technology and Computer Sciences

|  |  |
| --- | --- |
| **PURPOSE:** | This unit standard is intended:  1. to provide a proficient knowledge of the areas covered.  2. for those entering the workplace in the area of systems development.  Qualifying learners are able to:  1. Apply the fundamental principles of procedural programming design techniques.  2. Demonstrate an understanding of the features of a procedural computer program that will solve a given simple problem.  3. Operate procedural computer program development toolsThe performance of all elements is to a standard that allows for further learning in this area. |
| **LEARNING ASSUMED TO BE IN PLACE:** | |
| Open.  The credit value of this unit is based on a person having the prior knowledge and skills to:  1. be able to apply the principles of Computer Programming (SGB-ID = SDG001). | |

|  |
| --- |
| **SESSION 1.**  **Apply the fundamental principles of program design techniques to the given specification.** |
| **Learning Outcomes** |
| * 1. The application includes the drawing of a program structure diagram for a given simple problem. * 2. The application includes the drawing of a decision tree for a given simple problem. * 3. The application includes the creation of a decision table for a given simple problem * 4. The application allows previously-prepared design technique outputs to be read and desk-checked for accuracy. |

**The application includes the drawing of a program structure diagram for a given simple problem.**

**Key Design Considerations**

**Determine the Application Type**

Choosing the appropriate application type is the key part of the process of designing an application. Your choice is governed by your specific requirements and infrastructure limitations. Many applications must support multiple types of client, and may make use of more than one of the basic archetypes. This guide covers the following basic application types:

* Applications designed for mobile devices.
* Rich client applications designed to run primarily on a client PC.
* Rich Internet applications designed to be deployed from the Internet, which support rich UI and media scenarios.
* Service applications designed to support communication between loosely coupled components.
* Web applications designed to run primarily on the server in fully connected scenarios.

**Determine the Deployment Strategy**

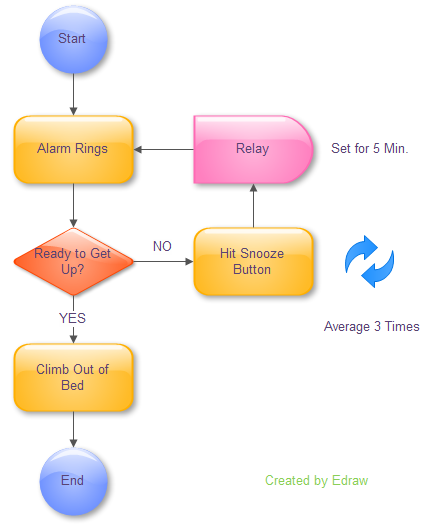
Your application may be deployed in a variety of environments, each with its own specific set of constraints such as physical separation of components across different servers, a limitation on networking protocols, firewall and router configurations, and more. Several common deployment patterns exist, which describe the benefits and considerations for a range of distributed and non-distributed scenarios. You must balance the requirements of the application with the appropriate patterns that the hardware can support, and the constraints that the environment exerts on your deployment options. These factors will influence your architecture design.

**Determine the Appropriate Technologies**

When choosing technologies for your application, the key factors to consider are the type of application you are developing and your preferred options for application deployment topology and architectural styles. Your choice of technologies will also be governed by organization policies, infrastructure limitations, resource skills, and so on. You must compare the capabilities of the technologies you choose against your application requirements, taking into account all of these factors before making decisions.

**A graphic representation of an algorithm, often used in the design phase of programming to work out the logical flow of a program**.

**Examples of Program Structure Diagram**



**Who can use them and how**

* Software developers: Use Real-Time Object-Oriented Modeling (ROOM) notation to model data communication within real-time applications.
* CASE designers: Use transitions to illustrate actor behavior in environments and toward other actors.
* Software architects: Show the reactive behavior of architecture components in a closed system.

