COURSE GUIDE

CIT 432
SOFTWARE ENGINEERING

Course Team
Dr. B.C.E. Mbam (Course Developer/Writer) - EBSU
Dr. Juliana Ndunagu (Programme Leader/Coordinator) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

© 2022 by NOUN Press
National Open University of Nigeria
Headquarters
University Village
Plot 91, Cadastral Zone
Nnamdi Azikiwe Expressway
Jabi, Abuja

Lagos Office
14/16 Ahmadu Bello Way
Victoria Island, Lagos

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Printed 2022

ISBN: 978-058-335-1

CONTENTS PAGE

Introduction
Software Engineering is a second semester course. It is a two credit degree course available to all students offering **……………………………………………**
The course consists of 17 units which will enable you to develop the skills necessary for you to develop, operate and maintain software. They are no compulsory pre-requisites to it, although it is good to have a basic knowledge of operating computer.

CIT432:

What You Will Learn in This Course
This Course consists of units and a course guide. This course guide tells you briefly what the course about, what course materials you will be using and how you can work with these materials. In addition, it advocates some general guidelines for the amount of time you are likely to spend on each unit of the course in order to complete it successfully.

It gives you guidance in respect of your Tutor-Marked Assignment which will be made available in the assessment available. There will be regular tutorial classes that are related to the course. It is advisable for you to attend these tutorial sessions. The course will prepare you for challenges you will meet in the field of software engineering.

Course Aim

The aim of the course is simple. The couse aims to provide you with an understanding of Software Engineering; it also aims to provide you with solutions to problem in software as a whole.

Course Objectives

To achieve the aims set out, the course has a set of objectives which are included at the beginning of the unit. You should read these objectives before you study the unit. You may wish to refer to them during your study to check to check on your progress. You should always look at the nit objectives after completion of each unit. By doing so, you would have followed the instruction in the unit.

Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aims of the course as a whole. In addition to the aims above, this course sets to achieve some objectives. Thus, after going through the course, you be able to:

Explain the basic concept of software
Explain what software engineering is
Trace the history of software engineering.
Explain who a software engineer is
Explain the software crisis.
Give an overview of software development.
Explain software development life cycle model.
Explain the concept of Modularity.
Explain Pseudo code.
Explain programming environment.
Explain Case Tools.
Explain Hipo .
Explain Implementation and Testing
Explain Software Quality Assuarance.
Explain Compatibility.
Explain Software verification and Validation

Working through This Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self assessment exercises and tutor - marked assignments, and at some point in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Materials

The major components of the course are:

1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

Study Units

The study units in this course are as follows:

MODULE 1 BASIC CONCEPT OF SOFTWARE

Unit 1	Computer Software
Unit 2	What is Software Engineering
Unit 3	History of Software Engineering
Unit 4	Software Engineer
Unit 5	Software Crisis

MODULE 2 SOFTWARE DEVELOPMENT

Unit 1	Overview of Software Development
Unit 2	Software Development Life Cycle Model
Unit 3	Modularity
Unit 4	Pseudocode
Unit 5	Programming Enviroment Case Tools and Hipo Diagram

MODULE 3 IMPLEMENTATION AND TESTING

Unit 1	Implementation
Unit 2	Testing Phase
Unit 3	Software Quality
Unit 4	Compatibility
Unit 5	Verification

MODULE 4: FORMAL METHODS

Unit 1:General Information

Unit 2: Introduction to Formal Methods

Unit 3:Approaches to formal methods and their use in software development

Unit 4: Proposition

Unit 5: Predicates

Unit 6: Sets

Unit 7: Series or Sequence

Each unit consists of one or two weeks's work and include an introduction, objectives, reading materials, conclusion, summary, Tutor Marked Assignment (TMAs), references and other resources. The unit directs you to work on execises related to the required reading. In general, these exrcises test you on the materials you have just covered or required you to apply it in some way and thereby assist you to evaluate your progress and to reinforce your comprehension of the material. In addition to TMAs, these exercises will help you in achieving the stated learning objectives of rhe individual units and of the course as a whole.

Presentation Schedule

Your course materials have important dates for the early and timely completion and submission of your TMAs and attending tutorials. You should remember that you are required to submit all your assignments by the stipulated time and date. You should guard against falling behind in your work.

Assessment

There are three aspects to the assessment of the course. First is made up of self-assessment exercises, second consists of the Tutor_Marked Assigment and third is the written examination/end of course examination.

You are advised to do the exercises. In tackling the assignments, you are expected to apply information, knowledge and techniques you gathered during the course. The assignments must be submitted to your facilitator for formal assessments in accordance with the deadlines stated in the presentation schedule and the assignment file. The work you submit to your tutor for assessment will count for 30% of your total course work. At the end of the course you will need to sit for a final or end of course examination of about a three hour duration. This examination will count for
70% of your total course mark.

Tutor-Marked Assignment

The TMA is a continuous assessment component of your course. It accounts for 30 % of the total score. You will be given four (4) TMAs to answer. Three of these must be answered before you are allowed to sit for the end of course examination. The TMAs would be given to you by your facilitator and returned after you have done the assignment. Assignment questions for the units in this course are contained in the assignment file. You will be able to complete your assignment from the information and the material contained in your reading, references and the study units. However, it is desirable in all degree level of education to demostrrate that you have read and researched more into your references, which will give you a wider view point and may provide you with a deeper understanding of the subject.

Make sure that each assignment reaches your facilitator on or before the deadline given in the presentation schedule and assignment file. If for any reason you can not complete your work on time, contact your facilitator before the assignment is due to discuss the possibility of an extension. Extension will not be granted after the due date unless there are exceptional circumstances.

Final Examination and Grading

The end of your examination for Software Engineering will be for about 3 houurs and it has a value of 70% of the total course work. The examination will consist of questions, which will reflect the type of self-testing, practice exercise and tutor- marked assignment problems you are previously encountered. All areas of the course will be assessed.

You ate to use the time between finishing the last unit and sitting for the examination to revise the whole course. You might find it useful to review your self-test, TMAs and comments on them before the examination. The end of course examination covers information from all parts of the course.

Course Marking Scheme

Assignment	Marks
Assignment 1-4	Four assignments, best three marks of the four count at 10% each- 30% of course marks
End of course examination	70% of overall course marks
Total	100% of course materials.

Facilitator/Tutor and Tutorials

There are 16 hours of tutorials provided in support of the course. You will be notified of the dates, times and location of these tutorials as well as the name and phone number of your facilitator, as soon as you as you are allocated a tutorial group.

Your facilitator will mark and comment on your assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail you Tutor Marked Assignment to your facilitator before the schedule date. |(at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

Do not delay to contact your facilitator by telephone or e-mail if you need assistance

The following might be the circumstances in which you would find assistance necessary, you would have to contact your facilitator if :

Understand any part of the study or assigned reading
You have difficulty with the self- tests
You have a question or problem with an assignment or with the grading of an assignment

You should endeavour to attend the tutorials. This is the only chance to have face to face contact with your course facilitator and to ask question which are answered instantly. You can raise any problem encountered in the course of your study.

To gain much benefits from the course tutorials, prepare a question list before attending them. You will learn a lot from participating actively in the discussions.

MODULE 1: Basic Concept of Software Engineering

Unit 1: Computer software

1.0 Introduction

The Computer system has two major components namely hardware and software. The hardware component is physical (can be touched or held). The non physical part of the computer system is the software. As the voice of man is non physical yet it so important for the complete performance of man, so is the software. In this unit, the categories of software are examined.

2.0 Objectives

By the end of this unit, you will be able to:

Define what software is

Differentiate between System, Application and programming Software.

Explain the role of System Software.

3.0 Definition of software

Computer software is a general name for all forms of programs. A program itself is a sequence of instruction which the computer follows to perform a given task.

3.1 Types of software

Software can be categorised into three major types namely **system software**, programmingsoftware and application software..

3.1.2 System software

System software helps to run the computer hardware and the entire computer system. It includes the following:

device drivers
operating systems
servers
utilities
windowing systems

The function of systems software is to assist the applications programmer from the details of the particular computer complex being used, including such peripheral devices as communications, printers, readers, displays and keyboards, and also to partition the computer's resources such as memory and processor time in a safe and stable manner.

3.1.3 **Programming software**

Programming software offers tools to assist a programmer in writing programs, and software using different programming languages in a more convenient way. The tools include:

compilers
debuggers
interpreters
linkers
text editors

3.1.4 Application software

Application software is a class of software which the user of computer needs to accomplish one or more definite **tasks**. The common applications include the following:

industrial automation
business software
computer games
quantum chemistry and solid state physics software
telecommunications (i.e., the internet and everything that flows on it)
databases
educational software
medical software
military software
molecular modeling software
photo-editing
spreadsheet
Word processing
Decision making software

Activity A Differentiate between hardware and software

4.0 Conclusion

A major component of computer system is the software and it plays a major role in the functioning of the system.

5.0 Summary

In this unit we have learnt that:
Computer software is a general name for all forms of programs.
System software helps run the computer hardware and computer system. Programming software offers tools to assist a programmer in writing programs.

Application software is a class of software which the user of computer needs to accomplish one or more definite tasks.
Briefly explain the role of system software

6.0 Tutor Marked Assignment

1 What is Software?

2 With four (4) examples each, differentiate between System and Application software

3 What is Programming software? Give five (5) examples

7.0 Further Reading and Other Resources

Hally, Mike (2005:79). *Electronic brains/Stories from the dawn of the computer age*. British Broadcasting Corporation and Granta Books, London. ISBN 1-86207-663-4.

GNU project: "Selling Free Software": "we encourage people who redistribute free software to charge as much as they wish or can."

Engelhardt, Sebastian (2008): "The Economic Properties of Software", Jena Economic Research Papers, Volume 2 (2008), Number 2008-045. (in Adobe pdf format)

UNIT 2`: What is Software Engineering?

1.0 Introduction

Software Engineering is the application of engineering to software. This unit looks at its goals and principles

2.0 Objectives

By the end of this unit, you will be able to:

Define what software engineering is
Explain the goals of software engineering
Explain the principles of software engineering.

3.0 Definition of Software Engineering.

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches. In other words, it is the application of engineering to software.

3.1 **Sub-disciplines of Software engineering**

Software engineering can be divided into ten sub-disciplines. They are as follows:

Software requirements: The elicitation, analysis, specification, and validation of requirements for software.
Software design: Software Design consists of the steps a programmer should do before they start coding the program in a specific language.It is usually done with Computer-Aided Software Engineering (CASE) tools and use standards for the format, such as the Unified Modeling Language (UML).
Softwaredevelopment: It is construction of software through the use of programming languages.
Softwaretesting **Software Testing** is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test.
Software maintenance: This deals with enhancements of Software systems to solve the problems the may have after being used for a long time after they are first completed..
Software configuration management: is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines.
Software engineering management: The management of software systems borrows heavily from projectmanagement.

Softwaredevelopmentprocess: A **software development process** is a structure imposed on the development of a software product. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.
Software engineering tools, (CASE which stands for Computer Aided Software Engineering) CASE tools are a class of software that automates many of the activities involved in various life cycle phases.
Software quality The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

3.2 Software Engineering Goals and Principles

3.2.1 Goals

Stated requirements when they are initially specified for systems are usually incomplete. Apart from accomplishing these stated requirements, a good software system must be able to easily support changes to these requirements over the system's life. Therefore, a major goal of software engineering is to be able to deal with the effects of these changes. The software engineering goals include:

Maintainability: Changes to software without increasing the complexity of the original system design should be possible.

Reliability: The software should be able to prevent failure in design and construction as well as recover from failure in operation. In other words, the software should perform its intended function with the required precision at all times.

Efficiency: The software system should use the resources that are available in an optimal manner.

Understand ability*:* The software should accurately model the view the reader has of the real world. Since code in a large, long-lived software system is usually read more times than it is written, it should be easy to read at the expense of being easy to write, and not the other way around.

3.2.2 Principles

Sounds engineering principles must be applied throughout development, from the design phase to final fielding of the system in order to attain a software system that satisfies the above goals. These include:

Abstraction: The purpose of abstraction is to bring out essential properties while omitting inessential detail. The software should be organized as a ladder of abstraction in which each level of abstraction is built from lower levels. The code is sufficiently conceptual so the user need not have a great deal of technical background in the subject. The reader should be able to easily follow the logical path of each of the various modules. The decomposition of the code should be clear.

Information Hiding: The code should include no needless detail. Elements that do not affect other segment of the system are inaccessible to the user, so that only the intended operations can be performed. There are no "undocumented features".

Modularity: The code is purposefully structured. Components of a given module are logically or functionally dependent.

Localization: The breakdown and decomposition of the code is rational. Logically related computational units are collected together in modules.

Uniformity*:* The notation and use of comments, specific keywords and formatting is consistent and free from unnecessary differences in other parts of the code.

Completeness: Nothing is deliberately missing from any module. All important or relevant components are present both in the modules and in the overall system as appropriate.

Confirm ability*:* The modules of the program can be tested individually with adequate rigor. This gives rise to a more readily alterable system, and enables the reusability of tested components.

Activity B **1** What is software engineering

2 Explain briefly the Sub-disciplines of Software engineering

4.0 Conclusion

Software Engineering as the application of engineering to software has overall goal to easily support changes to software requirements over the system's life. It is also characterised with sounds engineering principles which must be applied throughout development, from the design phase to final fielding of the system in order to attain a software system that satisfies the overall goal

5.0 Summary

In this unit, we have learnt that:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of

software, and the study of these approaches. In other words, it is the application of engineering to software.
The goals of Software engineering include: Maintainability, Reliability, Efficiency, Understand ability.
The principles of software engineering include: Abstraction, Information Hiding, Modularity, Localization, Uniformity, Completeness, Confirm ability

6.0 Tutor Marked Assignment

1 Discuss the goals of software engineering

2 Discuss the principles of software engineering

7.0 **Further Reading and Other Resources**

"The mythical man-month", Frederick P. Brooks, Jr., Anniversary Edition, Addison-Wesley, 1995

"Fundamentals of software engineering", Carlo Ghezzi et al, Prentice-Hall, 1991

"Software engineering: A practitioner's approach", Roger S. Pressman, Third Edition, McGraw-Hill, 1992

"Classical and object-oriented software engineering", Stephen R. Schach, Third Edition, Irwin, 1996

"Software Engineering", Ian Sommerville, Fifth Edition, Addison-Wesley 1996

Unit 3 History of Software Engineering

1.0 Introduction

This unit traces the historical development of software engineering from 1968 till date.

2.0 Objectives

By the end of this unit, you will be able to:

Explain the historical development of software engineering.

3.0 Overview of Software Engineering.

In the 1968, software engineering originated from the NATO Software Engineering Conference. It came at the time of software crisis. The field of software engineering has since then been growing gradually as a study dedicated to creating qualified software. In spite of being around for a long time, it is a relatively young field compared to other fields of engineering. Though some people are still confused whether software engineering is actually engineering because software is more of invisible course. Although it is disputed what impact it has had on actual software development over the last more than 40 years, the field's future looks bright according to Money Magazine and Salary.com who rated "software engineering" as the best job in America in 2006.

The early computers had their software wired with the hardware thereby making them to be inflexible because the software could not easily be upgraded from one machine to another. This problem necessitated the development. Programming languages started to appear in the 1950s and this was also another major step in abstraction. Major languages such as FORTRAN, ALGOL, and COBOL were released in the late 1950s to deal with scientific, algorithmic, and business problems respectively. E.W. Dijkstra wrote his seminal paper, "Go To Statement Considered Harmful", in 1968 and David Parnas introduced the key concept of modularity and information hiding in 1972 to help programmers deal with the ever increasing complexity of software systems. A software system for managing the hardware called an operating system was also introduced, most notably by Unix in 1969. In 1967, the Simula language introduced the object-oriented programming paradigm.

The technological advancement in software has always been driven by the ever changing manufacturing of various types of computer hardware. The more the new technologies upgrade, from vacuum tube to transistor, and to microprocessor were emerging, the more the necessity to upgrade and even write new software. In the mid 1980s software experts had a consensus for centralised construction of software with the use of software development Life Cycle from system analysis. This period gave birth to object-oriented programming languages. Open-source software started to appear in the early 90s in the form of Linux and other software introducing the "bazaar" or decentralized style of constructing software.[10] Then the Internet and World Wide Web hit in the mid 90s changing the engineering of software once again. Distributed Systems gained sway as a way to design systems and the Java programming language was introduced as another step in abstraction having its own virtual machine. Programmers collaborated and wrote

the Agile Manifesto that favored more light weight processes to create cheaper and more timely software.

3.1 Evolution of Software Engineering

There are a number of areas where the evolution of software engineering is notable:

Professionanism: The early 1980s witnessed software engineering becoming a full-fledged profession like computer science and other engineering fields.

Impact of women: In the early days of computer development (1940s, 1950s, and 1960s,), the men were found in the hardware sector because of the mental demand of hardwaring heavy duty equipment which was too strenuous for women. The witing of software was delegated to the women. Some of the women who were into many programming jobs at this time include Grace Hopper and Jamie Fenton. Today, many fewer women work in software engineering than in other professions, this reason for this is yet to be ascertained.

Processes: Processes have become a great part of software engineering and re praised for their ability to improve software and sharply condemned for their potential to narrow programmers.

Cost of hardware: The relative cost of software versus hardware has changed substantially over the last 50 years. When mainframes were costly and needed large support staffs, the few organizations purchasing them also had enough to fund big, high-priced custom software engineering projects. Computers can now be said to be much more available and much more powerful, which has a lot of effects on software. The larger market can sustain large projects to create commercial packages, as the practice of companies such as Microsoft. The inexpensive machines permit each programmer to have a terminal capable of fairly rapid compilation. The programs under consideration can use techniques such as garbage collection, which make them easier and faster for the programmer to write. Conversely, many fewer organizations are concerned in employing programmers for large custom software projects, instead using commercial packages as much as possible.

3.2 The Pioneering Era

The most key development was that new computers were emerging almost every year or two, making existing ones outdated. Programmers had to rewrite all their programs to run on these new computers. They did not have computers on their desks and had to go to the "computer room" or “computer laboratory”. Jobs were run by booking for machine time or by operational staff. Jobs were run by inserting punched cards for input into the computer’s card reader and waiting for results to come back on the printer.

The field was so new that the idea of management using schedule was absent. Guessing the completion time of project predictions was almost unfeasible Computer hardware was application-based. Scientific and business tasks needed different machines. High level languages like FORTRAN, COBOL, and ALGOL were developed to take care of the need to frequently translate old software to meet the needs of new machines. Systems software was given out for free by the vendors since it must to be installed in the computer before it is sold. Custom software was sold by a few companies but no sale of packaged software.

Organisation such as like IBM's scientific user group SHARE gave out software free and as a result reuse was order of the day. Academia did not yet teach the principles of computer science. Modular programming and data abstraction were already being used in programming.

3.3 1945 to 1965: The origins

The term *software engineering* came into existence in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering but fount it difficult to marry engineering with software.

In 1968 and 1969, two conferences on software engineering were sponsored by the NATO Science Committee. This gave the field its initial boost. It was widely believed that these conferences marked the official start of the profession of *software engineering*.

3.4 1965 to 1985: The software crisis

Software engineering was prompted by the *software crisis* of the 1960s, 1970s, and 1980s. It was the crisis that identified many of the problems of software development. This era was also characterised by: run over budget and schedule, property damage and loss of life caused by poor project management. Initially the software crisis was defined in terms of productivity, but advanced to emphasize quality.

Cost and Budget Overruns: The OS/360 operating system was a classic example. It was a decade-long project from the 1960s and eventually produced one of the most complex software systems at the time.
Property Damage: Software defects can result in property damage. Poor software security allows hackers to steal identities, costing time, money, and reputations.
Life and Death: Software defects can kill. Some embedded systems used in radiotherapy machines failed so disastrously that they administered poisonous doses of radiation to patients. The most famous of these failures is the *Therac 25* incident.

3.5 1985 to 1989: No silver bullet

For years, solving the software crisis was the primary concern for researchers and companies producing software tools. Apparently, they proclaim every new technology and practice from the 1970s to the 1990s as a *silver bullet* to solve the software crisis. Tools, discipline, formal methods, process, and professionalism were published as silver bullets:

Tools: Particularly underline tools include: Structured programming, object-oriented programming, CASE tools, Ada, Java, documentation, standards, and Unified Modeling Language were touted as silver bullets.
Discipline: Some pundits argued that the software crisis was due to the lack of discipline of programmers.
Formal methods: Some believed that if formal engineering methodologies would be applied to software development, then production of software would become as predictable an industry as other branches of engineering. They advocated proving all programs correct.
Process: Many advocated the use of defined processes and methodologies like the Capability Maturity Model.
Professionalism: This led to work on a code of ethics, licenses, and professionalism.

Fred Brooks (1986), *No Silver Bullet* article, argued that no individual technology or practice would ever make a 10-fold improvement in productivity within 10 years.

Debate about silver bullets continued over the following decade. Supporter for Ada, components, and processes continued arguing for years that their favorite technology would be a silver bullet. Skeptics disagreed. Eventually, almost everyone accepted that no silver bullet would ever be found. Yet, claims about *silver bullets* arise now and again, even today.

" No silver bullet" means different things to different people; some *take" no silver bullet" to mean* that software engineering failed. The pursuit for a single key to success never worked. All known technologies and practices have only made incremental improvements to productivity and quality. Yet, there are no silver bullets for any other profession, either. Others interpret *no silver bullet* as evidence that software engineering has finally matured and recognized that projects succeed due to hard work.

However, it could also be pointed out that there are, in fact, a series of *silver bullets* today, including lightweight methodologies, spreadsheet calculators, customized browsers, in-site search engines, database report generators, integrated design-test coding-editors with memory/differences/undo, and specialty shops that generate niche software, such as information websites, at a fraction of the cost of totally customized website development. Nevertheless, the field of software engineering looks as if it is too difficult and different for a single "silver bullet" to improve most issues, and each issue accounts for only a small portion of all software problems.

3.6 1990 to 1999: Importance of the Internet

The birth of internet played a major role in software engineering. With its arrival, information could be gotten from the World Wide Web speedily. Programmers could handle illustrations, maps, photographs, and other images, plus simple animation, at a very fast rate.