**The application includes the drawing of a decision tree for a given simple problem.**

**A decision tree is a kind of flowchart** -- a graphical representation of the process for making a decision or a series of decisions. Businesses use them to determine company policy; sometimes simply for choosing what policy is, other times as a published tool for their employees. Individuals can use decision trees to help them make difficult decisions by reducing them to a series of simpler, or less emotionally laden, choices. Regardless of the context or type of decision, the structure of a decision tree remains the same. Or you can say A decision tree is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm. **Brainstorm each of the variables in the decision you want the decision tree to help you make**. Write them down on a sheet of paper, or in the margin of your main sheet.

[](http://www.wikihow.com/Image:Create-a-Decision-Tree-Step-1.jpg)

**Prioritize the variables you've listed and write them down in order**. Depending on the kind of decision you're making, you can prioritize the variables chronologically, by order of importance, or both.

[](http://www.wikihow.com/Image:Create-a-Decision-Tree-Step-2.jpg)

* For a simple work vehicle, you might prioritize your car decision trees as price, fuel efficiency, model, style and options. If you were buying the car as a gift for your spouse, the priorities might go style, model, options, price, and fuel efficiency.

**Key Points:**

**Decision trees provide an effective method of Decision Making because they:**

* Clearly lay out the problem so that all options can be challenged.
* Allow us to analyze fully the possible consequences of a decision.
* Provide a framework to quantify the values of outcomes and the probabilities of achieving them.
* Help us to make the best decisions on the basis of existing information and best guesses.

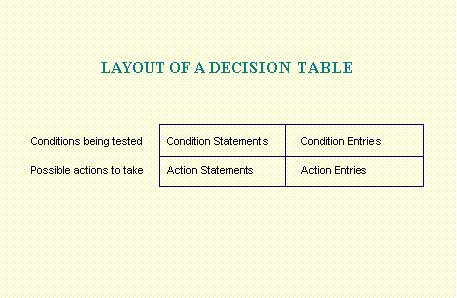
**The application includes the creation of a decision table for a given simple problem**

**Software engineering benefits**

Decision tables, especially when coupled with the use of a domain-specific language, allow developers and policy experts to work from the same information, the decision tables themselves. Tools to render nested if statements from traditional programming languages into decision tables can also be used as a debugging tool. Decision tables have proven to be easier to understand and review than code, and have been used extensively and successfully to produce specifications for complex systems.[

**Method**

The Decision Table is divided into four quadrants.



The upper half lists the conditions being tested; the lower half lists the possible actions to be taken.  Each column represents a certain type of condition or rule.

**Guidelines for constructing a decision table**

**Steps to Develop a Decision Table**

To Construct a Decision Table:

1)         Draw boxes for the top and bottom left quadrants.

2)         List the conditions in the top, left quadrant. When possible, phrase the conditions as questions that can be answered with a Y for yes and an N for no.  This type of Decision Table is known as a limited entry table.  When a Decision Table requires more than two values for a condition, it is known as an extended entry table.

3)         List the possible actions in the bottom, left quadrant.

4)         Count the possible values for each condition and multiply these together to determine how many unique combinations of conditions are present.  Draw one column in the top and bottom right quadrants for each combination.

For example, if there are two conditions and the first condition has two possible values while the second has three possible values, draw six (2 \* 3) columns.

5)         Enter all possible combinations of values in the columns in the top, right quadrant of the table.

6)         For each column (each unique combination of conditions), mark an X in the bottom, right quadrant in the appropriate action row.  The X marks the intersection between the required action and each unique combination of condition values.

**The application allows previously-prepared design technique outputs to be read and desk-checked for accuracy.**

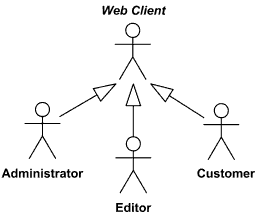
**Determine the Quality Attributes**

Quality attributes—such as security, performance, and usability—can be used to focus your thinking on the critical problems that your design should solve. Depending on your requirements, you might need to consider every quality attribute, or you might only need to consider a subset. For example, every application design must consider security and performance, but not every design needs to consider interoperability or scalability. Understand your requirements and deployment scenarios first so that you know which quality attributes are important for your design. Keep in mind that quality attributes may conflict; for example, security often requires a tradeoff against performance or usability.

|  |
| --- |
| **SESSION 2.**  **Demonstrate an understanding of the features of a computer program.** |
| **Learning Outcomes** |
| * 1. The demonstration includes the research of a problem in terms of inputs and outputs. * 2. The demonstration includes the features of a procedural computer program that will solve the given problem. * 3. The demonstration outlines why a batch or online program will be the best solution to the problem |

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

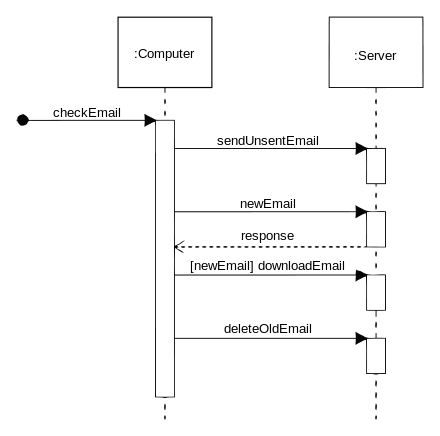
An **actor** in the Unified Modeling Language (UML) "specifies a role played by a user or any other system that interacts with the subject. "An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is external to the subject. “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:

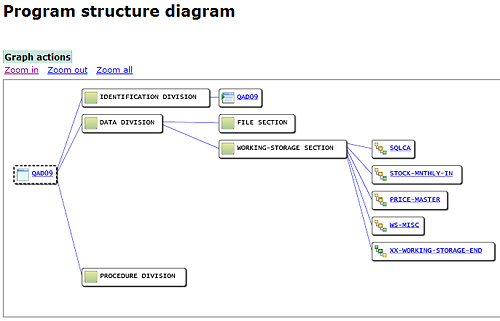
* Requirement analysis and high level design.
* Model the context of a system.
* Reverse engineering.
* Forward engineering.

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 demonstration includes the features of a procedural computer program that will solve the given problem.**

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 sub activities as they are used in organization



**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 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**

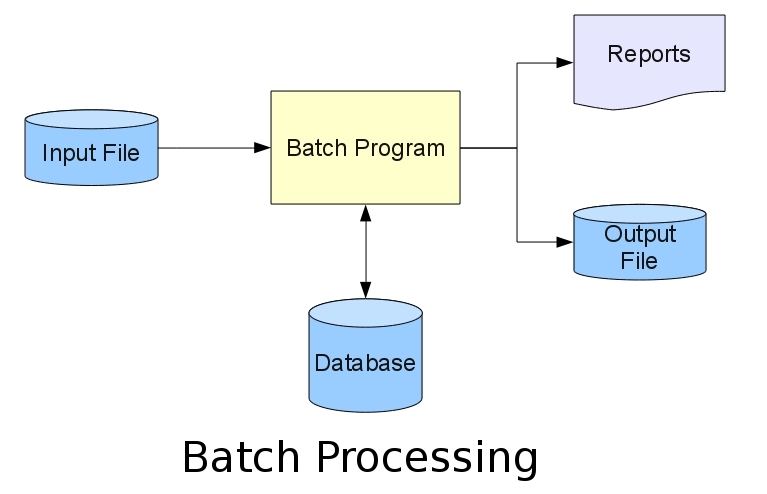
First it takes advantage of the sequential structure of decision tree branches so that the order of checking conditions and executing actions is immediately noticeable. Second, Conditions and actions of decision trees are found on some branches but not on others which contrasts with decision tables, 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.