It became easier to display and retrieve information as a result of the usage of browser on the HTML language. The widespread of network connections brought in computer viruses and worms on MS Windows computers. These new technologies brought in a lot good innovations such as e-mailing, web-based searching, e-education to to mention a few. As a result, many software systems had to be re-designed for international searching. It was also required to translate the information flow in multiple foreign languages Many software systems were designed for multi-language usage, based on design concepts from human translators.

3.7 2000 to Present: Lightweight Methodologies

This era witnessed increasing demand for software in many smaller organizations. There was also the need for inexpensive software solutions and this led to the growth of simpler, faster methodologies that developed running software, from requirements to deployment. There was a change from rapid-prototyping to entire *lightweight methodologies.* For example, Extreme Programming (XP), tried to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing, vast number of small software systems.

3.8 What is it today

Software Engineering as a profession is now being defined as a field of human experts in boundary and content. Software Engineering is rated as one of the best job in developed economies in terms of growth, pay, and flexibility and so on.

3.8.1Important figures in the history of software engineering

Listed below are some renowned software engineers:

Charles Bachman (born 1924) is particularly known for his work in the area of databases.
Fred Brooks (born 1931)) best-known for managing the development of OS/360.
Peter Chen, known for the development of entity-relationship modeling.
Edsger Dijkstra (1930-2002) developed the framework for proper programming.
David Parnas (born 1941) developed the concept of information hiding in modular programming.

Activity C What is the situation of software Engineering today?

4.0 Conclusion

This unit has looked at the historical development of software engineering. It has considered among other things, the pioneering era, 1945-1965: the origins, 1965-1985: thee software crisis, **1985 to 1989: No silver bullet, 1990 to 1999: Prominence of the Internet, 2000 to Present, Lightweight Methodologies,, Software engineering today and the prominent figures in the history of software engineering**

5.0 Summary

In this unit, we have learnt that:

Software engineering has historical development which can be traced from 1968 till date.

6.0 **Tutor Marked Assignment**

Discuss the historical development of software engineering

7.0 **Further Reading and Other Resources**

Pressman, Roger S (2005). ***Software Engineering**: A Practitioner's Approach* (6th ed.). Boston, Mass: McGraw-Hill. ISBN0072853182.

Sommerville, Ian (2007) [1982]. ***Software Engineering*** (8th ed.). Harlow, England: Pearson Education. ISBN 0-321-31379-8. http://www.pearsoned.co.uk/HigherEducation/Booksby/Sommerville/.

Ghezzi, Carlo (2003) [1991]. *Fundamentals of **Software Engineering*** (2nd (International) ed.). Pearson Education @ Prentice-Hall.

Unit 4 Software Engineer

1.0 Introduction

In unit 3 the historical development of software engineering was discussed. If you will recall, it traced among other things, the pioneering era, 1945-1965: the origins, 1965-1985: thee software crisis, **1985 to 1989: No silver bullet, 1990 to 1999: Prominence of the Internet, 2000 to Present, Lightweight Methodologies, Software engineering today and the prominent figures in the history of software engineering. The material in this unit will explain who a software engineer is, his tasks,** technical and functional knowledge as well as occupational characteristics. **It is expected of you that at the end of the unit, you will have achieved the objectives listed below.**

2.0 Objectives

By the end of this unit, you will be able to:

Define who a software engineer is
Explain the various tasks of a software engineer.
Explain Technical and Functional Knowledge of a Software Engineer
Explain the occupational characteristic of a software engineer.

3.0 Who is a Software Engineer?

A **software engineer** is an individual who applies the principles of software engineering to the design, development, testing, and evaluation of the software and systems in order to meet with client's requirements. He/she fully designs software, tests, debugs and maintains it. Software engineer needs knowledge of varieties of computer programming languages and applications; to enable him cope with the varieties of works before him. In view of this, he can sometimes be referred to as a computer programmer.

3.1 Functions of a Software Engineer

Analyses information to determine, recommend, and plan computer specifications and layouts, and peripheral equipment modifications.

Analyses user needs and software requirements to determine feasibility of design within time and cost constraints.

Coordinates software system installation and monitor equipment functioning to ensure specifications are met.

Designs, develops and modifies software systems, using scientific analysis and mathematical models to predict and measure outcome and consequences of design

Determines system performance standards.

Develops and direct software system testing and validation procedures, programming, and documentation.

Modifies existing software to correct errors; allow it to acclimatise to new hardware, or to improve its performance.

Obtains and evaluates information on factors such as reporting formats required, costs, and security needs to determine hardware configuration.

Stores, retrieves, and manipulates data for analysis of system capabilities and requirements.

3.8.2 Technical and Functional Knowledge and requirements of a Software Engineer

Most employers commonly recognise the technical and functional knowledge statements listed below as general occupational qualifications for Computer Software Engineers Although it is not required for the software engineer to have all of the knowledge on the list in order to be a successful performer, adequate knowledge, skills, and abilities are necessary for effective delivery of service.

The Software Engineer should have Knowledge of:

Circuit boards, processors, chips, electronic equipment, and computer hardware and software, as well as applications and programming.
Practical application of engineering science and technology. This includes applying principles, techniques, procedures, and equipment to the design and production of various goods and services.
Arithmetic, algebra, geometry, calculus, statistics, and their applications.

Structure and content of the English language including the meaning and spelling of words, rules of composition, and grammar.

Business and management principles involved in strategic planning, resource allocation, human resources modelling, leadership technique, production methods, and coordination of human and material resources.

Principles and methods for curriculum and training design, teaching and instruction for individuals and groups, and the measurement of training effects.

Design techniques, tools, and principles involved in production of precision technical plans, blueprints, drawings, and models.

Administrative and clerical procedures and systems such as word processing, managing files and records, stenography and transcription, designing forms, and other office procedures and terminology.

Principles and processes for providing customer and personal services. This includes customer needs assessment, meeting quality standards for services, and evaluation of customer satisfaction.

Transmission, broadcasting, switching, control, and operation of telecommunications systems.

3.3 Occupational features of a software Engineer

Occupations have traits or characteristics which give important clues about the nature of the work and work environment and offer you an opportunity to match your own personal interests to a specific occupation.

Software engineer occupational characteristics or features can be categorised as: **Realistic**, **Investigative** and **Conventional** as described below:

Realistic — Realistic occupations frequently involve work activities that include practical, hands-on problems and solutions. They often deal with plants, animals, and real-world materials like wood, tools, and machinery. Many of the occupations require working outside, and do not involve a lot of paperwork or working closely with others.

Investigative — Investigative occupations frequently involve working with ideas, and require an extensive amount of thinking. These occupations can involve searching for facts and figuring out problems mentally.

Activity D Discus the various tasks of software engineer.

4.0 Conclusion

This unit has explained to you who software engineer is. You have also been informed of about his various task and occupational characteristics.

5.0 Summary

In this unit, we have learnt that:

A software engineer is an individual who applies the principles of software engineering to the design, development, testing, and evaluation of the software and systems in order to meet with client’s requirements.

The tasks of a software engineer include: analysis of information, analysis of user needs and software requirements, coordination of software system installation, designs, development and modification of software systems etc.
The software engineer should have functional and technical knowledge that will assist in service delivery.
Occupational characteristics of a software engineer are categorise as : Realistic, Investigative and Conventional

6.0 Tutor Marked Assignment

1 Who is a software engineer?
2 Explain the Technical and Functional Knowledge of a Software Engineer.
3 Discuss the occupational characteristic of a software engineer.

7.0 Further Reading and Other Resources

Bureau of Labor Statistics, U.S. Department of Labor, *USDL 05-2145: Occupational Employment and Wages, November 2004*

McConnell, Steve (July10, 2003. *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers*. ISBN 978-0321193674.

UNIT 5: Software Crisis.

1.0 Introduction

In the last unit, you have learnt about the software engineer- his task, technical and functional knowledge as well as occupational characteristic. In this unit, we are going to learn about software crisis. You will learn among other things, the manifestation of software crisis, the causes of software engineering crisis and the solution to the crisis. Thus after studying this unit certain things will be required of you. They are listed in the objectives below.

2.0 Objectives

By the end of this unit, you will be able to:

- Define software crisis.
- Explain the manifestation of software crisis
- Explain the causes of software engineering crisis.
- Explain the solution of software crisis.

3.0 What is Software Crisis?

The term **software crisis** was used in the early days of software engineering. It was used to describe the impact of prompt increases in computer power and the difficulty of the problems which could be tackled. In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs. The sources of the software crisis are complexity, expectations, and change.

Conflicting requirements has always hindered software development process. For instance, while users demand a large number of features, customers generally want to minimise the amount they must pay for the software and the time required for its development.

F. L. Bauer coined the term "software crisis" at the first NATO Software Engineering Conference in 1968 at Garmisch, Germany. The term was used early in Edsger Dijkstra's 1972 ACM Turing Award Lecture:

The major cause of the software crisis is that the machines have become more powerful! This implied that: as long as there were no machines, programming was no problem at all; when there were few weak computers, programming became a mild problem, and now with huge computers, programming has equally become a huge problem.

3.1 Manifestation of Software Crisis

The crisis manifested itself in several ways:

Projects running over-budget.
Projects running over-time.
Software was very inefficient.
Software was of low quality.
Software often did not meet requirements.

Projects were unmanageable and code difficult to maintain.
Software was never delivered.

3.2 Causes of Software Engineering Crisis

The challenging practical areas include: fiscal, human resource, infrastructure, and marketing.. The very causes of failure in software development industries can be from two areas twofold: **1) Poor marketing efforts, and 2) Lack of quality products**.

3.2.1 Poor marketing efforts

The problem of poor marketing efforts is more noticeable in the developing economies, where consumers of software products prefers imported software to the detriment of locally developed ones. This problem is compounded by poor marketing approaches and the fact that most of the hardware was not manufactured locally. Though the use of software in our industries, service providing organizations, and other commercial institutions is increasing appreciably, the demand of locally developed software products is not going faster at the same rate.

One of the major reasons of this is lack of any established national policy that can speed up the creation of internal market for locally developed software products. Relatively low price of foreign (especially from the neighbouring country) software attracts the consumers in acquiring foreign products rather than buying local one.

One may wants to ask why the clients will go for local software. In this situation, the question may also be why is that the foreign software products are cheaper than the locally developed software products? The answers to these questions are not far fetched. The cost of initial take off of producing software product is significantly much higher than its subsequent versions because the latter can be produced by merely copying the initial one.

Most of the foreign software products available in the market are their succeeding versions. For this reason, the consumers in our country do not have to bear the initial cost of the development. Furthermore, this software is more reliable as they already have reputable high report. Many international commercial companies use these products efficiently.

On the contrary, most of the software firms in Bangladesh for example, need to charge the initial cost for development for their clients even though the reliability of their products is quite uncertain. Consequently, the local clients are not interested in buying local software products. To change this situation, the government must take steps by imposing high tax on foreign software products and by implementing strict copyright act for the use of software products.

International market

Apart from developing internal software market, we also need to aim at the international market. At present, as our software firms have no high report in developing software products, competing with other country will just be a fruitless effort. India for example has a high profile as far as software development is concern. India has been in global market for at least twenty years. India can take advantage of buying software from global market because of the long-time experience as well as availability of many high level IT experts at relatively low cost compared with the developed countries. Apart from these, India has professional immigrant communities in the US and in other developed countries who have succeeded in influencing the global market to procure software projects for India.

We cannot, therefore compete with India at this time to buy software projects from the global market. However, there is the need to have a policy to boost our marketing strategy to procure global software projects. One of the ways to do this is to allow country like Bangladesh through its embassies/high commissions to open up a special software marketing unit in different developed countries. Apart from this our professional expatriates living in the USA and other developed countries can also assist by setting up software firms to procure software projects to be developed in Bangladesh at low cost.

In the area of software development, timing is a essential factor. Inability to deliver the product to time can lead to loss of clients. . Our observation has shown that client cancel out work order when the software firms failed to meet up the deadline. Failure to meet up deadline for any software project may result in negative attitude to our software marketing efforts

Pricing. One of the major challenges to software developer is how to put price on the product. Most of the time, the question is "How much should our product go for. On one hand, asking too little price will be jeopardized because in that case developers will no be able to brake even. On the other hand, charging too much for the product will be a barrier to our marketing efforts. In order to solve this problem, scientific economic theories needs to be applied.

These theories must be applied when the software companies fix the prices of their product. One major lesson here is that we that are just starting in the global software market should minimise our profit margin.

3.2.2 Lack of quality products

Since most of the systems are to be used in real time environment, quality assurance is of primary concern. Presently our software companies are yet to be on ground as far as developing quality software is concerned. It will be of interest to note that presently we have over 200 software developing firms and only 20 of them have earned ISO 9001 certification and not even a single one has gotten CMM/CMM1 level 3. Even though certification is not important yardstick for quality of software product, yet ISO certification is important because it focuses on the general aspects of development to certify the quality. It must also be stated that if a software product could pass at least

level three of CMM/CMM1 then we can classify this as quality product. The hindrances to achieving quality software on part of our software industries are discussed below:

3.2.1 Lack of expertise in producing sound user requirements: Allowing the developing firms to go through some defined software development steps as suggested in software engineering discipline is a pathway to ensure the quality of software products.. The very first step is to analyze the users' requirement and designing of the system vastly depends on defining users' requirement precisely.

Ideally system analysts should do all sorts of analysis to produce user requirement analysis documents. Regrettably, in Bangladesh, a few firms do not pay much attention to producing sound user requirement documents. This reveals lack of theoretical knowledge in system analysis and design. To produce high quality requirement analysis documents there is needs for an in-depth theoretical knowledge in system analysis and design. But many of local software development firms lack the expertise in this field. In order to rectify this problem, academics in the field have to be consulted to give necessary assistance that will gear towards producing sound user requirement analysis documents.

Lack of expertise in designing the system: Aside user requirement analysis, another important aspect is the development process is the designing part of the software product. The design of any system affects the effectiveness of any implemented software. Again, one of the major problems confronting our software industries is non availability of expert software designers. It is a fact to point out that out what we have on ground are programmers or coders but the number of experienced and expert software engineers is till not many.

In fact, we rarely have resourceful persons who can guide large and complex software projects properly our software industries. The result is that there are no quality end products It may be mentioned here that sound academic knowledge in software engineering is a must for developing a quality software system. A link between industries and academic institutions can improve this situation. The utilisation of theoretical sound knowledge of academics in industrial software project cannot be overlooked. Besides depending on the complexity of the project, software firms may need to involve foreign experts for specific period to complete the project properly.

Lack of knowledge in developing model

There is need to follow some specific model in software development process. The practice in many software development firms is not to follow any particular model, and this has so much affected the quality of software product. It is mandatory for a software developer, therefore, to select a model prior to starting a software project so as to have quality product.

Absence of proper software testing procedure: For us to have quality software production the issue of software testing should be taken with utmost seriousness. demands exhaustive test to check its performance. Many theoretical testing

methodologies abound to check the performance and integrity of the software. It is rather unfortunate to note that many developing firms go ahead to , hastily deliver the end products to their clients without performing extensive test. The result of this is that many software products are not free from bugs. It should be pointed here that fixing the bugs after is costlier than during the developing time. It is therefore important for developers to perform the test phase of the development before delivering the end product to the clients.

Inconsistent documentation: Documentation is a very important aspect of software development. Most of the time, the document produced by some software firms is either incomplete or inconsistent. Since software is ever-growing product, documentation in coding must be produced and preserved for the future possible enhancement of the software.

Solution to Software Crisis

Various processes and methodologies have been developed over the last few decades to "tame" the software crisis, with varying degrees of success. However, it is widely agreed that there is no "silver bullet" — that is, no single approach which will prevent project overruns and failures in all cases. In general, software projects which are large, complicated, poorly-specified, and involve unfamiliar aspects, are still particularly vulnerable to large, unanticipated problems

Activity E What is the major cause software crisis

4.0 Conclusion

In this unit you have learnt about the crisis in software engineering- its manifestation, causes and solution.

5.0 Summary

In this unit, we have learnt that:

Software crisis refers to the difficulty of writing correct, understandable, and verifiable computer programs

.The crisis manifested itself in several ways such as: Projects running over-budget, Projects running over-time, Software was very inefficient, software was of low quality, Software often did not meet requirements, Projects were unmanageable and code difficult to maintain, Software was never delivered.

The very causes of failure in software development industries can be seen as twofold: **1) Poor marketing efforts, and 2) Lack of quality products**.

6.0 Tutor Marked Assignment

1 What is a software crisis?

2 Discus how software crisis manifested itself in the early day of software engineering.

3 Explain the causes of software crisis.

7.0 Further Reading And Other Resources

Frederick P. (1987). *No Silver Bullet: Essence and Accidents of Software Engineering*. (Reprinted in the 1995 edition of *The Mythical Man-Month*)

Disjkstra, Edsger (originally published March 1968; re-published, January 2008). "(A Look Back at) Go To Statement Considered Harmful". Association for Computing Machinery, Inc. (ACM). http://mags.acm.org/communications/200801/?pg=9. Retrieved 2008-06-12.

MODULE 2: Software Development

Unit 1: Overview of Software Development

1.0 Introduction

In the last unit, you have learnt about the software crisis- its manifestation, causes, as well as solution to the crisis. In this unit, we are going to look at the overview of software development. You will learn specifically about the overview of various stages involved in software development. After studying this unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to:

- Define clearly software development.
- List clearly the stages of software development

3.0 Definition of Software Development

Software development is the set of activities that results in software products. Software development may include research, new development, modification, reuse, re-engineering, maintenance, or any other activities that result in software products. Particularly the first phase in the software development process may involve many departments, including marketing, engineering, research and development and general management.

The term software development may also refer to computer programming, the process of writing and maintaining the source code.

3.1 Stages of Software Development

There are several different approaches to software development. While some take a more structured, engineering-based approach, others may take a more incremental approach, where software evolves as it is developed piece-by-piece. In general, methodologies share some combination of the following stages of software development:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

These stages are collectively referred to as the software development lifecycle (SDLC)., These stages may be carried out in different orders, depending on approach to software development. Time devoted on different stages may also vary. The detail of the documentation produced at each stage may not be the same.. In “waterfall” based approach, stages may be carried out in turn whereas in a more "extreme" approach, the stages may be repeated over various cycles or iterations. It is important to note that more

“extreme” approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More “extreme” approaches also encourage continuous testing throughout the development lifecycle. It ensures bug-free product at all times. The “waterfall” based approach attempts to assess the majority of risks and develops a detailed plan for the software before implementation (coding) begins. It avoids significant design changes and re-coding in later stages of the software development lifecycle.

Each methodology has its merits and demerits. The choice of an approach to solving a problem using software depends on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best choice. On the other hand, if the problem is unique (at least to the development team) and the structure of the software solution cannot be easily pictured, then a more "extreme" incremental approach may work best..

Activity F What do you think determine the choice of approach in software development?

4.0 **Conclusion**

This unit has introduce you to software development. You have been informed of the various stages of software development.

5.0 Summary

In this unit, we have learnt that:

Software development is the set of activities that results in software products. . Most methodologies share some combination of the following stages of software development: market research, gathering requirements for the proposed business solution, analyzing the problem, devising a plan or design for the software-based solution , implementation (coding) of the software, testing the software, deployment, maintenance and bug fixing

6.0 Tutor Marked Assignment

1 What is software development?

2 Briefly explain the various stages of software development.

7.0 Further Reading And Other Resources

A.M. Davis (2005). *Just enough requirements management: where software development meets marketing*.

Unit 2: Software Development Life Cycle Model

1.0 Introduction

The last unit exposed you to the overview of software development. In this unit you will learn about the various lifecycle models (the phases of the software life cycle) in general. You will also specifically learn about the requirement and the design phases

2.0 Objectives

By the end of this unit, you will be able
to: Define software life cycle
model Explain the general
model
Explain Waterfall Model
Explain V-Shaped Life Cycle Model
Explain Incremental Model
Explain Spiral Model
Discus the requirement and design phases

3.0 **Definition of Life Cycle Model**

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are a lot of models, and many companies adopt their own, but all have very similar patterns. According to Raymond Lewallen (2005), the general, basic model is shown below:

3.1 The General Model

General Life Cycle Model

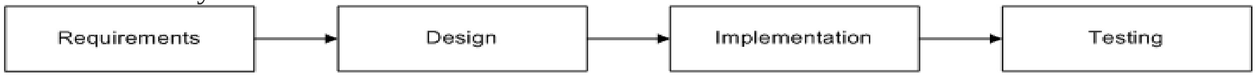


Fig 1 **the General Model**

Source: http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

Each phase produces deliverables needed by the next phase in the life cycle. Requirements are converted into design. Code is generated during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.

3.2 Waterfall Model

This is the most common life cycle models, also referred to as a linear-sequential life cycle model. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin. At the end of each phase, there is

always a review to ascertain if the project is in the right direction and whether or not to carry on or abandon the project. Unlike the general model, phases do not overlap in a waterfall model.

Waterfall Life Cycle

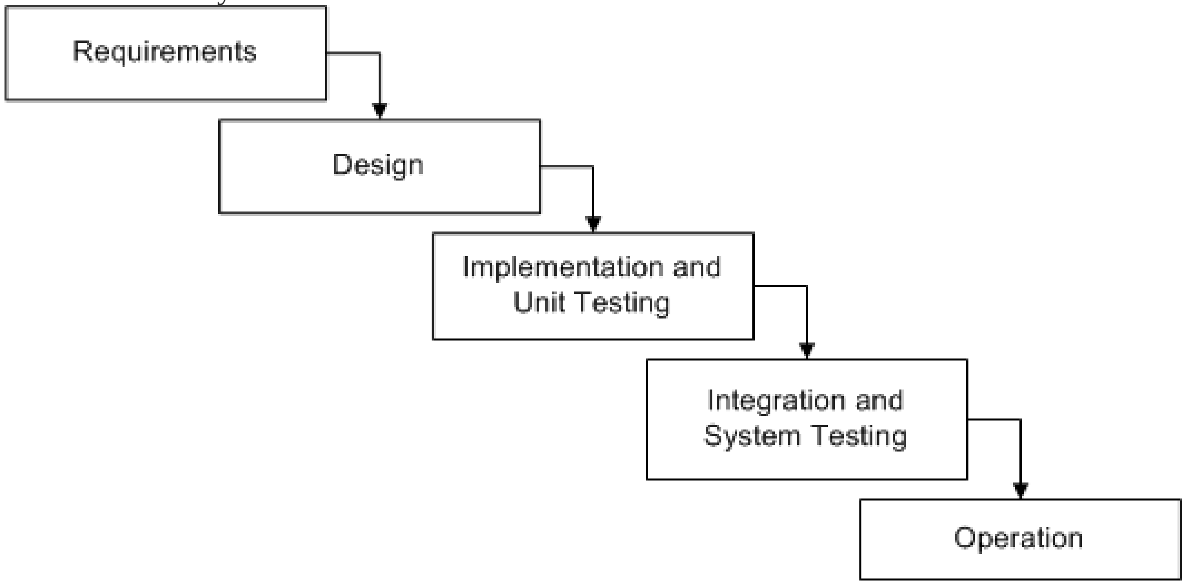

Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Fig 2 Waterfall Life Cycle

Source: http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

3.2.1 Advantages

Simple and easy to use.
Easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
Phases are processed and completed one at a time.
Works well for smaller projects where requirements are very well understood.

3.2.2 Disadvantages

Adjusting scope during the life cycle can kill a project
No working software is produced until late during the life cycle.
High amounts of risk and uncertainty.
Poor model for complex and object-oriented projects.
Poor model for long and ongoing projects.

Poor model where requirements are at a moderate to high risk of changing.

3.3 V-Shaped Model

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well.

The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.

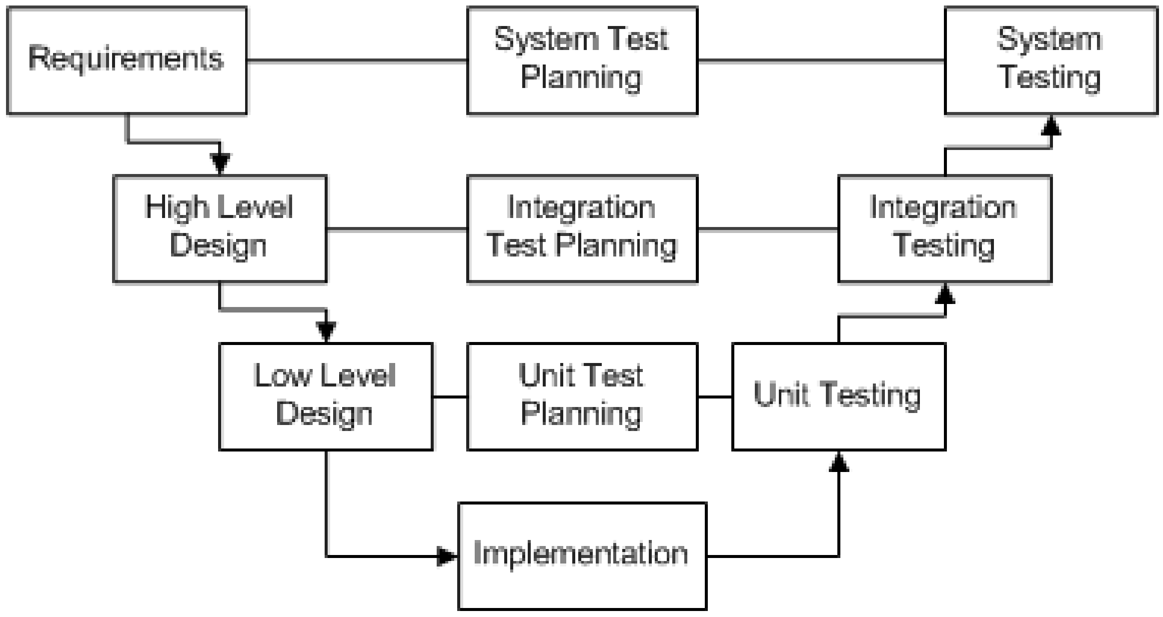


Fig 3 V-Shaped Life Cycle Model

Source: http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

3.3.1 Advantages

Simple and easy to use.
Each phase has specific deliverables.
Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
Works well for small projects where requirements are easily understood.

3.3.2 Disadvantages

Very rigid, like the waterfall model.
Little flexibility and adjusting scope is difficult and expensive.
Software is developed during the implementation phase, so no early prototypes of the software are produced.
Model doesn't provide a clear path for problems discovered during testing phases.

3.4 Incremental Model

The incremental model is an intuitive approach to the waterfall model. It is a kind of a "multi-waterfall" cycle. In that multiple development cycles take at this point. Cycles are broken into smaller, more easily managed iterations. Each of the iterations goes through the requirements, design, implementation and testing phases.

The first iteration produces a working version of software and this makes possible to have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

Incremental Life Cycle Model

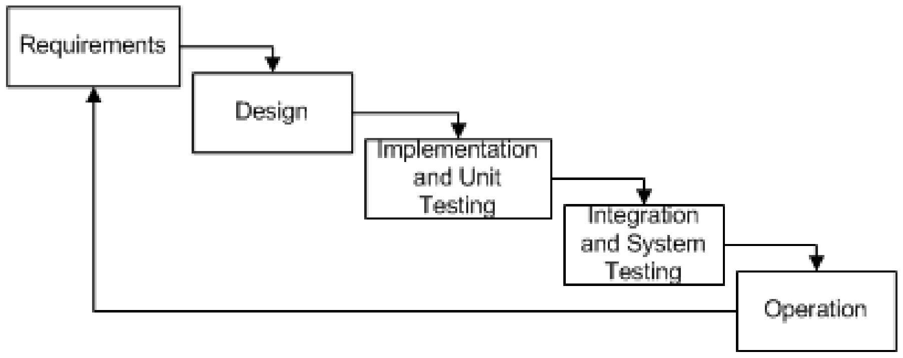

Requirements
Design
Implementation and Unit Testing
Integration and System Testing
Operation

Fig 4 Incremental Life Cycle Model

Source: http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

3.4.1 Advantages

Generates working software quickly and early during the software life cycle.

More flexible – inexpensive to change scope and requirements.
Easier to test and debug during a smaller iteration.
Easier to manage risk because risky pieces are identified and handled during its iteration.
Each of the iterations is an easily managed landmark

3.4.2 Disadvantages

Each phase of an iteration is rigid and do not overlap each other.
Problems as regard to system architecture may arise as a result of inability to gathered requirements up front for the entire software life cycle.

3.5 Spiral Model

The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases namely Planning, Risk Analysis, Engineering and Evaluation. A software project continually goes through these phases in iterations which are called spirals. In the baseline spiral requirements are gathered and risk is assessed. Each subsequent spiral builds on the baseline spiral.

Requirements are gathered during the planning phase. In the risk analysis phase, a process is carried out to discover risk and alternate solutions. A prototype is produced at the end of the risk analysis phase.

Software is produced in the engineering phase, alongside with testing at the end of the phase. The evaluation phase provides the customer with opportunity to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the angular component denotes progress, and the radius of the spiral denotes cost.

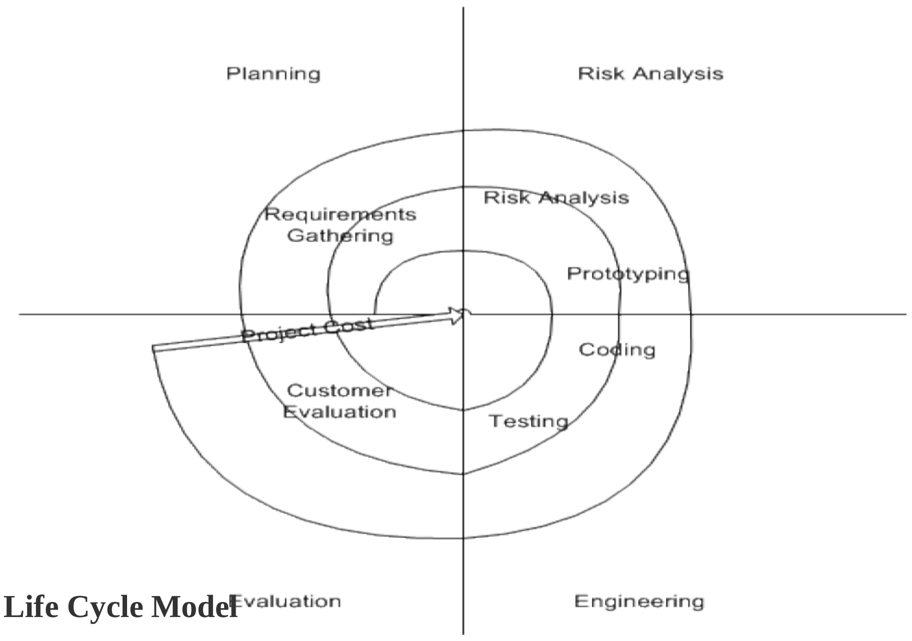


Fig 5 Spiral Life Cycle Model

Source: http://codebetter.com/blogs/raymond.lewallen/archive/2005/07/13/129114.aspx.

3.5.1 Merits

High amount of risk analysis
Good for large and mission-critical projects.
Software is produced early in the software life cycle.

3.5.2 Demerits

Can be a costly model to use.
Risk analysis requires highly specific expertise.
Project’s success is highly dependent on the risk analysis phase.
Doesn’t work well for smaller projects.

3.6 Requirements Phase

Business requirements are gathered in this phase. This phase is the main center of attention of the project managers and stake holders. Meetings with managers, stake holders and users are held in order to determine the requirements. Th general questions that require answers during a requirements gathering phase are: Who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system? A list of functionality that the system should provide, which describes functions the system should perform, business logic that processes data, what data is stored and used by the system, and how the user interface should work is produced at this point. The requirements development phase may have been preceded by a feasibility study, or a conceptual analysis phase of the project. The requirements phase may be divided into requirements elicitation (gathering the requirements from stakeholders), analysis (checking for consistency and completeness), specification (documenting the requirements) and validation (making sure the specified requirements are correct)

In systems engineering, a **requirement** can be a description of *what* a system must do, referred to as a *Functional Requirement*. This type of requirement specifies something that the delivered system must be able to do. Another type of requirement specifies something about the system itself, and how well it performs its functions. Such requirements are often called *Non-functional requirements*, or 'performance requirements' or 'quality of service requirements.' Examples of such requirements include usability, availability, reliability, supportability, testability, maintainability, and (if defined in a way that's verifiably measurable and unambiguous) ease-of-use.

3.6.1 Types of Requirements

Requirements are categorised as:

Functional requirements which describe the functionality that the system is to execute; for example, formatting some text or modulating a signal.
Non-functional requirements which are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as quality requirements or Constraint requirements No matter how the problem is solved the constraint requirements must be adhered to.

It is important to note that functional requirements can be directly implemented in software. The non-functional requirements are controlled by other aspects of the system. For example, in a computer system reliability is related to hardware failure rates, performance controlled by CPU and memory. Non-functional requirements can in some cases be broken into functional requirements for software. For example, a system level non-functional safety requirement can be decomposed into one or more functional requirements. In addition, a non-functional requirement may be converted into a process requirement when the requirement is not easily measurable. For example, a system level maintainability requirement may be decomposed into restrictions on software constructs or limits on lines or code.

3.6.2 Requirements analysis

Requirements analysis in systems engineering and software engineering, consist of those activities that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

3.6.3 The Need for Requirements Analysis

Studies reveal that insufficient attention to Software Requirements Analysis at the beginning of a project is the major reason for critically weak projects that often do not fulfil basic tasks for which they were designed. Software companies are now spending time and resources on effective and streamlined Software Requirements Analysis Processes as a condition to successful projects that support the customer's business goals and meet the project's requirement specifications.

3.6.4 Requirements Analysis Process: Requirements Elicitation, Analysis And Specification

Requirements Analysis is the process of understanding the client needs and expectations from a proposed system or application. It is a well-defined stage in the Software Development Life Cycle model.

Requirements are a description of how a system should behave, in other words, a description of system properties or attributes. Considering the numerous levels of dealings between users, business processes and devices in worldwide corporations today, there are immediate and composite requirements from a single application, from different levels within an organization and outside it

The Software Requirements Analysis Process involves the complex task of eliciting and documenting the requirements of all customers, modelling and analyzing these requirements and documenting them as a foundation for system design.

This job (requirements analysis process) is dedicated to a specialized Requirements Analyst. The Requirements Analysis function may also come under the scope of Project Manager, Program Manager or Business Analyst, depending on the organizational hierarchy.

3.6.5 Steps in the Requirements Analysis Process

3.6.5.1 Fix system boundaries

This is initial step and helps in identifying how the new application fit in into the business processes, how it fits into the larger picture as well as its capacity and limitations.

3.6.5.2 Identify the customer

This focuses on identifying who the ‘users’ or ‘customers’ of an application are that is to say knowing the group or groups of people who will be directly or indirectly impacted by the new application. This allows the Requirements Analyst to know in advance where he has to look for answers.

3.6.5.3 Requirements elicitation

Here information is gathered from the multiple stakeholders identified. The Requirements Analyst brings out from each of these groups what their requirements from the application are and what they expect the application to achieve. Taking into account the multiple stakeholders involved, the list of requirements gathered in this manner could go into pages. The level of detail of the requirements list depends on the number and size of user groups, the degree of complexity of business processes and the size of the application.

3.6.5.3.1 Problems faced in Requirements Elicitation

- Ambiguous understanding of processes
- Inconsistency within a single process by multiple users
- Insufficient input from stakeholders
- Conflicting stakeholder interests
- Changes in requirements after project has begun

3.6.5.3.2 Tools used in Requirements Elicitation

Tools used in Requirements Elicitation include stakeholder interviews and focus group studies. Other methods like flowcharting of business processes and the use of existing documentation like user manuals, organizational charts, process models and systems or process specifications, on-site analysis, interviews with end-users, market research and competitor analysis are also used widely in Requirements Elicitation.

There are of course, modern tools that are better equipped to handle the complex and multilayered process of Requirements Elicitation. Some of the current Requirements Elicitation tools in use are:

- Prototypes
- Use cases
- Data flow diagrams
- Transition process diagrams

User interfaces

3.6.5.4 Requirements Analysis

The moment all stakeholder requirements have been gathered, a structured analysis of these can be done after modeling the requirements. Some of the Software Requirements Analysis techniques used are requirements animation, automated reasoning, knowledge-based critiquing, consistency checking, analogical and case-based reasoning.

3.6.5.5. Requirements Specification

After requirements have been elicited, modeled and analyzed, they should be documented in clear, definite terms. A written requirements document is crucial and as such its circulation should be among all stakeholders including the client, user-groups, the development and testing teams. It has been observed that a well-designed, clearly documented Requirements Specification is vital and serves as a:

Base for validating the stated requirements and resolving stakeholder conflicts, if any

Contract between the client and development team

Basis for systems design for the development team

Bench-mark for project managers for planning project development lifecycle and goals

Source for formulating test plans for QA and testing teams

Resource for requirements management and requirements tracing

Basis for evolving requirements over the project life span

Software requirements specification involves scoping the requirements so that it meets the customer's vision. It is the result of teamwork between the end-user who is usually not a technical expert, and a Technical/Systems Analyst, who is expected to approach the situation in technical terms.

The software requirements specification is a document that lists out stakeholders' needs and communicates these to the technical community that will design and build the system. It is really a challenge to communicate a well-written requirements specification, to both these groups and all the sub-groups within. To overcome this, Requirements Specifications may be documented separately as:

User Requirements - written in clear, precise language with plain text and use cases, for the benefit of the customer and end-user

System Requirements - expressed as a programming or mathematical model, meant to address the Application Development Team and QA and Testing Team.

Requirements Specification serves as a starting point for software, hardware and database design. It describes the function (Functional and Non-Functional specifications) of the system, performance of the system and the operational and user-interface constraints that will govern system development.

3.7 Requirements Management

Requirements Management is the all-inclusive process that includes all aspects of software requirements analysis and as well ensures verification, validation and traceability of requirements. Effective requirements management practices assure that all system requirements are stated unmistakably, that omissions and errors are corrected and that evolving specifications can be included later in the project lifecycle.

3.7 Design Phase

The software system design is formed from the results of the requirements phase. This is where the details on how the system will work are produced. Deliverables in this phase include hardware and software, communication, software design.

3.8 Definition of software design

A software design is a meaningful engineering representation of some software product that is to be built. A design can be traced to the customer's requirements and can be assessed for quality against predefined criteria. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces and components.

The design process is very important. As a labourer, for example one would not attempt to build a house without an approved blueprint so as not to risk the structural integrity and customer satisfaction. In the same way, the approach to building software products is no unlike. The emphasis in design is on quality. It is pertinent to note that, this is the only phase in which the customer’s requirements can be precisely translated into a finished software product or system. As such, software design serves as the foundation for all software engineering steps that follow regardless of which process model is being employed.

During the design process the software specifications are changed into design models that express the details of the data structures, system architecture, interface, and components. Each design product is re-examined for quality before moving to the next phase of software development. At the end of the design process a design specification document is produced. This document is composed of the design models that describe the data, architecture, interfaces and components.

3.9 Design Specification Models

Data design – created by changing the analysis information model (data dictionary and ERD) into data structures needed to implement the software. Part of the data design may occur in combination with the design of software architecture. More detailed data design occurs as each software component is designed.

Architectural design - defines the relationships among the major structural elements of the software, the "design patterns" that can be used to attain the requirements that have been defined for the system, and the constraint that affect the way in which the architectural patterns can be applied. It is derived from the system specification, the analysis model, and the subsystem interactions defined in the analysis model (DFD).

Interface design - explains how the software elements communicate with each other, with other systems, and with human users. Much of the necessary information required is provided by the e data flow and control flow diagrams.

Component-level design – It converts the structural elements defined by the software architecture into procedural descriptions of software components using information acquired from the process specification (PSPEC), control specification (CSPEC), and state transition diagram (STD).

3.10 Design Guidelines

In order to assess the quality of a design (representation) the yardstick for a good design should be established. Such a design should:

exhibit good architectural structure

be modular

contain distinct representations of data, architecture, interfaces, and components (modules)

lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable design patterns

lead to components that exhibit independent functional characteristics

lead to interfaces that reduce the complexity of connections between modules and with the external environment

be derived using a reputable method that is driven by information obtained during software requirements analysis

These criteria are not acquired by chance. The software design process promotes good design through the application of fundamental design principles, systematic methodology and through review.

3.11 Design Principles

Software design can be seen as both a process and a model.

"The design process is a series of steps that allow the designer to describe all aspects of the software to be built. However, it is not merely a recipe book; for a competent and successful design, the designer must use creative skill, past experience, a sense of what makes "good" software, and have a commitment to quality.

The set of principles which has been established to help the software engineer in directing the design process are:

The design process should not suffer from tunnel vision – Alternative approaches should be considered by a good designer. Designer should judge each approach based on the requirements of the problem, the resources available to do the job and any other constraints.

The design should be traceable to the analysis model – because a single element of the design model often traces to multiple requirements, it is necessary to have a means of tracking how the requirements have been satisfied by the model

The design should not reinvent the wheel – Systems are constructed using a suite of design patterns, many of which may have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Design time should be spent in expressing truly fresh ideas and incorporating those patterns that already exist.

The design should reduce intellectual distance between the software and the problem as it exists in the real world – This means that, the structure of the software design should imitate the structure of the problem domain.

The design should show uniformity and integration – a design is uniform if it appears that one person developed the whole thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

The design should be structured to degrade gently, even with bad data, events, or operating conditions are encountered – Well-designed software should never "bomb". It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

The design should be reviewed to minimize conceptual (semantic) errors – there is sometimes the tendency to focus on minute details when the design is reviewed, missing the forest for the trees. The designer team should ensure that major conceptual elements of the design have been addressed before worrying about the syntax if the design model.

Design is not coding, coding is not design – Even when detailed designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made of the coding level address the small implementation details that enable the procedural design to be coded.

The design should be structured to accommodate change

The design should be assessed for quality as it is being created

With proper application of design principles, the design displays both external and internal quality factors. External quality factors are those factors that can readily be observed by the user, (e.g. speed, reliability, correctness, usability). Internal quality factors have to do with technical quality more so the quality of the design itself. To achieve internal quality factors the designer must understand basic design concepts.

3.12 Fundamental Software Design Concepts

Over the past four decades, a set of fundamental software design concepts has evolved, each providing the software designer with a foundation from which more sophisticated design methods can be applied. Each concept assists the soft ware engineer to answer the following questions:

What criteria can be used to partition software into individual components?

How is function or data structure detail separated from a conceptual representation of software?

Are there uniform criteria that define the technical quality of a software design?

The fundamental design concepts are:

Abstraction - allows designers to focus on solving a problem without being concerned about irrelevant lower level details (procedural abstraction - named sequence of events, data abstraction - named collection of data objects)

Refinement - process of elaboration where the designer provides successively more detail for each design component

Modularity - the degree to which software can be understood by examining its components independently of one another

Software architecture - overall structure of the software components and the ways in which that structure provides conceptual integrity for a system

Control hierarchy or **program structure** - represents the module organization and implies a control hierarchy, but does not represent the procedural aspects of the software (e.g. event sequences)

Structural partitioning - horizontal partitioning defines three partitions (input, data transformations, and output); vertical partitioning (factoring) distributes control in a top-down manner (control decisions in top level modules and processing work in the lower level modules).

Data structure - representation of the logical relationship among individual data elements (requires at least as much attention as algorithm design)

Software procedure - precise specification of processing (event sequences, decision points, repetitive operations, data organization/structure)

Information hiding - information (data and procedure) contained within a module is inaccessible to modules that have no need for such information

Activity G 1 What are the steps in requirement Analysis process?

2 What are the fundamental design concepts ?

4.0 **Conclusion**

Software life cycle models describe phases of the software cycle and the order in which those phases are executed.

5.0 Summary

In this unit, we have learnt that:

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. .
In general model, each phase produces deliverables required by the next phase in the life cycle. Requirements are translated into design. Code is produced during implementation that is driven by the design. Testing verifies the deliverable of the implementation phase against requirements.
In a waterfall model, each phase must be completed in its entirety before the next phase can begin. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. Unlike what I mentioned in the general model, phases do not overlap in a waterfall model.
Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.
The incremental model is an intuitive approach to the waterfall model. Multiple development cycles take place here, making the life cycle a "multi-waterfall" cycle. Cycles are divided up into smaller, more easily managed iterations. Each iteration passes through the requirements, design, implementation and testing phases.
The spiral model is similar to the incremental model, with more emphases placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spirals, starting in the planning phase, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral.
In requirement phase business requirements are gathered and that the phase is the main focus of the project managers and stake holders.
The software system design is produced from the results of the requirements phase and it is the phase is where the details on how the system will work is produced

6.0 Tutor Marked Assignment

1 What is software life cycle model?
2 Explain the general model
3 Compare and contrast General and Waterfall Models
4 Explain V-Shaped Life Cycle Model
5 Explain Incremental Model
6 Compare and contrast Incremental and Spiral Models
7 Discus the requirement and design phases

7.0 Further Reading And Other Resources

Blanchard, B. S., & Fabrycky, W. J.(2006) Systems engineering and analysis (4th ed.) New Jersey: Prentice Hall.