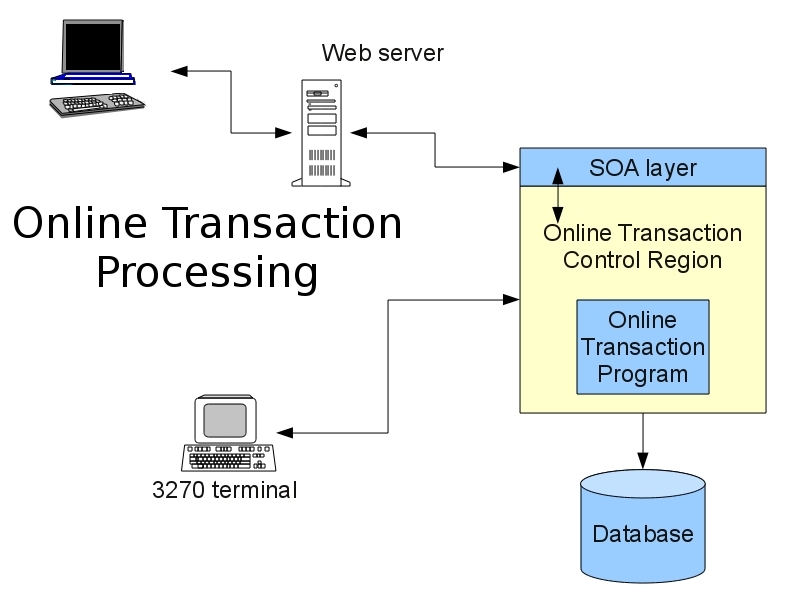
**The demonstration outlines why a batch or online program will be the best solution to the problem**

**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.





|  |
| --- |
| **SESSION 3.**  **Demonstrate an understanding of how to document program designs using appropriate tools.** |
| **Learning Outcomes** |
| * 1. The operation demonstrates the use of the editor of the development tools to produce procedural program source code * 2. The operation includes the use of the syntax checker of the tools to check for syntax errors. * 3. The operation uses the tool to compile the procedural source code produced. |

**The operation demonstrates the use of the editor of the development tools to produce program source code.**

A **programming tool or software development tool** is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs, that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object. Sometimes called text editor, a program that enables you to create and edit text files. There are many different types of editors, but they all fall into two general categories: line editors: A primitive form of editor that requires you to specify a specific line of text before you can make changes to it. screen -oriented editors: Also called full-screen editors, these editors enable you to modify any text that appears on the display screen by moving the cursor to the desired location.

**EDITOR COMMANDS**

|  |  |
| --- | --- |
| **Command** | **Description** |
| Ctrl-a [Home] | Moves the cursor to the beginning of the current line. |
| Ctrl-b [Left Arrow] | Moves the cursor backwards one character. |
| Ctrl-c | Copys highlighted text (the current selection) to a temporary holding area. |
| Ctrl-d ( [Delete] on Windows ) | Deletes the character to the right of the cursor. |
| Ctrl-e [End] | Moves the cursor to the end of the current line. |
| Ctrl-f | Find a sequence of characters. A prompt bar pops up for entering the desired sequence of characters. An [Esc] aborts the find operation. |
| Ctrl-g | Find the next occurance of a sequence of characters, specified by last FIND or SEARCH. |

**The operation includes the use of the syntax checker of the tools to check for syntax errors.**

In computer science, **a syntax error** refers to an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language. For compiled languages syntax errors occur strictly at compile-time. A program will not compile until all syntax errors are corrected. For interpreted languages, however, not all syntax errors can be reliably detected until run-time, and it is not necessarily simple to differentiate a syntax error from a semantic error; many don't try at all. In 8-bit home computers that used BASIC interpreter as their primary user interface, the SYNTAX ERROR error message became somewhat notorious, as this was the response to any command or user input the interpreter couldn't parse. A syntax error may also occur when an invalid equation is entered into a calculator. This can be caused, for instance, by opening brackets without closing them, or less commonly, entering several decimal points in one number.