Ummings, Haag (2006). Management Information Systems for the Information Age. Toronto, McGraw-Hill Ryerson

Unit 3 Modularity

1.0 Introduction

In unit 2 we discussed about software lifecycle models in general and also in detailed the requirement and the design phases of software development. In this unit we will look at Modudularity in programming.

2.0 Objectives

By the end of this unit, you will be able to:

Define Modularity
Differentiate between logical and physical modularity
Explain benefits of modular design
Explain approaches of writing modular program
Explain Criteria for using modular design
Outlines the attributes of a good module
Outline the steps to creating effective module
Differentiate between Top-down and Bottom-up programming approach

What is Modularity?

Modularity is a general systems concept which is the degree to which a system's components may be separated and recombined. It refers to both the tightness of coupling between components, and the degree to which the "rules" of the system architecture enable (or prohibit) the mixing and matching of components

The concept of modularity in computer software has been promoted for about five decades. In essence, the software is divided into separately names and addressable components called modules that are integrated to satisfy problem requirements. It is important to note that a reader cannot easily understand large programs with a single module. The number of variables, control paths and sheer complexity make understanding almost impossible. As a result a modular approach will allow for the software to be intellectually manageable. However, it is important to note that software cannot be subdivided indefinitely so as to make the effort required to understand or develop it negligible. This is because the more the number of modules, the less the effort to develop them.

3.14 **Logical Modularity**

Generally in software, **modularity can be categorized as logical or physical**. **Logical Modularity** is concerned with the **internal organization of code into logically-related**

units. In modern high level languages, logical modularity usually starts with the class, the smallest code group that can be defined. In languages such as Java and C#, classes can be further combined into packages which allow developers to organize code into group of related classes. Depending on the environment, **a module can be** implemented as a single **class**, several **classes** in a package, or an entire **API** (a collection of packages). You should be able to **describe the functionality of tour module in a single sentence** (i.e. this module calculates tax per zip code) regardless of the implementation scale of your module,). Your module should expose its functionality as simple interfaces that shield callers from all implementation details. The functionality of a module should be accessible through a published interface that allows the module to expose its functionalities to the outside world while hiding its implementation details.

3.15 Physical Modularity

Physical Modularity is probably the earliest form of modularity introduced in software creation. Physical modularity consists of two main components namely: (1) **a file that contains compiled code** and other resources and (2) **an executing environment** that understand how to execute the file. Developers build and assemble their modules into compiled assets that **can be distributed** as single or multiple files. In Java for example, the jar file is the unit of physical modularity for code distribution (.Net has the assembly). The file and its associated meta-data are designed to be loaded and executed by the run time environment that understands how to run the compiled code. Physical modularity can also be affected by the context and scale of abstraction. Within Java, for instance, the developer community has created and accept s**everal physical modularity strategies** to address different aspects of enterprise development 1) **WAR** for web components 2) **EJB** for distributed enterprise components 3) **EAR** for enterprise application components 4) vendor specific modules such as **JBoss Service Archive** (SAR). These are usually **a variation of the JAR file format** with special meta data to target the intended runtime environment. The **current trend of adoption** seems to be pointing to **OSGi** as a **generic physical module format**. **OSGi** provides the Java environment with **additional functionalties** that should allow developers to model their modules **to scale from small emddeable to complex enterprise components** (a lofty goal in deed).

3.16 Benefits of Modular Design

Scalable Development: a modular design allows a **project to be naturally subdivided along the lines of its modules**. A developer (or groups of developers) can be assigned a module to implement independently which can produce an asynchronous project flow.
Testable Code Unit: when your code is partition into functionally-related chunks, it **facilitates the testing of each module** independently. With the proper testing framework, developers can exercise each module (and its constituencies) without having to bring up the entire project.
Build Robust System: in the monolithic software design, as your system grows in complexity so does its propensity to be brittle (changes in one section causes failure in another). Modularity lets you **build complex system composed of smaller parts** that can be **independently managed and maintained**. Fixes in one portion of the code does not necessarily affect the entire system.
Easier Modification & Maintenance: post-production system maintenance is another crucial benefit of modular design. Developers have the **ability to fix and make non-infrastructural changes** to module **without affecting other modules**. The updated module can independently go through the build and release cycle without the need to re-build and redeploy the entire system.

Functionally Scalable: depending on the level of sophistication of your modular design, it's possible to **introduce new functionalities with little or no change to existing modules**. This allows your software system to scale in functionality without becoming brittle and a burden on developers.

3.17 Approaches of writing Modular program

The three basic approaches of designing Modular program are:

Process-oriented design

This approach places the emphasis on the process with the objective being to design modules that have high cohesion and low coupling. (Data flow analysis and data flow diagrams are often used.)

Data-oriented design

In this approach the data comes first. That is the structure of the data is determined first and then procedures are designed in a way to fit to the structure of the data.

Object-oriented design

In this approach, the objective is to first identify the objects and then build the product around them. In concentrate, this technique is both data- and process-oriented.

3.18 Criteria for using Modular Design

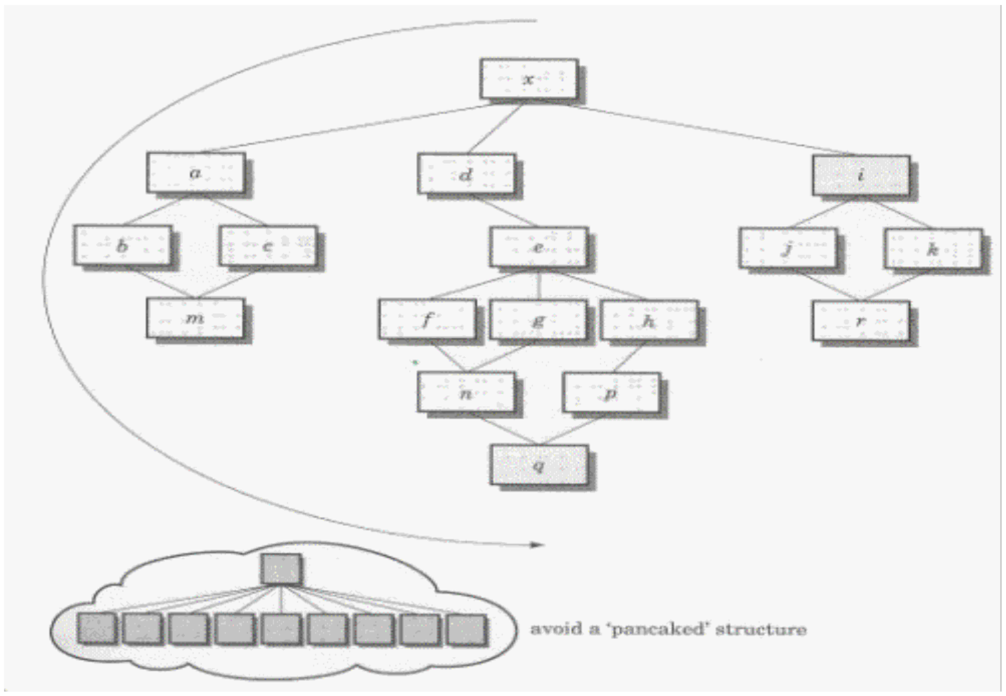
Modular decomposability – If the design method provides a systematic means for breaking problem into sub problems, it will reduce the complexity of the overall problem, thereby achieving a modular solution.

Modular compos ability - If the design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understand ability – If a module can be understood as a stand-alone unit (without reference to other modules) it will be easier to build and easier to change.

Modular continuity – If small changes to the system requirements result in changes to individual modules, rather than system-wide changes, the impact of change-induced side-effects will be minimised

Modular protection – If an abnormal condition occurs within a module and its effects are constrained within that module, then impact of error-induced side-effects are minimised



3.19 Attributes of a good Module

Functional independence - modules have high cohesion and low coupling

Cohesion - qualitative indication of the degree to which a module focuses on just one thing

Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world

3.20 Steps to Creating Effective Module

Evaluate the first iteration of the program structure to reduce coupling and improve cohesion. Once program structure has been developed modules may be exploded or imploded with aim of improving module independence.

- o An exploded module becomes two or more modules in the final program structure.
- o An imploded module is the result of combining the processing implied by two or more modules.

An exploded module normally results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data and interface complexity.

Attempt to minimise structures with high fan-out; strive for fan-in as structure depth increases. The structure shown inside the cloud in Fig. 3 does not make effective use of factoring.

Fig 6 Example of a program structure

Keep the scope of effect of a module within the scope of control for that module.

- o The scope of effect of a module is defined as all other modules that are affected by a decision made by that module. For example, the scope of control of module e is all modules that are subordinate i.e. modules f, g, h, n, p and q.

Evaluate module interfaces to reduce complexity, reduce redundancy, and improve consistency.

- o Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e. seemingly unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be re-evaluated.

Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single task).

- o A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details. Modules that have internal "memory" can be unpredictable unless care is taken in their use.
- o A module that restricts processing to a single task exhibits high cohesion and is viewed favourably by a designer.

Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

- o This warns against content coupling. Software is easier to understand and maintain if the module interfaces are constrained and controlled.

3.21 Programming Languages that formally support module concept

Languages that formally support the module concept include IBM/360 Assembler, COBOL, RPG and PL/1, Ada, D, F, Fortran, Haskell, OCaml, Pascal, ML, Modula-2, Erlang, Perl, Python and Ruby. The IBM System i also uses Modules in RPG, COBOL and CL, when programming in the ILE environment. Modular programming can be performed even where the programming language lacks explicit syntactic features to support named modules.

Software tools can create modular code units from groups of components. Libraries of components built from separately compiled modules can be combined into a whole by using a linker.

3.22 Module Interconnection Languages

Module interconnection languages (**MILs**) provide formal grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. MILs enable the separation between programming-in-the-small and programming-in-the-large. Coding a module represents programming in the small, while assembling a system with the help of a MIL represents programming in the large. An example of MIL is MIL-75.

3.23 Top-Down Design

Top-down is a programming style, the core of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces. Finally, the components are precise enough to be coded and the program is written. It is the exact opposite of the bottom-up programming approach which is common in object-oriented languages such as C++ or Java.

The method of writing a program using top-down approach is to write a main procedure that names all the major functions it will need. After that the programming team examines the requirements of each of those functions and repeats the process. These compartmentalized sub-routines finally will perform actions so straightforward they can be easily and concisely coded. The program is done when all the various sub-routines have been coded.

Merits of top-down programming:

Separating the low level work from the higher level abstractions leads to a modular design.
Modular design means development can be self contained.
Having "skeleton" code illustrates clearly how low level modules integrate.
Fewer operations errors
Much less time consuming (each programmer is only concerned in a part of the big project).
Very optimized way of processing (each programmer has to apply their own knowledge and experience to their parts (modules), so the project will become an optimized one).
Easy to maintain (if an error occurs in the output, it is easy to identify the errors generated from which module of the entire program).

3.24 Bottom-up approach

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then connected together to form bigger subsystems, which 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.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.

. This bottom-up approach has one drawback. We need to use a lot of perception to decide the functionality that is to be provided by the module. This approach is more suitable if a system is to be developed from existing system, because it starts from some existing modules. Modern software design approaches usually mix both top-down and bottom-up approaches.

Activity H What are the steps to create effective modules?

4.0 Conclusion

The benefits of modular programming cannot be overemphasised. It among other things, allows for scalar development, it facilitates code testing, helps in building robust system, allows for easier modification and maintenance.

5.0 Summary

In this unit, we have learnt that:

Modularity is a general systems concept, the degree to which a system's components may be separated and recombined. It refers to both the tightness of coupling between components, and the degree to which the "rules" of the system architecture enable (or prohibit) the mixing and matching of components

Physical Modularity is probably the earliest form of modularity introduced in software creation. Physical modularity consists of two main components namely: (1) **a file that contains compiled code** and other resources and (2) **an executing environment** that understand how to execute the file. Developers build and assemble their modules into compiled assets that **can be distributed** as single or multiple files.

Logical Modularity is concerned with the **internal organization of code into logically-related units**.

Modular programming is beneficial in that:It allows for scalar development, it facilitates code testing, helps in building robust system, allows for easier modification and maintenance.

The three basic approaches of designing Modular program are: Process-oriented design, Data-oriented design and Object-oriented design.

Criteria for using Modular Design include: Modular decomposability, Modular compos ability, Modular understand ability, Modular continuity, and Modular protection.

Attributes of a good Module include: Functional independence, Cohesion, and Coupling

Steps to Creating Effective Module include: Evaluate the first iteration of the program structure to reduce coupling and improve cohesion, Attempt to minimise structures with high fan-out; strive for fan-in as structure depth increases, Define modules whose function is predictable and not overly restrictive (e.g. a module that only implements a single task), Strive for controlled entry modules, avoid pathological connection (e.g. branches into the middle of another module)

Top-down is a programming style, the core of traditional procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces. Finally, the components are precise enough to be coded and the program is written.

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then connected together to form bigger subsystems, which are linked, sometimes in many levels, until a complete top-level system is formed

6.0 Tutor Marked Assignment

What is Modularity?
Differentiate between logical and physical modularity
What are the benefits of modular design
Explain the approaches of writing modular program

What are the Criteria for using modular design
Outlines the attributes of a good module
Outline the steps to creating effective module
Differentiate between Top-down and Bottom-up programming approach

7.0 Futher Reading And Other Resouces

Laplante, Phil (2009). *Requirements Engineering for Software and Systems* (1st ed.). Redmond, WA: CRC Press. ISBN 1-42006-467-3. http://beta.crcpress.com/product/isbn/9781420064674.

McConnell, Steve (1996). *Rapid Development: Taming Wild Software Schedules* (1st ed.). Redmond, WA: Microsoft Press. ISBN 1-55615-900-5. http://www.stevemcconnell.com/.

Wiegers, Karl E. (2003). *Software Requirements 2: Practical techniques for gathering and managing requirements throughout the product development cycle* (2nd ed.). Redmond: Microsoft Press. ISBN 0-7356-1879-8.

Andrew Stellman and Jennifer Greene (2005). *Applied Software Project Management*. Cambridge, MA: O'Reilly Media. ISBN 0-596-00948-8. http://www.stellman-greene.com.

Unit 4 Pseudo code

1.0 Introduction

In the last unit, you have learnt about **Modudularity in programming. Its** benefits, design approaches and criteria, attributes of a good Module and the steps to creating effective module. You equally learnt about Top-Down and Bottom-up approaches in programming. This unit ushers you into Pseudo code a way to create a logical structure that will describing the actions, which will be executed by the application. After studying this unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to:

Define Pseudo code

Explain General guidelines for writing Pseudo code.

Give examples of Pseudo codes

3.26 Definition of Pseudo code

Pseudo-code is a non-formal language, a way to create a logical structure, describing the actions, which will be executed by the application. Using pseudo-code, the developer shows the application logic using his local language, without applying the structural rules of a specific programming language. The big advantage of the pseudo-code is that the application logic can be easily comprehended by any developer in the development team. In addition, when the application algorithm is expressed in pseudo-code, it is very easy to convert the pseudo-code into real code (using any programming language).

3.26 General guidelines for writing Pseudo code

Here are a few general guidelines for writing your pseudo code:

Mimic good code and good English. Using aspects of both systems means adhering to the style rules of both to
some degree. It is still important that variable names be mnemonic, comments be included where useful, and English
phrases be comprehensible (full sentences are usually not necessary).

Ignore unnecessary details. If you are worrying about the placement of commas, you are using too much detail. It is a
good idea to use some convention to group statements (begin/end, brackets, or whatever else is clear), but you shouldn't
obsess about syntax.

Don't belabor the obvious. In many cases, the type of a variable is clear from context; unless it is critical that it is specified to be an integer or real, it is often unnecessary to make it explicit.

Take advantage of programming shorthands. Using if-then-else or looping structures is more concise than writing
out the equivalent in English; general constructs that are not peculiar to a small number of languages are good candidates
for use in pseudocode. Using parameters in specifying procedures is concise, clear, and accurate, and hence should not
be omitted from pseudocode.

Consider the context. If you are writing an algorithm for quicksort, the statement *use quicksort to sort the values* is
hiding too much detail; if you have already studied quicksort in a class and later use it as a subroutine in another

algorithm, the statement would be appropriate to use.

Don't lose sight of the underlying model. It should be possible to see through" your pseudocode to the model below;
if not (that is, you are not able to analyze the algorithm easily), it is written at too high a level.

Check for balance. If the pseudocode is hard for a person to read or difficult to translate into working code (or worse
yet, both!), then something is wrong with the level of detail you have chosen to use.

3.27 Examples of Pseudocode

Example 1 - Computing Sales Value Added (VAT) Tax : Pseudo-code the task of computing the final price of an item after figuring in sales tax. Note the three types of instructions: input (get), process/calculate (=) and output (display)

```
1.      get price of item
2.      get VAT rate
3.      VAT = price of time times VAT rate
4       final price = price of item plus VAT
5.      display final price
6.      stop
```

Variables: price of item, sales tax rate, sales tax, final price

Note that the operations are numbered and each operation is unambiguous and effectively computable. We also extract and list all variables used in our pseudo-code. This will be useful when translating pseudo-code into a programming language

Example 2 - Computing Weekly Wages: Gross pay depends on the pay rate and the number of hours worked per week. However, if you work more than 50 hours, you get paid time-and-a-half for all hours worked over 50. Pseudo-code the task of computing gross pay given pay rate and hours worked.

```
1.      get hours worked
2.      get pay rate
3.      if hours worked ≤ 50 then
3.1     gross pay = pay rate times hours worked
4.      else
4.1     gross pay = pay rate times 50 plus 1.5 times pay rate times (hours
        worked minus 50)
5.      display gross pay
```

```
6.      halt

        variables:  hours worked, ray rate, gross pay
```

This example presents the *conditional* control structure. On the basis of the true/false question asked in line 3, line 3.1 is executed if the answer is True; otherwise if the answer is False the lines subordinate to line 4 (i.e. line 4.1) is executed. In both cases pseudo-code is resumed at line 5.

Example 3 - Computing a Question Average: Pseudo-code a routine to calculate your question average.

```
1.      get number of questions
2.      sum = 0
3.      count = 0
4.      while count < number of questions
        4.1     get question grade
        4.2     sum = sum + question grade
        4.3     count = count + 1
5.      average = sum / number of question
6.      display average
7.      stop

        variables: number of question, sum ,count, question grade, average
```

This example presents an *iterative* control statement. As long as the condition in line 4 is True, we execute the subordinate operations 4.1 - 4.3. When the condition is False, we return to the pseudo-code at line 5.

This is an example of a *top-test* or *while do* iterative control structure. There is also a *bottom-test* or *repeat until* iterative control structure which executes a block of statements until the condition tested at the end of the block is False.

Some Keywords That Should be Used

For looping and selection, The keywords that are to be used include Do While...EndDo; Do Until...Enddo; Case...EndCase; If...Endif; Call ... with (parameters); Call; Return ....; Return; When; Always use scope terminators for loops and iteration.

As verbs, use the words Generate, Compute, Process, etc. Words such as set, reset, increment, compute, calculate, add, sum, multiply, ... print, display, input, output, edit, test , etc. with careful indentation tend to foster desirable pseudocode.

Do not include data declarations in your pseudo code.

Activity I Write a pseudo code to find the average of even number between 1 and 20

4.0 Conclusion

The role of pseudo-code in program design cannot be underestimated. When it used, it is not only that logic of application can easily be understood but it can easily be converted into real code.

5.0 Summary

In this unit, you have learnt about the essence of pseudo code in program design

6.0 Tutor Marked Assignment

What is Pseudo code
Explain the General guidelines for writing Pseudo code.
Write a Pseudo code to find the average of even number between 1 and 20.

7.0 Futher Reading And Other Resouces

Robertson, L. A. (2003) *Simple Program Design: A Step-by-Step Approach.* 4th ed. Melbourne: Thomson.

Unit 5 **Programming Environment, CASE Tools & HIPO Diagrams**

1.0 Introduction

In the last unit, you have learnt about **pseudo code. In this unit you will be exposed to** Programming Environment, CASE Tools & HIPO Diagrams. After studying this unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to:

Explain Programming Environment

Discuss Case Tools.
Explain Hipo Diagrams.

3.0 Definition of Programming Environment

Programming environments gives the basic tools and Application Programming Interfaces, or APIs, necessary to construct programs. Programming environments help the creation, modification, execution and debugging of programs. The goal of integrating a programming environment is more than simply building tools that share a common data base and provide a consistent user interface. Altogether, the programming environment appears to the programmer as a single tool; there are no firewalls separating the various functions provided by the environment.

3.1 History of Programming Environment

The history of software tools began with the first computers in the early 1950s that used linkers, loaders, and control programs. In the early 1970s the tools became famous with Unix with tools like grep, awk and make that were meant to be combined flexibly with pipes. The term "software tools" came from the book of the same name by Brian Kernighan and P. J. Plauger. Originally, Tools were simple and light weight. As some tools have been maintained, they have been integrated into more powerful integrated development environments (IDEs). These environments combine functionality into one place, sometimes increasing simplicity and productivity, other times part with flexibility and extensibility. The workflow of IDEs is routinely contrasted with alternative approaches, such as the use of Unix shell tools with text editors like Vim and Emacs.

The difference between tools and applications is unclear. For example, developers use simple databases (such as a file containing a list of important values) all the time as tools. However a full-blown database is usually thought of as an application in its own right.

For many years, computer-assisted software engineering (CASE) tools were preferred. CASE tools emphasized design and architecture support, such as for UML. But the most successful of these tools are IDEs.

The ability to use a variety of tools productively is one quality of a skilled software engineer.

3.2 Types of Programming Environment

Software development tools can be roughly divided into the following categories:

performance analysis tools
debugging tools
static analysis and formal verification tools

correctness checking tools
memory usage tools
application build tools
integrated development environment

3.3 Forms of Software tools

Software tools come in many forms namely :

Bug Databases: Bugzilla, Trac, Atlassian Jira, LibreSource, SharpForge
Build Tools: Make, automake, Apache Ant, SCons, Rake, Flowtracer, cmake, qmake
Code coverage: C++test,GCT, Insure++, Jtest, CCover
Code Sharing Sites: Freshmeat, Krugle, Sourceforge. See also Code search engines.
Compilation and linking tools: GNU toolchain, gcc, Microsoft Visual Studio, CodeWarrior, Xcode, ICC

Debuggers: gdb, GNU Binutils, valgrind. Debugging tools also are used in the process of debugging code, and can also be used to create code that is more compliant to standards and portable than if they were not used.

Disassemblers: Generally reverse-engineering tools.
Documentation generators: Doxygen, help2man, POD, Javadoc, Pydoc/Epydoc, asciidoc
Formal methods: Mathematically-based techniques for specification, development and verification
GUI interface generators
Library interface generators: Swig
Integration Tools

Memory Use/Leaks/Corruptions Detection: dmalloc, Electric Fence, duma, Insure ++. Memory leak detection: In the C programming language for instance, memory leaks are not as easily detected - software tools called memory debuggers are often used to find memory leaks enabling the programmer to find these problems much more efficiently than inspection alone.

Parser generators: Lex, Yacc
Performance analysis or profiling
Refactoring Browser
Revision control: Bazaar, Bitkeeper, Bonsai, ClearCase, CVS, Git, GNU arch, Mercurial, Monotone, Perforce, PVCS, RCS, SCM, SCCS, SourceSafe, SVN, LibreSource Synchronizer
Scripting languages: Awk, Perl, Python, REXX, Ruby, Shell, Tcl
Search: grep, find
Source-Code Clones/Duplications Finding

Source code formatting
Source code generation tools
Static code analysis: C++test, Jtest, lint, Splint, PMD, Findbugs, .TEST
Text editors: emacs, vi, vim

3.4 Integrated development environments

Integrated development environments (IDEs) merge the features of many tools into one complete package. They are usually simpler and make it easier to do simple tasks, such as searching for content only in files in a particular project. IDEs are often used for development of enterprise-level applications.Some examples of IDEs are:

Delphi
C++ Builder (CodeGear)
Microsoft Visual Studio
EiffelStudio
GNAT Programming Studio
Xcode
IBM Rational Application Developer
Eclipse
NetBeans
IntelliJ IDEA
WinDev
Code::Blocks
Lazarus

3.5 What is CASE Tools?

CASE tools are a class of software that automates many of the activities involved in various life cycle phases. For example, when establishing the functional requirements of a proposed application, prototyping tools can be used to develop graphic models of application screens to assist end users to visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, as mentioned, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.

It is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.

3.6 Types of CASE Tools

Some typical CASE tools are:

Configuration management tools
Data modeling tools
Model transformation tools
Program transformation tools
Refactoring tools
Source code generation tools, and
Unified Modeling Language

Many CASE tools not only yield code but also generate other output typical of various systems analysis and design methodologies such as:

data flow diagram
entity relationship diagram
logical schema
Program specification
SSADM.
User documentation

3.7 History of CASE

The term CASE was originally formulated by software company, Nastec Corporation of Southfield, Michigan in 1982 with their original integrated graphics and text editor GraphiText, which also was the first microcomputer-based system to use hyperlinks to cross-reference text strings in documents Under the direction of Albert F. Case, Jr. vice president for product management and consulting, and Vaughn Frick, director of product management, the DesignAid product suite was expanded to support analysis of a wide range of structured analysis and design methodologies, notable Ed Yourdon and Tom DeMarco, Chris Gane & Trish Sarson, Ward-Mellor (real-time) SA/SD and Warnier-Orr (data driven).

The next competitor into the market was Excelerator from Index Technology in Cambridge, Mass. While DesignAid ran on Convergent Technologies and later Burroughs Ngen networked microcomputers, Index launched Excelerator on the IBM PC/AT platform. While, at the time of launch, and for several years, the IBM platform did not support networking or a centralized database as did the Convergent Technologies or Burroughs machines, the allure of IBM was strong, and Excelerator came to prominence. Hot on the heels of Excelerator were a rash of offerings from companies such as Knowledgeware (James Martin, Fran Tarkenton and Don Addington), Texas Instrument's IEF and Accenture's FOUNDATION toolset (METHOD/1, DESIGN/1, INSTALL/1, FCP).

CASE tools were at their peak in the early 1990s. At the time IBM had proposed AD/Cycle which was an alliance of software vendors centered around IBM's Software repository using IBM DB2 in mainframe and OS/2:

The application development tools can be from several sources: from IBM, from vendors, and from the customers themselves. IBM has entered into relationships with Bachman Information Systems, Index Technology Corporation, and Knowledgeware, Inc. wherein selected products from these vendors will be marketed through an IBM complementary marketing program to provide offerings that will help to achieve complete life-cycle coverage.

With the decline of the mainframe, AD/Cycle and the Big CASE tools died off, opening the market for the mainstream CASE tools of today. Interestingly, nearly all of the leaders of the CASE market of the early 1990s ended up being purchased by Computer Associates, including IEW, IEF, ADW, Cayenne, and Learmonth & Burchett Management Systems (LBMS).

3.8 Categories of Case Tools

CASE Tools can be classified into 3 categories:

Tools support only specific tasks in the software process.
Workbenches support only one or a few activities.
Environments support (a large part of) the software process.

Workbenches and environments are generally built as collections of tools. Tools can therefore be either stand alone products or components of workbenches and environments.

3.9 **CASE Environment**

An environment is a collection of CASE tools and workbenches that supports the software process. CASE environments are classified based on the focus/basis of integration

Toolkits
Language-centered
Integrated
Fourth generation
Process-centered

3.9.1 Toolkits

Toolkits are loosely integrated collections of products easily extended by aggregating different tools and workbenches. Typically, the support provided by a toolkit is limited to programming, configuration management and project management. And the toolkit itself is environments extended from basic sets of operating system tools, for example, the Unix Programmer's Work Bench and the VMS VAX Set. In addition, toolkits' loose integration requires user to activate tools by explicit invocation or simple control mechanisms. The resulting files are unstructured and could be in different format, therefore the access of file from different tools may require explicit file format conversion. However, since the only constraint for adding a new component is the formats of the files, toolkits can be easily and incrementally extended.

3.9.2 Language-centered

The environment itself is written in the programming language for which it was developed, thus enable users to reuse, customize and extend the environment. Integration of code in different languages is a major issue for language-centered environments. Lack of process and data integration is also a problem. The strengths of these environments include good level of presentation and control integration. Interlisp, Smalltalk, Rational, and KEE are examples of language-centered environments.

3.9.3 Integrated

These environments achieve presentation integration by providing uniform, consistent, and coherent tool and workbench interfaces. Data integration is achieved through the *repository* concept: they have a specialized database managing all information produced and accessed in the environment. Examples of integrated environment are IBM AD/Cycle and DEC Cohesion.

3.9.4 Fourth generation

Forth generation environments were the first integrated environments. They are sets of tools and workbenches supporting the development of a specific class of program: electronic data processing and business-oriented applications. In general, they include programming tools, simple configuration management tools, document handling facilities and, sometimes, a code generator to produce code in lower level languages. Informix 4GL, and Focus fall into this category.

3.9.5 Process-centered

Environments in this category focus on process integration with other integration dimensions as starting points. A process-centered environment operates by interpreting a process model created by specialized tools. They usually consist of tools handling two functions:

Process-model execution, and
Process-model production

Examples are East, Enterprise II, Process Wise, Process Weaver, and Arcadia.[6]

3.10 Application areas of CASE Tools

All aspects of the software development life cycle can be supported by software tools, and so the use of tools from across the spectrum can, arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

However, it is the tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, that are most frequently thought of as CASE tools. CASE applied, for instance, to a database software product, might normally involve:

Modeling business/real world processes and data flow
Development of data models in the form of entity-relationship diagrams
Development of process and function descriptions
Production of database creation SQL and stored procedures

3.11 CASE Risk

Common CASE risks and associated controls include:

Inadequate Standardization: Linking CASE tools from different vendors (design tool from Company X, programming tool from Company Y) may be difficult if the products do not use standardized code structures and data classifications. File formats can be converted, but usually not economically. Controls include using tools from the same vendor, or using tools based on standard protocols and insisting on demonstrated compatibility. Additionally, if organizations obtain tools for only a portion of the development process, they should consider acquiring them from a vendor that has a full line of products to ensure future compatibility if they add more tools.

Unrealistic Expectations: Organizations often implement CASE technologies to reduce development costs. Implementing CASE strategies usually involves high start-up costs. Generally, management must be willing to accept a long-term payback period. Controls include requiring senior managers to define their purpose and strategies for implementing CASE technologies.

Quick Implementation: Implementing CASE technologies can involve a significant change from traditional development environments. Typically, organizations should not use CASE tools the first time on critical projects or projects with short deadlines because of the lengthy training process. Additionally, organizations should consider using the tools on smaller, less complex projects and gradually implementing the tools to allow more training time.

Weak Repository Controls : Failure to adequately control access to CASE repositories may result in security breaches or damage to the work documents, system designs, or code modules stored in the repository. Controls include protecting the repositories with appropriate access, version, and backup controls.

3.12 HIPO Diagrams

The HIPO (Hierarchy plus Input-Process-Output) technique is a tool for planning and/or documenting a computer program. A HIPO model consists of a hierarchy chart that graphically represents the program's control structure and a set of IPO (Input-Process-Output) charts that describe the inputs to, the outputs from, and the functions (or processes) performed by each module on the hierarchy chart.

3.13 Strengths, weaknesses, and limitations

Using the HIPO technique, designers can evaluate and refine a program's design, and correct flaws prior to implementation. Given the graphic nature of HIPO, users and managers can easily follow a program's structure. The hierarchy chart serves as a useful planning and visualization document for managing the program development process. The IPO charts define for the programmer each module's inputs, outputs, and algorithms.

In theory, HIPO provides valuable long-term documentation. However, the "text plus flowchart" nature of the IPO charts makes them difficult to maintain, so the documentation often does not represent the current state of the program.

By its very nature, the HIPO technique is best used to plan and/or document a hierarchically structured program.

The HIPO technique is often used to plan or document a structured program A variety of tools, including pseudocode (and structured English can be used to describe processes on an IPO chart. System flowcharting symbols are sometimes used to identify physical input, output, and storage devices on an IPO chart.

3.14 Components of HIPO

A completed HIPO package has two parts. A hierarchy chart is used to represent the top-down structure of the program. For each module depicted on the hierarchy chart, an IPO (Input-Process-Output) chart is used to describe the inputs to, the outputs from, and the process performed by the module.

3.14.1 The hierarchy chart

It summarises the primary tasks to be performed by an interactive inventory program. Figure 7 shows one possible hierarchy chart (or visual table of contents) for that program. Each box represents one module that can call its subordinates and return control to its higher-level parent.

A Set of Tasks to Be Performed by an Interactive Inventory Program is:

Manage inventory
Update stock
Process sale
Process return
Process shipment
Generate report
Respond to query
Display status report
Maintain inventory data
Modify record
Add record
Delete record

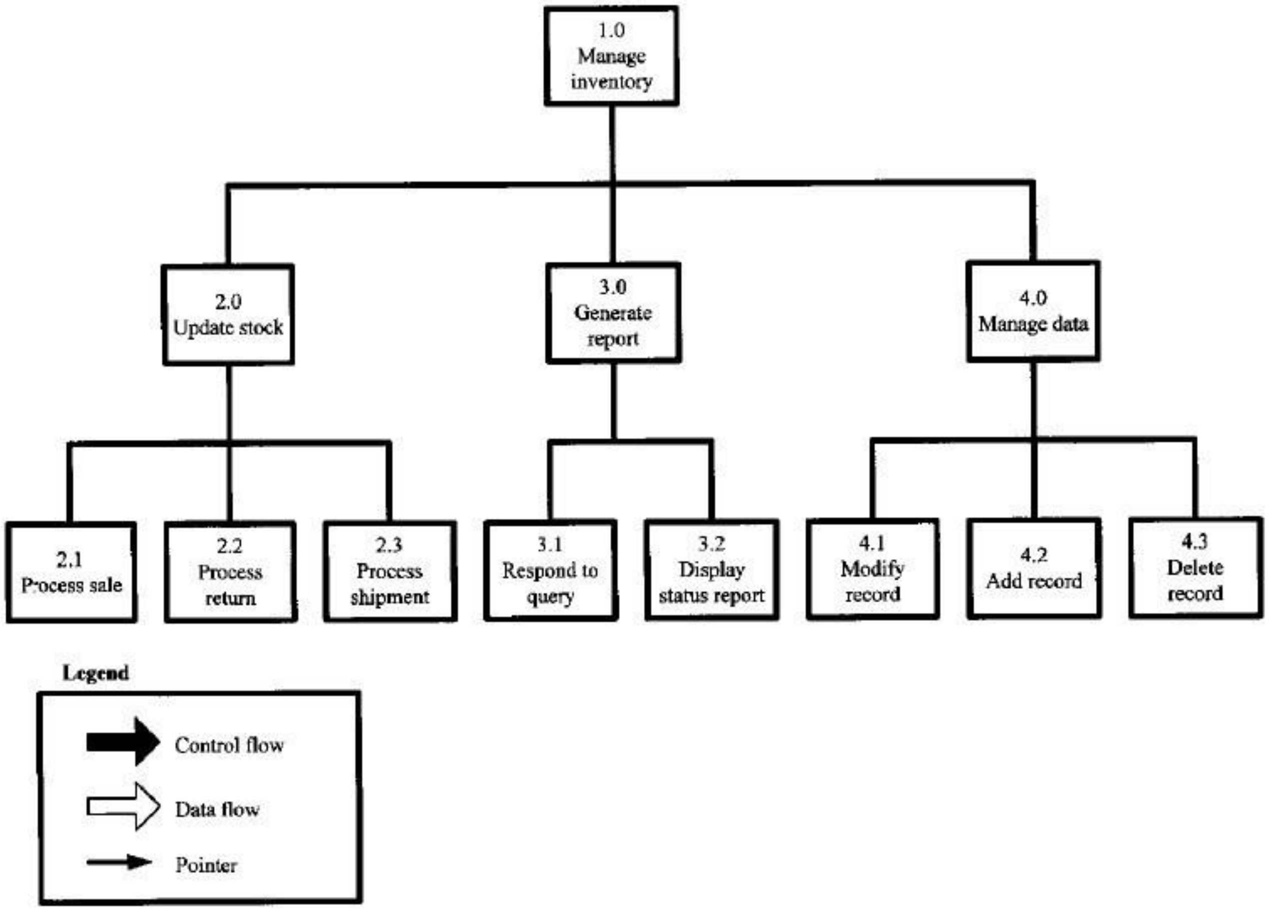


Figure 7 A hierarchy chart for an interactive inventory control program.

Source: www.hit.ac.il/staff/leonidM/information-systems/ch64.html

At the top of Figure 7 is the main control module, *Manage inventory* (module 1.0). It accepts a transaction, determines the transaction type, and calls one of its three subordinates (modules 2.0, 3.0, and 4.0).

Lower-level modules are identified relative to their parent modules; for example, modules 2.1, 2.2, and 2.3 are subordinates of module 2.0, modules 2.1.1, 2.1.2, and 2.1.3 are subordinates of 2.1, and so on. The module names consist of an active verb followed by a subject that suggests the module's function.

The objective of the module identifiers is to uniquely identify each module and to indicate its place in the hierarchy. Some designers use Roman numerals (level I, level II) or letters (level A, level B) to designate levels. Others prefer a hierarchical numbering scheme; e.g., 1.0 for the first level; 1.1, 1.2, 1.3 for the second level; and so on. The key is consistency.

The box at the lower-left of Figure 7 is a legend that explains how the arrows on the hierarchy chart and the IPO charts are to be interpreted. By default, a wide clear arrow represents a data flow, a wide black arrow represents a control flow, and a narrow arrow indicates a pointer.

3.14.2 The IPO charts

An IPO chart is prepared to document each of the modules on the hierarchy chart.

3.14.2.1 Overview diagrams

An overview diagram is a high-level IPO chart that summarizes the inputs to, processes or tasks performed by, and outputs from a module. For example, shows an overview diagram for process 2.0, *Update stock*. Where appropriate, system flowcharting symbols are used to identify the physical devices that generate the inputs and accept the outputs. The processes are typically described in brief paragraph or sentence form. Arrows show the primary input and output data flows.

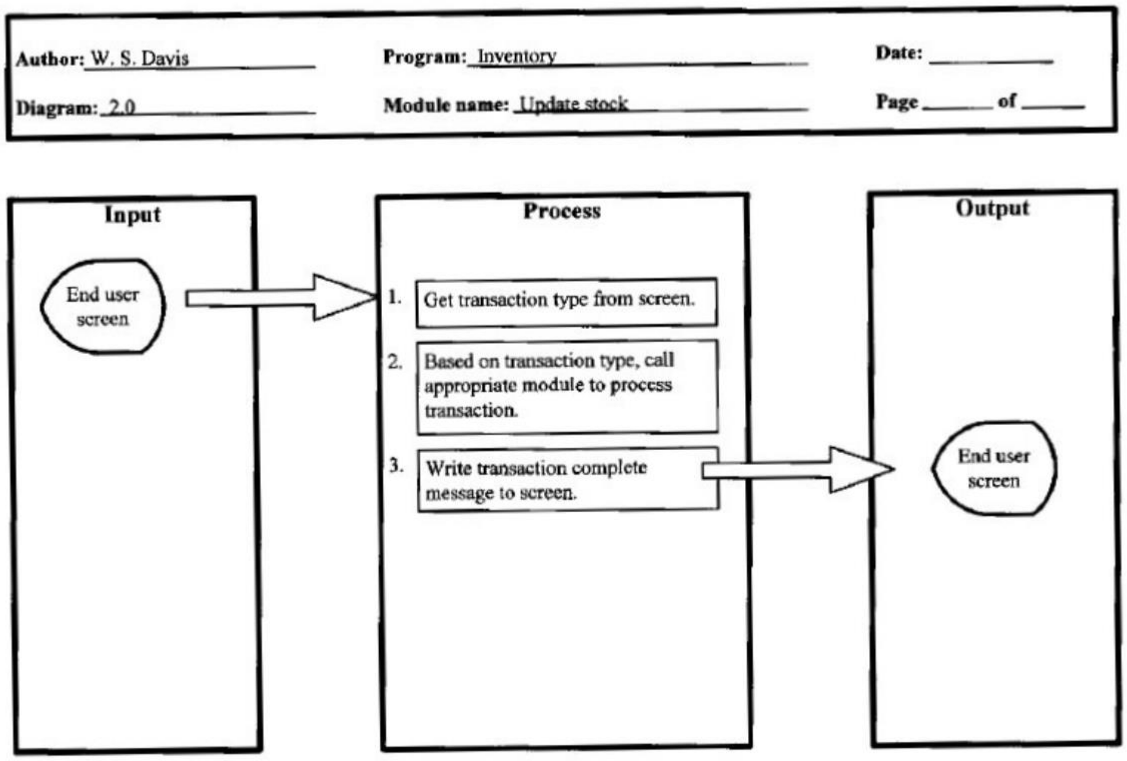

Author: W. S. Davis
Program: Inventory
Date:
Diagram: 2.0
Module name: Update stock
Page of
Input
Process
Output
End user screen
1. Get transaction type from screen.
2. Based on transaction type, call appropriate module to process transaction.
3. Write transaction complete message to screen.
End user screen

Figure 7.1 An overview diagram for process 2.0.

Source: www.hit.ac.il/staff/leonidM/information-systems/ch64.html

Overview diagrams are primarily planning tools. They often do not appear in the completed documentation package.

3.14.2.2 Detail diagrams

A detail diagram is a low-level IPO chart that shows how specific input and output data elements or data structures are linked to specific processes. In effect, the designer integrates a system flowchart into the overview diagram to show the flow of data and control through the module.

Figure 7.2 shows a detail diagram for module 2.0, *Update stock*. The process steps are written in pseudocode. Note that the first step writes a menu to the user screen and input data (the transaction type) flows from that screen to step 2. Step 3 is a case structure. Step 4 writes a *transaction complete* message to the user screen.

The solid black arrows at the top and bottom of the process box show that control flows from module 1.0 and, upon completion, returns to module 1.0. Within the case structure (step 3) are other solid black arrows.

Following case 0 is a return (to module 1.0). The two-headed black arrows following cases 1, 2, and 3 represent subroutine calls; the off-page connector symbols (the little home plates) identify each subroutine's module number. Note that each subroutine is documented in a separate IPO chart. Following the default case, the arrow points to an on-page connector symbol numbered 1. Note the matching on-page connector symbol pointing to the select structure. On-page connectors are also used to avoid crossing arrows on data flows.