**Syntax Checking Error Messages**

These messages are output when the compiler is checking your COBOL program for syntax and consistency. The descriptions for each message lists the text of each message, and where necessary explain the error or problem that causes the message and gives advice on how to prevent it. The severity is not listed, as the same message can be output with a different severity depending on the setting of directives.

**Format of Syntax Checking Error Messages**

Syntax checking error messages have the following format:

*Line-of-COBOL-code*

*nnnn-s* *code*\*\*\*\* (*mmmm*)\*\*

*message*

where the variables are:

|  |  |
| --- | --- |
| *nnnn* | The message number. |
| *mmmm* | The page where the previous error occurred. |
| *s* | One of the following severity codes:   |  |  | | --- | --- | | U | Unrecoverable. An unrecoverable error stops the COBOL system. These messages are produced by the run-time system. | | S | Severe. You must correct the syntax error or inconsistency in your program. Otherwise the compiler cannot generate code. | | E | Error. The compiler will make an assumption about what you meant. You might want to correct your program in case the compiler's assumption is not what you intended. | | W | Warning. This means there might be an error, although the program is syntactically correct. | | I | Information. This draws your attention to something in the source code that you might need to be aware of. It does not mean there is an error. | |

You can disable reporting of errors of E-level, W-level, and I-level, using the WARNING directive. When the Compiler has finished, the total number of errors in each category is also output. You can disregard some levels of errors and continue working. You can:

* Debug programs that have S-level, E-level, W-level, and I-level errors regardless of the setting of the E run-time switch.
* Produce object code from intermediate code that has E-level, W-level, and I-level errors, but not S-level errors.
* Run programs that have E-level, W-level, and I-level errors. If the E-level run-time switch is on, which overrides the default setting, you can also run programs with S-level errors.

The error messages can contain variable information. This information is indicated as an item in italics. For example:

User-name *data-name* not unique will have the name of the item that is not unique in place of the text *data-name*.

**List of Syntax Checking Error Messages**

**0001 Undefined error. Inform Technical Support**

Your program contains an error which the COBOL system has failed to recognize.

**Resolution:**  
Send Technical Support a copy of your source code to enable them to find the cause of the error.

**0002 Unexpected SQL error. Inform Technical Support**

Your program contains an SQL error which the COBOL system has failed to recognize.

**Resolution:**  
Send Technical Support a copy of your source code to enable them to find the cause of the error.

**0003 Illegal format: Literal**

The sequence of characters forming a literal in your source code does not conform to the rules governing the construction of such names. A literal can be either nonnumeric or numeric. If numeric it can be up to 18 digits in length, but it must not contain more than one sign character or more than one decimal point. A nonnumeric literal can consist of any allowable character in the computer's character set up to a maximum of 160 characters in the Procedure Division, or 2048 characters in the Data Division. A nonnumeric literal must be enclosed in quotation marks. If you have used a figurative constant as the literal make sure that it is referenced by an allowable reserved word (such as ZERO) which you have spelled correctly. A figurative constant and a numeric literal must not be enclosed in quotation marks. You might also have used the wrong class of literal for the context of the sentence. Alternatively, if you have used the figurative constant ALL in your code, you have not coded it in accordance with the rules governing the use of this constant. ALL must be followed by a nonnumeric literal and not by a numeric one.

**Resolution:**  
Revise your code to comply with the above rules.

**0004 Illegal character**

Your program contains a character that is not part of the COBOL language set.

**Resolution:**  
Replace the illegal character with a valid one.

**0005 User-name *user-name* not unique**

You have given the same user-name without qualification to more than one data item or procedure-name in your source code.

**Resolution:**  
You must rename or qualify the duplicated data items or procedure-names to ensure that uniqueness of reference is achieved.

**0007 specified in column 7 of otherwise blank line**

The indicator area, column 7, contains an illegal character.

**Resolution:**  
Legal characters are \*, D, -, / or space.

**0008 Unknown COPY file *filename* specified**

A file with the name *filename*, specified in conjunction with a COPY statement, cannot be found.

**0009 '.' missing**

Your code does not contain a period in a place where one is expected by the rules of COBOL syntax.

**Resolution:**  
Insert one at the relevant place.

**0010 Word starts or is continued in wrong area of source line**

The word starts either in area A when it should have started in area B, or in area B when it should have started in area A.

**0011 Reserved word missing or incorrectly used**

You have either used a reserved word in a place where a user defined word is expected or you have failed to use a reserved word where one is needed.

**Resolution:**  
Alter the reserved word into a user defined one or insert a reserved word according to the context of this message.

**The operation uses the tool to compile the program source code produced.**

User written code, standard functions, library functions.

**Library Functions:**

Q-Basic provides a number of functions. These inbuilt functions of Q-basic are called library functions. These are divided into string and numeric functions. LEFT$, LEN, MID$, LCASE$ etc are the examples of string functions and ABS, SQR, INT, VAL etc are the examples of numeric variables.

**User Defined Functions:**

While standard functions are pre-defined and provided for by QBasic, user-defined functions are completely defined and customized by the programmer. User-defined functions return a single value and are generally used to perform an operation that will be needed numerous times in a program. In QBasic, user-defined functions are referred to as procedures; similar to SUB procedures except function procedures return one value. Arguments may be sent into a function procedure for use in the function operation, but the value returned by the function will not be included in the parameter list. The value is returned in the function itself. Each user=defined function starts with BEGIN FUNCTION Funct Name (x,y,z) and ends with END FUNCTION. The code between these two lines is executed whenever the function is invoked from main program, from another function, or SUB, or from itself. Funct Name is a name for your function (choose a descriptive one). Arguments (x,y,z) are the variables passed to the function. The form of a function procedure is as follows:

**FUNCTION name ( parameter list )**

**REM**

**REM body of function**

**REM**

**END FUNCTION**

**Subroutines and functions:**

A subroutine (also called a "module") is a "mini-program" inside your program. In other words, it is a collection of commands--and can be executed anywhere in your program. To create a subroutine: Go to the "Edit" menu Select "New Sub" Enter a name for the subroutine Type a list of commands between SUB and END SUB. (Topic 1.1 SUB ….. END SUB statement: will provide detail information)

**Functions:**

Function is the same as a subroutine, except it returns a value. Also, you must leave out the CALL command. To return a value, set a variable with the same name as the function.

**Local and Global variable:**

When a variable is declared within a main module or procedure without using SHARED attribute, only code within that main module or procedure can access or change the value of that variable. This type of variable is called as LOCAL variable. When a variable is declared with SHARED attribute in a main module, it can be used in a procedure without passing it as parameter. Any SUB or FUNCTION procedure within the module can use this type of variable. This type of variable which is available to all SUB and FUNCTION procedure with the module is known as GLOBAL variable.

|  |
| --- |
| **SESSION 4.**  **Apply fundamental principles of problem analysis.** |
| **Learning Outcomes** |
| * The application provides an appreciation of the steps and techniques of program maintenance. * The application provides examples to demonstrate different problem analysis techniques (at least 2). * The application uses logic flow techniques to solve given elementary problems. |

**The application 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 1960�s and 70�s 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.