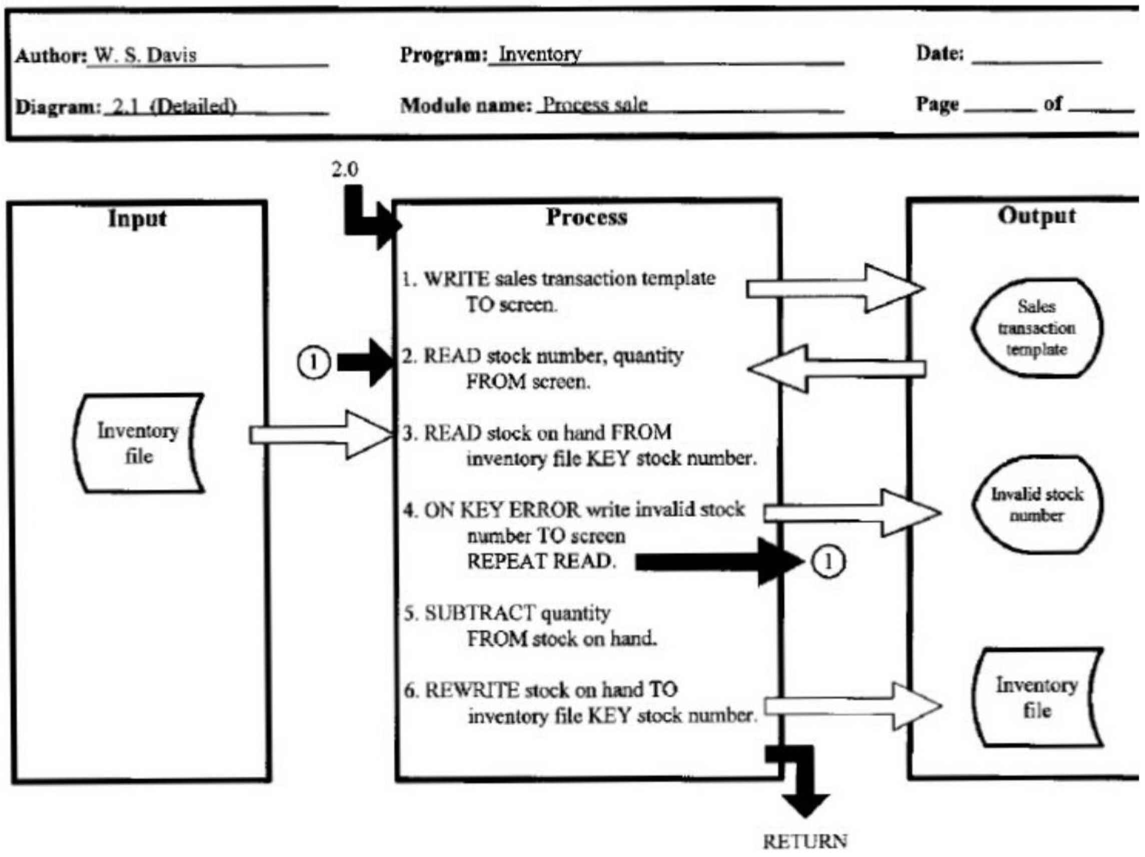

Author: W. S. Davis
Program: Inventory
Date:
Diagram: 2.1 (Detailed)
Module name: Process sale
Page of
2.0
Input
Process
Output
1. WRITE sales transaction template TO screen.
2. READ stock number, quantity FROM screen.
3. READ stock on hand FROM inventory file KEY stock number.
4. ON KEY ERROR write invalid stock number TO screen REPEAT READ.
5. SUBTRACT quantity FROM stock on hand.
6. REWRITE stock on hand TO inventory file KEY stock number.
1
1
Inventory file
Sales transaction template
Invalid stock number
Inventory file
RETURN

Figure 7.2 A detail diagram for process 2.1.

Source: www.hit.ac.il/staff/leonidM/information-systems/ch64.html

Often, detailed notes and explanations are written on an extended description that is attached to each detail diagram. The notes might specify access methods, data types, and so on.

Figure 64.4 shows a detail diagram for process 2.1. The module writes a template to the user screen, reads a stock number and a quantity from the screen, uses the stock number as a key to access an inventory file, and updates the stock on hand. Note that the logic repeats the data entry process if the stock number does not match an inventory record. A real IPO chart is likely to show the error response process in greater detail.

3.14.2.3 Simplified IPO charts

Some designers simplify the IPO charts by eliminating the arrows and system flowchart symbols and showing only the text. Often, the input and out put blocks are moved above the process block (Figure 64.5), yielding a form that fits better on a standard 8.5 × 11 (portrait orientation) sheet of paper. Some programmers insert modified IPO charts similar to Figure 64.5 directly into their source code as comments. Because the documentation is closely linked to the code, it is often more reliable than stand-alone HIPO documentation, and more likely to be maintained.

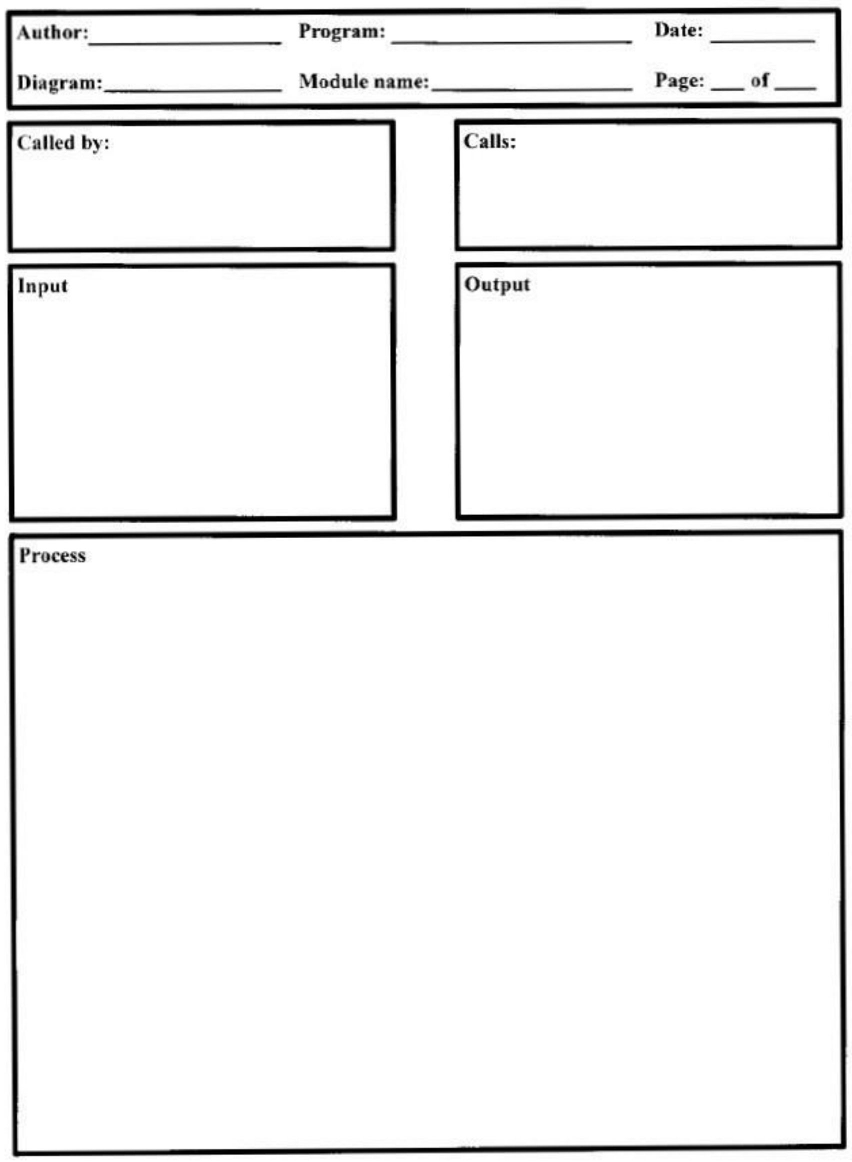


Fig 7.3 Simplified HIPO diaram

Source: www.hit.ac.il/staff/leonidM/information-systems/ch64.html

Detail diagram —
A low-level IPO chart that shows how specific input and output data elements or data structures are linked to specific processes.

Hierarchy chart —
A diagram that graphically represents a program's control structure.

HIPO (Hierarchy plus Input-Process-Output) —
A tool for planning and/or documenting a computer program that utilizes a hierarchy chart to graphically represent the program's control structure and a set

of IPO (Input-Process-Output) charts to describe the inputs to, the outputs from, and the functions performed by each module on the hierarchy chart.

IPO (Input-Process-Output) chart —
A chart that describes or documents the inputs to, the outputs from, and the functions (or processes) performed by a program module.

Overview diagram —
A high-level IPO chart that summarizes the inputs to, processes or tasks performed by, and outputs from a module.

Visual Table of Contents (VTOC) —
A more formal name for a hierarchy chart.

3.15 Software

In the 1970s and early 1980s, HIPO documentation was typically prepared by hand using a template. Some CASE products and charting programs include HIPO support. Some forms generation programs can be used to generate HIPO forms. The examples in this # were prepared using Visio.

Activity J Discuss the historical development of Case Tools

4.0 Conclusion

Programming tools are so important for effective program design.

5.0 Summary.

In this unit, you have learnt that:

Programming environments gives the basic tools and Application Programming Interfaces, or APIs, necessary to construct programs.

Using the HIPO technique, designers can evaluate and refine a program's design, and correct flaws prior to implementation.

CASE tools are a class of software that automates many of the activities involved in various life cycle phases

6.0 Tutor Marked Assignment

Explain Programming Environment
What is Case Tools?, enumerate different categories of case tools
What is HIPO technique?
With the aid of well labeled diagrams, discuss the components of Hipo.

7.0 Further Reading And Other Resources

Gane, C. and Sarson, T., *Structured Systems Analysis: Tools and Techniques,* Prentice-Hall, Englewood Cliffs, NJ, 1979.

IBM Corporation, *HIPO—A Design Aid and Documentation Technique*, Publication Number GC20-1851, IBM Corporation, White Plains, NY, 1974.
Katzan, H., Jr., *Systems Design and Documentation: An Introduction to the HIPO Method,* Van Nostrand Reinhold, New York, 1976.

Peters, L. J., *Software Design: Methods and Techniques,* Yourdon Press, New York, 1981.

Yourdon, E. and Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979.

Module 3 Implementation and Testing

Unit 1 Implementation

1.0 Introduction

This unit examines the implementation phase of software development. After studying the unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to: Define
clearly software Implementation Differentiate
between the three types of errors
Explain the application of Six Sigma to Software Implementation Projects
Discuss the Major Tasks in Implementation
Explain the Major Requirement in
Implementation Explain the Implementation
Support

3.0 What is Implementation?

Code is formed from the deliverables of the design phase during implementation. It is the longest phase of the software development life cycle. Since code is produce here, the developer regards this phase as the main focus of the life cycle. Implementation my overlap with both the design and testing phases. As we learnt in previous unit many tools exists (CASE tools) to actually automate the production of code using information gathered and produced during the design phase. The implementation phase concerns with issues of quality, performance, baselines, libraries, and debugging. The end deliverable is the product itself.

3.1 The Implementation Phase

Phase	Deliverable
Implementation	● Code
	● Critical Error Removal

Table 1 The Implementation Phase

Source: Ronald LeRoi Burback 1998-12-14

. 3.2 **Critical Error Removal**

There are three kinds of errors in a system, namely critical errors, non-critical errors, and unknown errors.

3.2.1 A **critical error** prevents the system from fully satisfying its usage. The errors have to be corrected before the system can be given to a customer or even before future development can progress.

3.2.2 A **non-critical error** is known but the occurrence of the error does not notably affect the system's expected quality. There may indeed be many known errors in the system. They are usually listed in the release notes and have well established work arounds.

Actually, the system is likely to have many, yet-to-be-discovered errors. The outcome of these errors is unknown. Some may become critical while some may be simply fixed by patches or fixed in the next release of the system.

3.3 Application of Six Sigma to Software Implementation Projects

Software implementation can be a demanding project. When a company is attempting new software integration, it can be hectic Six Sigma is a management approach meant to discover and control defects. A summary of Six Sigma can be found in Natasha Baker’s “Key Concepts of Six Sigma.” The technique consists of five steps:

· Define
· Measure
· Analyze
· Improve
· Control

Defining the Implementation

By defining the goals, projects, and deliverables your company will have greater direction during the changeover. The goals and projects *must* be measurable. The following questions, for examples, may be necessary: Is it your goal to have 25% of your staff comfortable enough to train the remaining staff? Do you want full implementation of the software by March? By utilizing Six Sigma metrics careful monitoring of team productivity and implementation success is possible.

2. Measurement of the Implementation

Goals and projects must be usable with metrics. By using Six Sigma measurement methods, it is possible to follow user understanding, familiarity, and progress accurately. It should be noted that, continuous data is more useful than discrete data. This is because it gives a better implementation success rate overview.

3. Implementation Analysis

Analysis is important to tackle defects occurrence. The Six Sigma method examines essential relationships and ensures all factors are considered. For example, in a software implementation trial, employees are frustrated and confused by new processes. Careful analysis will look at the reasons behind the confusion.

4. Implementation Improvement

After analysis, it is important to look at how the implementation could improve. In the example utilizing the team members, perhaps utilizing proficient resources to mentor struggling resources will help. Six Sigma improvements depends upon experimental design and carefully constructed analysis of data in order to keep further defects in the implementation process at bay.

5. Control of the Implementation

If implementation is going to be successful, control is important. It involves consistent monitoring for proficiency. This ensures that the implementation does not fail. Any deviations from the goals set demand correcting before they become defects. For example, if you notice your team does not adapt quickly enough, you need to identify the causes *before* the deadline. By carefully monitoring the implementation process this way, will minimise the defect. The two most important features of software implementation using Six Sigma are setting measurable goals and employing metrics in order to maximize improvement and minimize the chance of defects in the new process.

3.4 Major Tasks in Implementation

This part provides a brief description of each major task needed for the implementation of the system. The tasks described here are not particular to site but overall project tasks that are needed to install hardware and software, prepare
data, and verify the system. Include the following information for the description of each major task, if appropriate:
Add as many subsections as necessary to this section to describe all the major tasks adequately. The tasks described in this section are not site-specific, but generic or overall project tasks that are required to install hardware and software, prepare data, and verify the system.

Examples of major tasks are the following:

Providing overall planning and coordination for the implementation
Providing appropriate training for personnel
Ensuring that all manuals applicable to the implementation effort are available when needed
Providing all needed technical assistance

Scheduling any special computer processing required for the implementation
Performing site surveys before implementation
Ensuring that all prerequisites have been fulfilled before the implementation date
Providing personnel for the implementation team
Acquiring special hardware or software
Performing data conversion before loading data into the system
Preparing site facilities for implementation

3.5 Major Requirement in Implementation

3.5.1 Security

If suitable for the system to be implemented, there is need to include an overview of the system security features and requirements during the implementation.

3.5.1.1 System Security Features

It is pertinent to discuss the security features that will be associated with the system when it is implemented. It should include the primary security features associated with the system hardware and software. Security and protection of sensitive bureau data and information should be discussed.

3.5.1.2 Security during Implementation

This part addresses security issues particularly related to the implementation effort. It will be necessary to consider for example, if LAN servers or workstations will he installed at a site with sensitive data preloaded on non-removable hard disk drives. It will also be important to see to how security would be provided for the data on these devices during shipping, transport, and installation so as not to allow theft of the devices to compromise the sensitive data.

3.6 Implementation Support

This part describes the support such as: software, materials, equipment, and facilities necessary for the implementation, as well as the personnel requirements and training essential for the implementation.

3.6.1 Hardware, Software, Facilities, and Materials

This section, provides a list of support software, materials, equipment, and facilities required for the implementation..

3.6.1.1 Hardware

This section offers a list of support equipment and includes all hardware used for testing time implementation. For example, if a client/server database is implemented on a LAN, a network monitor or “sniffer” might be used, along with test programs. to determine the performance of the database and LAN at high- utilization rates

3.6.1.2 Software

This section provides a list of software and databases required to support the implementation. Identify the software by name, code, or acronym. Identify which software is commercial off-the-shelf and which is State-specific. Identify any software used to facilitate the implementation process **3.6.1.3 Facilities**

This section identifies the physical facilities and accommodations required during implementation. Examples include physical workspace for assembling and testing hardware components, desk space for software installers, and classroom space for training the implementation stall. Specify the hours per day needed, number of days, and anticipated dates.

3.6.1.4 Material

This section provides a list of required support materials, such as magnetic tapes and disk packs.

3.7 Personnel

This section describes personnel requirements and any known or proposed staffing requirements. It also describes the training, to be provided for the implementation staff.

3.7.1 Personnel Requirements and Staffing

This section, describes the number of personnel, length of time needed, types of skills, and skill levels for the staff required during the implementation period. If particular staff members have been selected or proposed for the implementation, identify them and their roles in the implementation.

3.7.2 Training of Implementation Staff

This section addresses the training, necessary to prepare staff for implementing and maintaining the system; it does not address user training, which is the subject of the Training Plan. It also describes the type and amount of training required for each of the following areas, if appropriate, for the system:

System hardware/software installation
System support
System maintenance and modification

Present a training curriculum listing the courses that will be provided, a course sequence and a proposed schedule. If appropriate, identify which courses particular types of staff should attend by job position description.

If training will be provided by one or more commercial vendors, identify them, the course name(s), and a brief description of the course content.

If the training will be provided by State staff, provide the course name(s) and an outline of the content of each course. Identify the resources, support materials, and proposed instructors required to teach the course(s).

3.8 Performance Monitoring

This section describes the performance monitoring tool and techniques and how it will be used to help decide if the implementation is successful.

3.9 Configuration Management Interface

This section describes the interactions required with the Configuration Management (CM) representative on CM-related issues, such as when software listings will be distributed, and how to confirm that libraries have been moved from the development to the production environment.

3.10 Implementation Requirements by Site

This section describes specific implementation requirements and procedures. If these requirements and procedures differ by site, repeat these subsections for each site; if they are the same for each site, or if there is only one implementation site, use these subsections only once. The “X” in the subsection number should be replaced with a sequenced number beginning with I. Each subsection with the same value of “X” is associated with the same implementation site. If a complete set of subsections will be associated with each implementation site, then “X” is assigned a new value for each site.

3.10.1 Site Name or identification for Site X

This section provides the name of the specific site or sites to be discussed in the subsequent sections.

3.10.2 Site Requirements

This section defines the requirements that must he met for the orderly implementation of the system and describes the hardware, software, and site-specific facilities requirements for this area.

Any site requirements that do not fall into the following three categories and were not described in Section 3, Implementation Support, may be described in this section, or other subsections may be added following Facilities Requirements below:

Hardware Requirements - Describe the site-specific hardware requirements necessary to support the implementation (such as. LAN hardware for a client/server database designed to run on a LAN).

Software Requirements - Describe any software required to implement the system (such as, software specifically designed for automating the installation process).

Data Requirements - Describe specific data preparation requirements and data that must be available for the system implementation. An example would be the assignment of individual IDs associated with data preparation.

Facilities Requirements - Describe the site-specific physical facilities and accommodations required during the system implementation period. Some examples of this type of information are provided in Section *3*.

3.10.3 Site implementation Details

This section addresses the specifics of the implementation for this site. Include a description of the implementation team, schedule, procedures, and database and data updates. This section should also provide information on the following:

Team--If an implementation team is required, describe its composition and the tasks to be performed at this site by each team member.

Schedule--Provide a schedule of activities, including planning and preparation, to be accomplished during implementation at this site. Describe the required tasks in chronological order with the beginning and end dates of each task. If appropriate, charts and graphics may be used to present the schedule.

Procedures--Provide a sequence of detailed procedures required to accomplish the specific hardware and software implementation at this site. If necessary, other documents may be referenced. If appropriate, include a step-by-step sequence of the detailed procedures. A checklist of the installation events may he provided to record the results of the process.

If the site operations startup is an important factor in the implementation, then address startup procedures in some detail. If the system will replace an already operating system, then address the startup and cutover processes in detail. If there is a period of parallel operations with an existing system, address the startup procedures that include technical and operations support during the parallel cycle and the consistency of data within the databases of the two systems.

Database--Describe the database environment where the software system and the database(s), if any, will be installed. Include a description of the different types of database and library environments (such as, production, test, and training databases).

Include the host computer database operating procedures, database file and library naming conventions, database system generation parameters, and any other information needed to effectively establish the system database environment.
Include database administration procedures for testing changes, if any, to the database management system before the system implementation.

Data Update--If data update procedures are described in another document, such as the operations manual or conversion plan, that document may be referenced here. The following are examples of information to be included:

- Control inputs
- Operating instructions
- Database data sources and inputs
- Output reports
- Restart and recovery procedures

3.11 Back-Off Plan

This section specifies when to make the go/no go decision and the factors to be included in making the decision. The plan then goes on to provide a detailed list of steps and actions required to restore the site to the original, pre-conversion condition,

3.12 Post-Implementation Verification

This section describes the process for reviewing the implementation and deciding if it was successful. It describes how an action item list will be created to rectify any noted discrepancies. It also references the Back-Off Plan for instructions on how to back-out the installation, if, as a result of the post-implementation verification, a no-go decision is made.

Activity K Explain the Major Requirement in Implementation

4.0 Conclusion

Implementation phase is vital aspect of software development. It is the longest phase of the software development life cycle. It is a phase where code is produced and as such tge developer regards it as the main focus of the software development life cycle.

5.0 Summary

In this unit**,** you have learnt that:

Code is formed from the deliverables of the design phase during implementation.
A **critical error** prevents the system from fully satisfying its usage. The errors have to be corrected before the system can be given to a customer or even before future development can progress.
A **non-critical error** is known but the occurrence of the error does not notably affect the system's expected quality.
The system is likely to have many, yet-to-be-discovered errors known as unknown errors which may become critical while some may be simply fixed by patches or fixed in the next release of the system.
The technique Six Sigma to Software Implementation Projects consists of five steps: Define, Measure, Analyze,· Improve, Control.

The Major Tasks in Implementation include: Providing overall planning and coordination for the implementation, Providing appropriate training for personnel Ensuring that all manuals applicable to the implementation effort are available when needed, Providing all needed technical assistance, Scheduling any special computer processing required for the implementation, Performing site surveys before implementation, Ensuring that all prerequisites have been fulfilled before the implementation date, Providing personnel for the implementation team, Acquiring special hardware or software, Performing data conversion before loading data into the system, Preparing site facilities for implementation.

Major Requirement in Implementation include: Security, Implementation Support, Personnel and Performance Monitoring

6.0 Tutor Marked Assignment

What is software Implementation
Differentiate between critical, non-critical and unknown errors
Explain the application of Six Sigma Software Implementation techniques.
Discuss the Major Tasks in Implementation
Explain the various Implementation Support

7.0 Further Reading And Other Resources

Moshe Bar and Karl Franz Fogel. *Open Source Development with CVS*. The Coriolis Group, Scottsdale, AZ, 2001.

Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, July 1998.

Stephen P. Berczuk and Brad Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, Boston, MA, 2002.

Don Bolinger, Tan Bronson, and Mike Loukides. *Applying RCS and SCCS: From Source Control to Project Control*. O'Reilly and Associates, Sebastopol, CA, 1995.

Unit 2 Testing Phase

1.0 Introduction

In the last unit, we looked at implementation phase of software development. In this unit, we shall consider the testing phase. It is important for stakeholders to have information

about the quality of product (software), hence the importance of testing cannot be overemphasised.

2.0 Objectives

By the end of this unit, you will be able to:

Define clearly software testing
Explain testing methods.
Explain software testing process
Explain testing tools

3.0 Definition of software testing

Software testing is an empirical examination carried out to provide stakeholders with information about the quality of the product or service under test. Software Testing in addition provides an objective, independent view of the software to allow the business to value and comprehend the risks associated with implementation of the software.. Software Testing can also be viewed as the process of validating and verifying that a software program/application/product (1) meets the business and technical requirements that guided its design and development; (2) works as expected; and (3) can be implemented with the same characteristics. It is important to note that depending on the testing method used, software testing, can be applied at any time in the development process, though most of the test effort occurs after the requirements have been defined and the coding process has been completed.

Testing can never totally detect all the defects within software. Instead, it provides a *comparison* that put side by side the state and behavior of the product against the instrument someone applies to recognize a problem. These instruments may include specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For instance, the audience for video game software is completely different from banking software. Software testing therefore, is the process of attempting to make this assessment whether the software product will be satisfactory to its end users, its target audience, its purchasers, and other stakeholders.

3.1 Brief History of software testing

In 1979, Glenford J. Myers introduced the separation of debugging from testing, illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. 1988, Dave Gelperin and William C. Hetzel classified the phases and goals in software testing in the following stages:

Until 1956 - Debugging oriented.
1957–1978 - Demonstration oriented.

1983–1987 - Evaluation oriented.
1988–2000 - Prevention oriented.

3.2 Testing methods

Traditionally, software testing methods are divided into **black box** testing ,**white box** testing and Grey **Box** Testing. A test engineer used these approaches to describe his opinion when designing test cases.

3.2.1.1 Black box testing

Black box testing considers the software as a "black box" in the sense that there is no knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

3.2.1.1 Specification-based testing: Specification-based testing intends to test the functionality of software based on the applicable requirements. Consequently, the tester inputs data into, and only sees the output from, the test object. This level of testing usually needs thorough test cases to be supplied to the tester, who can then verify that for a given input, the output value ,either "is" or "is not" the same as the expected value specified in the test case.
Specification-based testing though necessary, but it is insufficient to guard against certain risks.
Merits and Demerits: The black box testing has the advantage of "an unaffiliated opinion in the sense that there is no "bonds" with the code and the perception of the tester is very simple. He believes a code must have bugs and he goes for it. *But*, on the other hand, black box testing has the disadvantage of blind exploring because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

3.2.1.2 White box testing

In a **White box testing** the tester has the privilege to the internal data structures and algorithms including the code that implement these.

Types of white box testing
White box testing is of different types namely:

API testing (application programming interface) - Testing of the application using Public and Private APIs

Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
Fault injection methods
Mutation testing methods
Static testing - White box testing includes all static testing

3.2.1.3 Grey Box Testing

Grey box testing requires gaining access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output cannot be regarded as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This difference is important especially when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, changing a data repository can be seen as grey box, because the use would not ordinarily be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to ascertain boundary values or error messages.

3.2.2 Integration Testing

Integration testing is any type of software testing that seeks to reveal clash of individual software modules to each other. Such integration flaws can result, when the new modules are developed in separate *branches*, and then integrated into the main project.

3.2.3 Regression Testing

Regression testing is any type of software testing that attempts to reveal software regressions. Regression of the nature can occurs at any time software functionality, that was previously working correctly, stops working as anticipated. Usually, regressions occur as an unplanned result of program changes, when the newly developed part of the software collides with the previously existing. Methods of regression testing include re-running previously run tests and finding out whether previously repaired faults have re-appeared. The extent of testing depends on the phase in the release process and the risk of the added features.

3.2.4 Acceptance testing

One of two things below can be regarded as Acceptance testing:

1. A smoke test which is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing performed by the customer, usually in their lab environment on their own HW, is known as user acceptance testing (UAT).

3.2.5 Non Functional Software Testing

The following methods are used to test non-functional aspects of software:

Performance testing confirms to see if the software can deal with large quantities of data or users. This is generally referred to as software scalability. This activity of Non Functional Software Testing is often referred to as Endurance Testing.
Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of Non Functional Software Testing is oftentimes referred to as load (or endurance) testing.
Usability testing is used to check if the user interface is easy to use and understand.
Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.
Internationalization and localization is needed to test these aspects of software, for which a pseudo localization method can be used.

Compare to functional testing, which establishes the correct operation of the software in that it matches the expected behavior defined in the design requirements, non-functional testing confirms that the software functions properly even when it receives invalid or unexpected inputs. Non-functional testing, especially for software, is meant to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. An example of non-functional testing is software fault injection, in the form of fuzzing.

3.2.6 Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

3.3 Testing process

Testing process can take two forms: Usually the testing can be performed by an independent group of testers after the functionality is developed before it is sent to the customer. Another practice is to start software testing at the same time the project starts and it continues until the project finishes. The first practice always results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing.

Testing can be done on the following levels:

Unit testing tests the minimal software component, or module. Each unit (basic component) of the software is tested to verify that the detailed design for the unit has been correctly implemented. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

Integration testing exposes defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.
System testing tests a completely integrated system to verify that it meets its requirements.
System integration testing verifies that a system is integrated to any external or third party systems defined in the system requirements.

Before shipping the final version of software, *alpha* and *beta* testing are often done additionally:

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.
Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Finally, acceptance testing can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Benchmarks may be employed during regression testing to ensure that the performance of the newly modified software will be at least as acceptable as the earlier version or, in the case of code optimization, that some real improvement has been achieved.

3.4.2 Testing Tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

Program monitors, permitting full or partial monitoring of program code including:

- Instruction Set Simulator, permitting complete instruction level monitoring and trace facilities
- Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
- Code coverage reports

Formatted dump or Symbolic debugging, tools allowing inspection of program variables on error or at chosen points
Automated functional GUI testing tools are used to repeat system-level tests through the GUI
Benchmarks, allowing run-time performance comparisons to be made
Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Activity l Discuss the various testing methods.

4.0 **Conclusion**

It has been made abundantly clear that software testing is so important in assessing whether the software product will be satisfactory to its end users, its target audience, its purchasers, and other stakeholders.

5.0 Summary

In this unit, you have learnt that:

Software testing is an empirical examination carried out to provide stakeholders with information about the quality of the product or service under test.
Traditionally, software testing methods are divided into **black box** testing, white **box** testing and Grey **Box** Testing. A test engineer used these approaches to describe his opinion when designing test cases.
Black box testing considers the software as a "black box" in the sense that there is no knowledge of internal implementation.
In White **box testing** the tester has the privilege to the internal data structures and algorithms including the code that implement these.
Grey box testing requires gaining access to internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level.

Manipulating input data and formatting output cannot be regarded as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test.
Testing process can take two forms: (1) usually the testing can be performed by an independent group of testers after the functionality is developed before it is sent to the customer. (2) Another practice is to start software testing at the same time the project starts and it continues until the project finishes. The first practice always results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing.
Testing/debug tools include features such as:

- **Program monitors,** permitting full or partial monitoring of program code
- **Formatted dump or Symbolic** debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI

- o Benchmarks, allowing run-time performance comparisons to be made
- o **Performance analysis** (or profiling tools) that can help to highlight hot spots and resource usage

6.0 Tutor-Marked Assignment

What is software testing?
Explain software testing process
Explain testing tools

7.0 Further Reading And Other Resources

Exploratory Testing, Cem Kaner, Florida Institute of Technology, *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL, November 2006

Software errors cost U.S. economy $59.5 billion annually, NIST report

Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. ISBN 0-471-04328-1.

Dr. Dobb's journal of software tools for the professional programmer (M&T Pub) **12** (1-6): 116. 1987.

Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782. Laycock, G. T. (1993) (PostScript). *The Theory and Practice of Specification Based Software Testing*. Dept of Computer Science, Sheffield University, UK. http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz. Retrieved 2008-02-13.

Unit 3 Software Quality Assurance (SQA)

1.0 Introduction

In the last unit, we looked at testing phase of software development. In this unit, we shall consider the Software Quality Assurance (SQA). There is need to ensure that the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and standards are followed hence the need for Software Quality Assurance cannot be underestimated.

2.0 Objectives

By the end of this unit, you will be able to: Define
clearly Software Quality Assurance Explain the
concept of standards and procedures. Discuss
Software Quality Assurance Activities
Discuss SQA Relationships to Other Assurance Activities
Discuss Software Quality Assurance During the Software Acquisition Life Cycle.

3.0 Concepts and Definitions

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

3.1 Standards and Procedures

Establishing standards and procedures for software development is critical, since these provide the structure from which the software evolves. Standards are the established yardsticks to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared.

Standards and procedures establish the prescribed methods for developing software; the SQA role is to ensure their existence and adequacy. Proper documentation of standards and procedures is necessary since the SQA activities of process monitoring, product evaluation and auditing rely upon clear definitions to measure project compliance.

3.1.1 Types of standards include:

Documentation Standards specify form and content for planning, control, and product documentation and provide consistency throughout a project.

Design Standards specify the form and content of the design product. They provide rules and methods for translating the software requirements into the software design and for representing it in the design documentation.

Code Standards specify the language in which the code is to be written and define any restrictions on use of language features. They define legal language structures, style conventions, rules for data structures and interfaces, and internal code documentation.

Procedures are explicit steps to be followed in carrying out a process. All processes should have documented procedures. Examples of processes for which procedures are needed are configuration management, non-conformance reporting and corrective action, testing, and formal inspections.

If developed according to the NASA DID, the Management Plan describes the software development control processes, such as configuration management, for which there have to be procedures, and contains a list of the product standards.

Standards are to be documented according to the Standards and Guidelines DID in the Product Specification. The planning activities required to assure that both products and processes comply with designated standards and procedures are described in the QA portion of the Management Plan.

3.2 Software Quality Assurance Activities

Product evaluation and process monitoring are the SQA activities that assure the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and

standards are followed. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring. Review of the Management Plan should ensure that appropriate SQA approval points are built into these processes.

3.2.1 Product evaluation is an SQA activity that assures standards are being followed. Ideally, the first products monitored by SQA should be the project's standards and procedures. SQA assures that clear and achievable standards exist and then evaluates compliance of the software product to the established standards. Product evaluation assures that the software product reflects the requirements of the applicable standard(s) as identified in the Management Plan.

3.2.2 Process monitoring is an SQA activity that ensures that appropriate steps to carry out the process are being followed. SQA monitors processes by comparing the actual steps carried out with those in the documented procedures. The Assurance section of the Management Plan specifies the methods to be used by the SQA process monitoring activity.

A fundamental SQA technique is the audit, which looks at a process and/or a product in depth, comparing them to established procedures and standards. Audits are used to

review management, technical, and assurance processes to provide an indication of the quality and status of the software product.

The purpose of an SQA audit is to assure that proper control procedures are being followed, that required documentation is maintained, and that the developer's status reports accurately reflect the status of the activity. The SQA product is an audit report to management consisting of findings and recommendations to bring the development into conformance with standards and/or procedures.

3.3. SQA Relationships to Other Assurance Activities

Some of the more important relationships of SQA to other management and assurance activities are described below.

3.3.1 Configuration Management Monitoring

SQA assures that software Configuration Management (CM) activities are performed in accordance with the CM plans, standards, and procedures. SQA reviews the CM plans for compliance with software CM policies and requirements and

provides follow-up for nonconformances. SQA audits the CM functions for adherence to standards and procedures and prepares reports of its findings.

The CM activities monitored and audited by SQA include baseline control, configuration identification, configuration control, configuration status accounting, and configuration authentication. SQA also monitors and audits the software library. SQA assures that:

Baselines are established and consistently maintained for use in subsequent baseline development and control.

Software configuration identification is consistent and accurate with respect to the numbering or naming of computer programs, software modules, software units, and associated software documents.

Configuration control is maintained such that the software configuration used in critical phases of testing, acceptance, and delivery is compatible with the associated documentation.

Configuration status accounting is performed accurately including the recording and reporting of data reflecting the software's configuration identification, proposed changes to the configuration identification, and the implementation status of approved changes.

Software configuration authentication is established by a series of configuration reviews and audits that exhibit the performance required by the software requirements specification and the configuration of the software is accurately reflected in the software design documents.

Software development libraries provide for proper handling of software code, documentation, media, and related data in their various forms and versions from the time of their initial approval or acceptance until they have been incorporated into the final media.

Approved changes to baselined software are made properly and consistently in all products, and no unauthorized changes are made.

3.3.2 Verification and Validation Monitoring

SQA assures Verification and Validation (V&V) activities by monitoring technical reviews, inspections, and walkthroughs.The SQA role in formal testing is described in the next section. The SQA role in reviews,inspections, and walkthroughs is to observe, participate as needed, and verify that they were properly conducted and documented. SQA also ensures that any actions required are assigned, documented, scheduled, and updated.Formal software reviews should be conducted at the end of each phase of the life cycle to identify problems and determine whether the interim product meets all applicable requirements. Examples of formal reviews are the Preliminary Design Review (PDR), Critical Design Review (CDR), and Test Readiness Review (TRR). A review looks at the overall picture of the product being developed to see if it satisfies its requirements. Reviews are part of the development process, designed to provide a ready/not-ready decision to begin the next phase. In formal reviews, actual work done is compared with established standards. SQA's main objective in reviews is to assure that the Management and Development Plans have been followed, and that the product is ready to proceed with the next phase of development. Although the decision to proceed is a management decision, SQA is responsible for advising management and participating in the decision. An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis to find errors. For inspections and walkthroughs, SQA assures, at a minimum that the process is properly completed and that needed follow-up is done. The inspection process may be used to measure compliance to standards.

3.3.3 Formal Test Monitoring

SQA assures that formal software testing, such as Acceptance testing, is done in accordance with plans and procedures. SQA reviews testing documentation for completeness and adherence to standards. The documentation review includes test plans,test specifications, test procedures, and test reports. SQA monitors testing and provides follow-up on nonconformances. By test monitoring, SQA assures software completeness and readiness for delivery. The objectives of SQA in monitoring formal software testing are to assure that:

The test procedures are testing the software requirements in accordance with test plans.

The test procedures are verifiable.

The correct or "advertised" version of the software is being tested (by SQA monitoring of the CM activity).

The test procedures are followed.

Nonconformances occurring during testing (that is, any incident not expected in the test procedures) are noted and recorded.

Test reports are accurate and complete.

Regression testing is conducted to assure nonconformances have been corrected.

Resolution of all nonconformances takes place prior to delivery.

Software testing verifies that the software meets its requirements. The quality of testing is assured by verifying that project requirements are satisfied and that the testing process is in accordance with the test plans and procedures.

3.4 Software Quality Assurance during the Software Acquisition Life Cycle

In addition to the general activities described in subsections C and D, there are phase-specific SQA activities that should be conducted during the Software Acquisition

Life Cycle. At the conclusion of each phase, SQA concurrence is a key element in the management decision to initiate the following life cycle phase. Suggested activities for each phase are described below.

3.4.1 Software Concept and Initiation Phase

SQA should be involved in both writing and reviewing the Management Plan in order to assure that the processes, procedures, and standards identified in the plan are appropriate, clear, specific, and auditable. During this phase, SQA also provides the QA section of the Management Plan.

3.4.2 Software Requirements Phase

During the software requirements phase, SQA assures that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements.

3.4.3 Software Architectural (Preliminary) Design Phase

SQA activities during the architectural (preliminary) design phase include:

Assuring adherence to approved design standards as designated in the Management Plan.

Assuring all software requirements are allocated to software components.

Assuring that a testing verification matrix exists and is kept up to date.

Assuring the Interface Control Documents are in agreement with the standard in form and content.

Reviewing PDR documentation and assuring that all action items are resolved.

Assuring the approved design is placed under configuration management.

3.4.4 Software Detailed Design Phase

SQA activities during the detailed design phase include:

Assuring that approved design standards are followed.

Assuring that allocated modules are included in the detailed design.

Assuring that results of design inspections are included in the design.

Reviewing CDR documentation and assuring that all action items are resolved.

3.4.5 Software Implementation Phase

SQA activities during the implementation phase include the audit of:

Results of coding and design activities including the schedule contained in the Software Development Plan.

Status of all deliverable items.

Configuration management activities and the software development library.

Nonconformance reporting and corrective action system.

3.4.6 Software Integration and Test Phase

SQA activities during the integration and test phase include:

Assuring readiness for testing of all deliverable items.

Assuring that all tests are run according to test plans and procedures and that any non-conformances are reported and resolved.

Assuring that test reports are complete and correct.

Certifying that testing is complete and software and documentation are ready for delivery.

Participating in the Test Readiness Review and assuring all action items are completed.

3.4.7 Software Acceptance and Delivery Phase

As a minimum, SQA activities during the software acceptance and delivery phase include assuring the performance of a final configuration audit to demonstrate that all deliverable items are ready for delivery.

3.4.8 Software Sustaining Engineering and Operations Phase

During this phase, there will be mini-development cycles to enhance or correct the software. During these development cycles, SQA conducts the appropriate phase-specific activities described above.

3.4.9 Techniques and Tools

SQA should evaluate its needs for assurance tools versus those available off-the-shelf for applicability to the specific project, and must develop the others it requires. Useful tools might include audit and inspection checklists and automatic code standards analyzers.

Activity J Discuss Software Quality Assurance during the Software Acquisition Life Cycle

4.0 Conclusion

There is the need to ensure Software quality and adherence to software product standards, processes, and procedures and this is what Software Quality Assurance is out to achieve.

5.0 Summary

In this unit, you have learnt that:

Software Quality Assurance (SQA) is a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures.

Standards are the established yardsticks to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared.

Product evaluation and process monitoring are the SQA activities that assure the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and

6.0 Tutor-Marked Assignment

What is Software Quality Assurance?
Explain the concept of standards and procedures.
Discuss Software Quality Assurance Activities

7.0 Further Reading And Other Resources

Pyzdek, T, "Quality Engineering Handbook", 2003, ISBN 0-8247-4614-7

Godfrey, A. B., "Juran's Quality Handbook", 1999, ISBN 0-07-034003-X

http://www.nrc.gov/reading-rm/doc-collections/cfr/part050/part050-appb.html

Unit 4 Compatibility

1.0 Introduction

In the last unit, we considered Software Quality Assurance (**SQA). We saw the essence of** Software Quality Assurance **to ensure** that **the** software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and standards are followed at testing phase of software development. In this unit, we shall look at Compatibility testing. After studying the unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to:

Define Compatibility Testing

Explain Usefulness of Compatibility Testing.

3.0 **What is Compatibility Testing?**

Software testing comes in different types. Compatibility testing is one of the several types of software testing which can be carried out on a system that is develop based on certain yardsticks and which has to perform definite functionality in an already existing setup/environment. Many things are decided n compatibility of a system/application being developed with, for example, other systems/applications, OS, Network. They include the use of the system/application in that environment, demand of the system/application etc. On many occasions, the reason while users prefer not to go for an application/system cannot be unconnected with it non-compatibility of such application/system with any other system/application, network, hardware or OS they are already using. This explains the reason why the efforts of developers may appear to be in vain. Compatibility testing can also be used to certify compatibility of the system/application/website built with various other objects such as other web browsers, hardware platforms, users, operating systems etc. It helps to find out how well a system performs in a particular environment such as hardware, network; operating system etc. Compatibility testing can be performed manually or with automation tools.

3.1 **Compatibility testing computing environment.**

. Computing environment that will require compatibly testing may include some or all of the below mentioned elements:

Computing capacity of Hardware Platform (IBM 360, HP 9000, etc.)..
Bandwidth handling capacity of networking hardware
Compatibility of peripherals (Printer, DVD drive, etc.)
Operating systems (MVS, UNIX, Windows, etc.)
Database (Oracle, Sybase, DB2, etc.)
Other System Software (Web server, networking/ messaging tool, etc.)
Browser compatibility (Firefox, Netscape, Internet Explorer, Safari, etc.)

Browser compatibility testing which can also be referred to as user experience testing requires that the web applications are tested on different web browsers, to ensure the following:

Users have the same visual experience irrespective of the browsers through which they view the web application.
In terms of functionality, the application must behave and respond the same way across different browsers.

Compatibility between versions: This has to do with testing of the performance of system/application in connection with its own predecessor/successor versions. This is sometimes referred to as backward and forward compatibility. For example, Windows 98 was developed with backward compatibility for Windows 95.

Software Compatibility testing: This is the evaluation of the performance of system/application in connection with other software. For example: Software compatibility with operating tools for network, web servers, messaging tools etc.

Operating System compatibility testing: This is the evaluation of the performance of system/application in connection with the underlying operating system on which it will be used.

Databases compatibility testing: Many applications/systems operate on databases. Database compatibility testing is used to evaluate an application/system's performance in connection to the database it will interact with.

3.3 Usefulness of Compatibility Testing

Compatibility testing can help developers understand the yardsticks that their system/application needs to reach and fulfil, so as to get acceptance by intended users who are already using some OS, network, software and hardware etc. It also helps the users to find out which system will better fit in the existing setup they are using.

3.4 Certification testing falls within the range of Compatibility testing. Product Vendors do run the complete suite of testing on the newer computing environment to get their application certified for a specific Operating Systems or Databases.

Activity K What is Browser compatibility testing

4.0 Conclusion

Compatibility testing is highly beneficial to software development. It can help developers understand the criteria that their system/application needs to attain and fulfil, in order to get accepted by intended users who are already using some OS, network, software and hardware etc. It also helps the users to find out which system will better fit in the existing setup they are using.

5.0 Summary

In this unit, we have learnt that:

Compatibility testing is one of the several types of software testing performed on a system that is built based on certain criteria and which has to perform specific functionality in an already existing setup/environment.
Compatibility testing can be automated using automation tools or can be performed manually and is a part of non-functional software testing.
Computing environment may contain some or all of the below mentioned elements:

- o Computing capacity of Hardware Platform (IBM 360, HP 9000, etc.)..
- o Bandwidth handling capacity of networking hardware o Compatibility of peripherals (Printer, DVD drive, etc.) o Operating systems (MVS, UNIX, Windows, etc.)
- o Database (Oracle, Sybase, DB2, etc.)
- o Other System Software (Web server, networking/ messaging tool, etc.)
- o Browser compatibility (Firefox, Netscape, Internet Explorer, Safari, etc.)

The most important use of the compatibility testing is to ensure its performance in a computing environment in which it is supposed to operate. This helps in figuring out necessary changes/modifications/additions required to make the system/application compatible with the computing environment.