**Description identifies different problem analysis techniques (at least 2).**

**Problem definition, Hierarchical (Top-Down) approach, Stepwise refinement, Modularity.**

**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, 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.

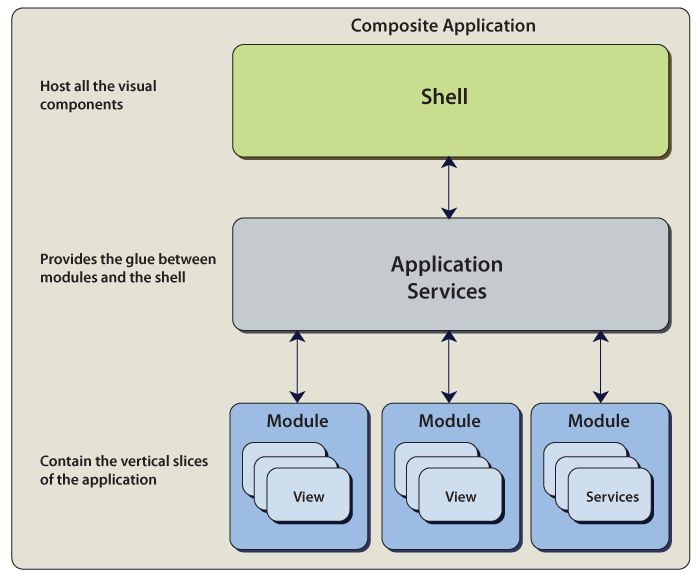
**Product design and development**

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.[]](http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design#cite_note-3) 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

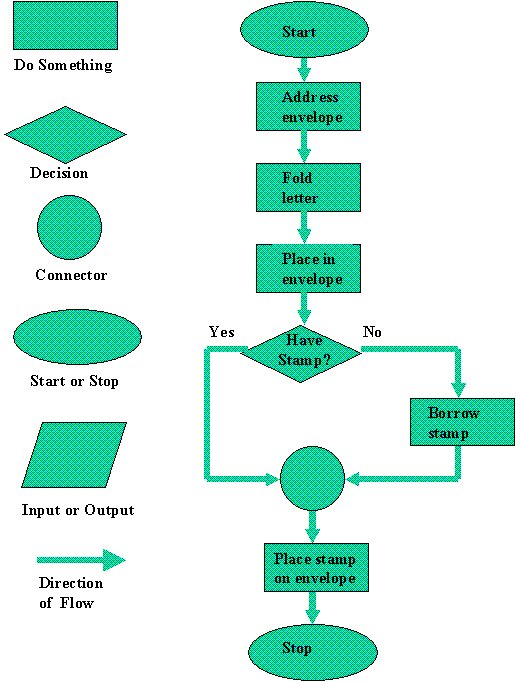
**Why Choose a Modular Design?**

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 application uses logic flow techniques to solve given elementary problems.**

1. **Planning the Solution**



Two common ways of planning the solution to a problem are to draw a flowchart and to write pseudocode, or possibly both. Essentially, a flowchart is a pictorial representation of a step-by-step solution to a problem. It consists of arrows representing the direction the program takes and boxes and other symbols representing actions. It is a map of what your program is going to do and how it is going to do it. The American National Standards Institute (ANSI) has developed a standard set of flowchart symbols. Figure 1 shows the symbols and how they might be used in a simple flowchart of a common everyday act-preparing a letter for mailing.  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. We will illustrate these later in this chapter, when we focus on language examples.

**Coding the Program**

As the programmer, your next step is to code the program-that is, to express your solution in a programming language. You will translate the logic from the flowchart or pseudocode-or some other tool-to a programming language. As we have already noted, a programming language is a set of rules that provides a way of instructing the computer what operations to perform. There are many programming languages: BASIC, COBOL, Pascal, FORTRAN, and C are some examples. Although programming languages operate grammatically, somewhat like the English language, they are much more precise. To get your program to work, you have to follow exactly the rules-the syntax-of the language you are using. Of course, using the language correctly is no guarantee that your program will work, any more than speaking grammatically correct English means you know what you are talking about. The point is that correct use of the language is the required first step. Then your coded program must be keyed, probably using a terminal or personal computer, in a form the computer can understand.   
  
One more note here: Programmers usually use a text editor, which is somewhat like a word processing program, to create a file that contains the program. However, as a beginner, you will probably want to write your program code on paper first.

1. **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.