6.0 Tutor-Marked Assignment

Define Compatibility Testing
Explain Usefulness of Compatibility Testing.
What are the elements in computing enviroment?

7.0 Further Reading And Other Resources

E. Anderson , Z. Bai , J. Dongarra , A. Greenbaum , A. McKenney , J. Du Croz , S. Hammerling , J. Demmel , C. Bischof , D. Sorensen, LAPACK: a portable linear algebra library for high-performance computers, Proceedings of the 1990 conference on Supercomputing, p.2-11, October 1990, New York, New York, United States

S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 -- Revision 2.3.2, Argonne National Laboratory, Sep. 2006.

Myra B. Cohen , Matthew B. Dwyer , Jiangfan Shi, Coverage and adequacy in software product line testing, Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis, p.53-63, July 17-20, 2006, Portland, Maine [doi>10.1145/1147249.1147257]

Unit 5 Software verification and validation

1.0 Introduction

In the last unit, we considered Compatibility testing .You will recall that, Compatibility testing is highly beneficial to software development. It can help developers understand the criteria that their system/application needs to attain and fulfil, in order to get accepted by intended users who are already using some OS, network, software and hardware etc. It also helps the users to find out which system will better fit in the existing setup they are using. In this unit we are going to look at software verification and validation. After studying the unit you are expected to have achieved the following objectives listed below.

2.0 Objectives

By the end of this unit, you will be able to: Define
software verification and validation Outline the
method of verification and validation
Discuss Software Verification & Validation Model
Discuss terms used in validation process

3.0 What is **Verification and Validation** (**V&V**)

Verification and Validation (**V&V**) is the process of checking that a software system meets specifications and that it fulfils its expected purpose. It is normally part of the software testing process of a project.

According to the Capability Maturity Model (CMMI-SW v1.1),

> Verification is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-STD-610].
> Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. [IEEE-STD-610]

In other words, validation ensures that the product actually meets the user's needs, and that the specifications were correct in the first place, while verification is ensuring that the product has been built according to the requirements and design specifications. Validation ensures that ‘you built the right thing’. Verification ensures that ‘you built it right’. Validation confirms that the product, as provided, will fulfill its intended use.

Looking at it from arena of modeling and simulation, the definitions of validation, verification and accreditation are similar:

Validation is the process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s).

Accreditation is the formal certification that a model or simulation is acceptable to be used for a specific purpose.
Verification is the process of determining that a computer model, simulation, or federation of models and simulations implementations and their associated data accurately represents the developer's conceptual description and specifications.

3.1 Classification of methods

In mission-critical systems where flawless performance is absolutely necessary, formal methods can be used to ensure the correct operation of a system. However, often for non-mission-critical systems, formal methods prove to be very costly and an alternative method of V&V must be sought out. In this case, syntactic methods are often used.

3.2 Test cases

A test case is a tool used in the Verification and Validation process.

The Quality Assurance (QA) team prepares test cases for verification and these help to determine if the process that was followed to develop the final product is right.

The Quality Certificate (QC) team uses a test case for validation and this will ascertain if the product is built according to the requirements of the user. Other methods, such as reviews, provide for validation in Software Development Life Cycle provided it is used early for validation.

3.3 Independent Verification and Validation

Verification and validation often is carried out by a separate group from the development team; in this case, the process is called "**Independent Verification and Validation**", or **IV&V**.

3.4 Regulatory environment

The task is must to meet the compliance requirements of law regulated industries, which is often guided by government agencies or industrial administrative authorities. FDA even demands to validate software versions and patches.

3.5 Software Verification & Validation Model

'Verification & Validation Model' is used in improvement of software project development life cycle

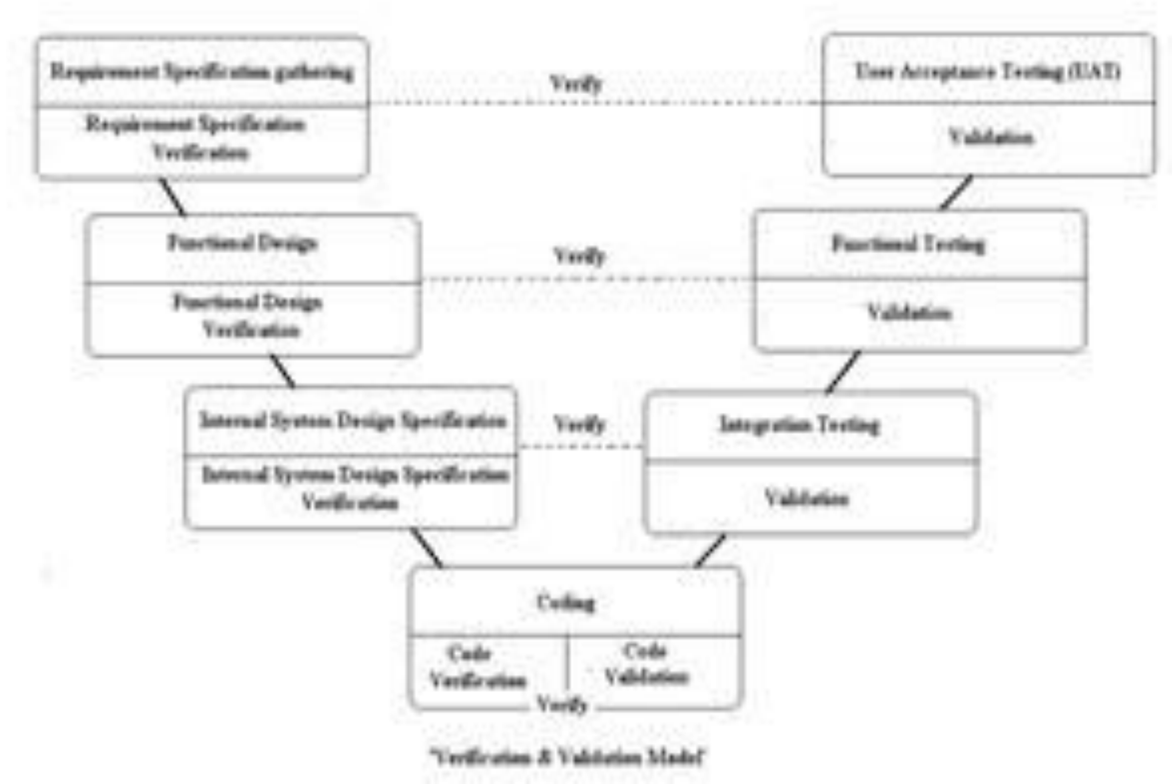

Requirement Specification gathering
Requirement Specification Verification
Verify
User Acceptance Testing (UAT)
Validation
Functional Design
Functional Design Verification
Verify
Functional Testing
Validation
Internal System Design Specification
Internal System Design Specification Verification
Verify
Integration Testing
Validation
Coding
Code Verification
Code Validation
Verify
"Verification & Validation Model"

Fig 8 Verification and Validation Model

Source: http://www.buzzle.com/editorials/4-5-2005-68117.asp

A perfect software product is developed when every step is taken in right direction. That is to say that "A right product is developed in a right manner". Software Verification Model helps to achieve this and also help to improve the quality of the software product.

The model will not only will not only makes sure that certain rules are followed at the time of development of a software but will also ensure that the product that is developed fulfils the required specifications. The result is that risk associated with any software project up to certain level is reduced by helping in detection and correction of errors and mistakes, which are unknowingly done during the development process.

3.6 Few terms involved in Verification:

3.61. Inspection:

Inspection involves a team of few people usually about 3-6 people. It usually led by a leader, which properly reviews the documents and work product during various phases of the product development life cycle. The product, as well as related documents is presented to the team, the members of which carry different interpretations of the presentation. The bugs that are discovered during the inspection are conveyed to the next level in order to take care of them.

3.6.2 Walkthroughs:

In walkthrough inspection is carried out without formal preparation (of any presentation or documentations). During the walkthrough, the presenter/author introduces the material to all the participants in order to make them familiar with it. Though walkthroughs can help in finding bugs, they are used for knowledge sharing or communication purpose.

3.6.3 Buddy Checks:

Buddy Checks does not involve a team rather, one person goes through the documents prepared by another person in order. to find out bugs which the author couldn’t find previously.

The activities involved in Verification process are: Requirement Specification verification, Functional design verification, internal/system design verification and code verification Each activity ascertains that the product is developed correctly and every requirement, every specification, design code etc. is verified.

3.7 Terms used in Validation process:

3.7.1 Code Validation/Testing:

Unit Code Validation or Unit Testing is a type of testing, which the developers conduct in order to find out any bug in the code unit/module developed by them. Code testing other than Unit Testing can be done by testers or developers.

3.7.2 Integration Validation/Testing:

Integration testing is conducted in order to find out if different (two or more) units/modules match properly. This test helps in finding out if there is any defect in the interface between different modules.

3.7.3 Functional Validation/Testing:

This type of testing is meant to find out if the system meets the functional requirements. In this type of testing, the system is validated for its functional behavior. Functional testing does not deal with internal coding of the project, in stead, it checks if the system behaves as per the expectations.

3.7.4 User Acceptance Testing or System Validation:

In this type of testing, the developed product is handed over to the user/paid testers in order to test it in real time state. The product is validated to find out if it works according to the system specifications and satisfies all the user requirements. As the user/paid testers use the software, it may happen that bugs that are yet undiscovered, come up, which are communicated to the developers to be fixed. This helps in improvement of the final product.

Activity L Discuss Software Verification & Validation Model

4.0 Conclusion

The importance of Verification and Validation cannot be overemphasisd. **Verification and Validation** (**V&V**) checks that a software system meets specifications and that it fulfils its intended purpose.

5.0 Summary

In this unit, we have learnt that:

Verification and Validation (**V&V**) is the process of checking that a software system meets specifications and that it fulfils its intended purpose.
In mission-critical systems where flawless performance is absolutely necessary, formal methods can be used to ensure the correct operation of a system. However, often for non-mission-critical systems, formal methods prove to be very costly and an alternative method of V&V must be sought out. In this case, syntactic methods are often used.
Verification and validation often is carried out by a separate group from the development team; in this case, the process is called "**Independent Verification and Validation**

6.0 Tutor-Marked Assignment

Define software verification and validation
Outline the method of verification and validation
Discuss terms used in validation process

7.0 Further Reading And Other Resources

Department of Defense Documentation of Verification, Validation & Accreditation (VV&A) for Models and Simulations, Missile Defense Agency, 2008

General Principles of Software validation; Final Guidance for Industry and FDA Staff" (PDF). FoodandDrugAdministration. 11 January 2002. http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085371.pdf. Retrieved 12 July 2009.

Guidance for Industry: Part 11, Electronic Records; Electronic Signatures — Scope and Application" (PDF). FoodandDrugAdministration. August 2003. http://www.fda.gov/downloads/Drugs/GuidanceComplianceRegulatoryInformation/Guidances/UCM072322.pdf. Retrieved 12 July 2009.

MODULE 5: FORMAL METHODS

This module is divided into five (5) units
Unit 1: Introduction to Formal Methods
Unit 2: Proposition
Unit 3: Predicates
Unit 4: Sets
Unit 5: Series or Sequence

Unit 1: Introduction to Formal Methods

1.0 Introduction

When creating a software there are few engineering stages that is normally followed to ensure that the software is built within time and budget. These stages collectively are called the software development life cycle (SDLC). The SDLC can be divided into seven (7) stages;
Initial Study, Analysis, Design, Development, Testing, Implementation and Implementation. To develop a high-quality software, where the number of bugs is greatly reduced, formal method comes into play. Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc manner.
A method is *formal* if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and, more relevant, specification, implementation, and correctness.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Give a background of formal methods
- Define the phrase formal methods
- State some advantages and disadvantages of formal methods
- Enumerate the stages of formal methods
- Enumerate the stages of SDLC
- Briefly describe each of the stage of SDLC

3.0 Main Content

3.1 Background

When a new system is to be implemented, the first step is to write a requirement specification (usually in natural language). The specification should correctly describe the system's desired behaviour and it should be complete and unambiguous, which can be hard to achieve. The specification is then transformed into code by a programmer, who has to understand the specification correctly and handle any ambiguities. Also, the programmer's way of coding and solving technical challenges can introduce faults in the code. Then there is the sheer size of the system; nowadays systems are so big that it can be hard to keep track of all the parts to make sure that they correctly follow the specification. Furthermore, there is often a team of programmers working together, which also is a source of faults since they will all have their own interpretations of the specification and of the information shared during the development process.

During and after the coding of the system, the system's functionality is usually tested to make sure that the resulting program satisfies the requirements and that no errors or bugs are present. Testing big and complex systems can be very time consuming and, due to the size of the system and the amount of code, an exhaustive testing is not practically feasible.Nevertheless, when the system is safety and security critical, correct functionality has to be guaranteed, which requires either exhaustive testing or a way of proving that the code correctly implements the specification.

The concept of formal methods introduces tools to mathematically describe a system (or parts of a system) in a specification and to prove that the resulting program meets the requirements described in the specification. A formal specification is precise and there is no risk for misinterpretations. Also, if there is a proof that the implementation abides by the specification, then one can be sure that the programmers have implemented what is described in the specification. In practice, one cannot completely guarantee that the resulting implementation is fault free, since the formal method used can have defects, or there might be some error in the proof. Nevertheless, increased use of formal methods and tools will result in better and more reliable methods and tools. To summarise: by using formal methods in the system development, errors can be found earlier and some classes of errors can be nearly eliminated.

A limitation of formal methods is that they only can be used to prove a system's correctness with respect to a specification. Therefore, just because a program has been mathematically proved to abide to the specification, there is no guarantee that the specification in itself is correct and fault free. Nevertheless, properties can be proved on the specification to strengthen the belief that the specification correctly represents the desired functionality.

A brief description of the stages of Software Development Life (SDLC) will enhance the understanding application of formal methods in software development.

When creating a software there are few engineering stages that is normally be followed to ensure that they software is built within the time and budget. These stages collectively are called the software development life cycle (SDLC).

The SDLC can be divided into seven (7) stages;

1. Initial Study: This is the first time the system development team meets the clients to collective information regarding the problem. Normally this stage delivers the proposal and quotation to the clients.
2. Analysis: After the client has agreed to the proposal and price, the team will go in and study the current system with the intention to discover the source of the problem. The System analyst will use diagrams and data collection techniques (observation, inspections, interview, etc) to aid them. Normally this stage delivers a report stating the source of the problem and more then one alternative solutions.
3. Design: After the client agrees with the analysis findings, the client will choose one (1) solution. From this one solution the system designer will create the specification. Take note that different IT section will require different specification. For the software section, the deliverables will take the form of a screen design, logic design, representation of the codes, etc.
4. Development: Based on the given specification, the respective IT section will develop the solution. For the software section, the deliverables will be a full running software program created from the specification.
5. Testing: The test documents (Test Plan and Test Case) are normally created by the System Analyst during the development stages. The tester (normally a 3rd party) will use the Test Plan and Test Case to complete the testing. The deliverables will be a letter from the tester stating the outcome of the test.
6. Implementation: At this stage onward the software is no longer a concern, the main objective now will be to prepare the environment. The implementation plan will list the tasks necessary to prepare the environment to accept new software, such as installation, training, conversion of data, change over method, etc. There are many deliverables here depending on what is listed in the implantation plan. For example, for user training a user manual is normally created.

7. Review: This is the final stage where the software user and team will sit down to review the software performance and to decide negotiate on the maintenance contract. If all goes well normally but not necessary a sign off letter will be the last deliverables.

3.2 Formal Method

Formal method is a way to takes the specification (written in natural language) and converts it into its mathematical equivalent. Thus, it is normally used in the SDLC Analysis and Design stages. The natural language usually contains ambiguous, incomplete and inconsistent statement.
Once a specification in English for example is translated to a mathematical form, it will remove all ambiguity and uncertainty in that statement.
Formal method will also bring to light all different probable perspective to any given variables and functions that could have been hidden behind the English language.
This can be done using a number of formal languages such as Z notation, VDM, Algebra, Functional Programming, etc.
Creating software need not use formal method, having said that, having formal method imbedded into the SDLC does give the software huge advantages and also a new set of disadvantages

In computer science, formal methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. Mathematically rigorous means that the specification consists of well-formed statements using mathematical logic and that a formal verification consists of rigorous deductions in that logic. The strength of formal methods is that they allow for a complete verification of the entire state space of the system and that the properties that can be proved to hold in the system will hold for all possible inputs. When formal methods cannot be used through the entire development process (due to the complexity of the system, lack of tool or other reasons), they can still successfully be used on parts of the system, for example for the requirements and high-level design or only on the most safety or security critical components.

The diversity of available formal methods is a result of the different modelling methods and proof approaches needed by different application domains. Also, different development phases of a system might require different tools and techniques.

Although many developed formal methods are the result of research efforts in universities, more and more tools and techniques are available outside the academic community. Several of the standards used for system development require formal methods at the highest levels of accreditation.
Some examples are the Common Criteria standard and the DO-178C standard Software Considerations in Airborne Systems and Equipment Certification for software for airborne systems in commercial aircraft.

3.3 Advantages

Some of the (plausible) advantages of the use of formal methods for software development are as follows.

- The development of a formal specification provides insights into and an understanding of the software requirements and software design. This reduces requirements errors and omissions. It provides a basis for an elegant software design.

  Indeed, it is sometimes helpful to develop a formal specification of *an existing system* if that system is complex, and it is to be changed or replaced, since this can detect subtle errors that would otherwise be included in the modified (or new) system. In some editions of his book, Pressman mentions a case involving the development of an operating system. Here at Calgary, this technique has been used to detect errors in VLSI chips before the chips have been fabricated.
- Formal software specifications are mathematical entities and may be analysed using mathematical methods. In particular, it may be possible to *prove* specification consistency and completeness. It may also be possible to prove that an implementation conforms to its specification. The absence of certain classes of errors may be demonstrated. However, program verification is expensive and the ability to reason about the specification itself is probably more significant.
- Formal specifications may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging. Sommerville discusses the possibility of ``animating'' formal specifications in order to produce prototypes of systems.
- Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases. For example, a function's *preconditions* and *postconditions* can be used to design (black box) tests, and a *class invariant*

can be useful when testing a class in an object-oriented system. These can all be written explicitly in (or deduced from) formal specifications.

NOTE: Formal Method forces the System Analyst and Designer to think carefully about the specification as it enforces proper engineering approach using discrete mathematics.
Formal Method forces the System Analyst and Designer to see all the different possible states for any given variables and functions thus will avoid many faults and therefore reduces the bugs and errors from the design stage onward.

3.4 Disadvantages

Disadvantages include the fact that these methods aren't always appropriate (there are some kinds of requirements that really are more easily, and accurately, specified using pictures with annotations), and involve the difficulty of adopting such methods in industry.

- Software management is often conservative and is unwilling to adopt new techniques for which payoff is not obvious. It is difficult to demonstrate that the relatively high cost of developing a formal system specification will reduce overall software development costs. In this respect, the use of mathematics in software engineering is different from in other engineering disciplines: Mathematical analysis of physical structures can result in cost savings in materials and allows cheaper designs to be used.
- Some software engineers, particularly those in senior positions, have not been trained in the techniques needed to develop formal software specifications. Developing specifications requires a familiarity with discrete mathematics and logic. Inexperience with these techniques makes the development of formal specifications more difficult than it would be otherwise.
- System customers are unlikely to be familiar with formal specification techniques. They may be unwilling to fund development activities that they cannot easily monitor.
- Some classes of software system requirements are difficult to specify using existing techniques. In particular, current techniques cannot be used to specify the interactive components of user interfaces. Some classes of parallel processing systems, such as interrupt-driven systems, are difficult to specify.

- There is a widespread ignorance of the practicality of current specification techniques and their applicability. The techniques have been used successfully in a significant number of nontrivial development projects.
- Most of the effort in specification research has been concerned with the specification of languages and their theoretical underpinnings. Relatively little effort has been devoted to method and tool support.

NOTE: Formal Method requires the person to know how to apply discrete mathematics. It will obviously slow down the analysis and design stage resources and time therefore also the cost of the project.
There are too many different formal methods and most of them are not compatible with each other.
Formal methods do not guarantee that a specification is complete. For each variable and function, it just forces the System Analyst and Designer to view the specification from a different perspective but it does not guarantee that variable and functions will not be left out.

3.5 Critical Software

Having known the advantages and disadvantages, most clients will see the justification to use formal methods for critical systems, but this thinking is now slowly fading as most clients realize the important and cost saving and convenience of having a good specification initially in the SDLC.
There are basically three (3) different types of critical systems;

1. Business Critical System

Business Critical System refers to a system where the honesty and integrity of the business is paramount. All data kept in the system must be accurate at all times. If a fault is found the entire process must be stop to allow correction. Most government, business and manufacturing company that requires payment are business critical.

2. Mission Critical System

Mission Critical System refers to a system where the continuous running of the system is paramount. Accurate takes a lower priority compare to the running of the system. Auto Teller Machine, Car ticketing system, Alarm Systems are mission critical.

3. Safety Critical System

Safety Critical System refers to a system where the safety of everyone directly or indirectly affected by the system is paramount. Functionality and Accurate takes a

lower priority compare to the safety of the users. Most medical, construction and oil rig systems are safety critical system.
Many organizations today require a combination of the above as such you may have a business mission critical system, a business safety critical system, etc.

3.6 Integrity Level

Integrity level refers to how much cost an organization is willing to spend and how much risk is an organization is willing to take when developing software.

Integrity Level	Cost	Risk	Example of System
1	Low	Low	Address Book System
2	Low	High	Global Tsunami Warning System
	High	Low	Waste Water System
3	High	High	Nuclear Reactor System

3.7 Stages in Formal Methods

1. Formal Specification

This is where normal system specification is use and translated using a formal language into a formal specification. There are basically two type of formal language; Model Oriented (VDM, Z, etc) and Properties Oriented (Algebraic Logic, Temporal Logic, etc). This is the cheapest way to handle formal method.

The formal specification generally does the following process.

- Get user requirement usually from the specification written in the natural language.
- Clarify the requirement using mathematical approach. This is to remove all ambiguous, incomplete and inconsistent statement.
- After statements are clearly identified. Then find all assumptions (Things that must be in place before something can happen) that is state or not stated within the clarified requirement.
- Then expose every possible logic defect (fault) or omission in the clarified requirement.
- Identify what are the exceptions (bad things) that will arise if the defects are not corrected.

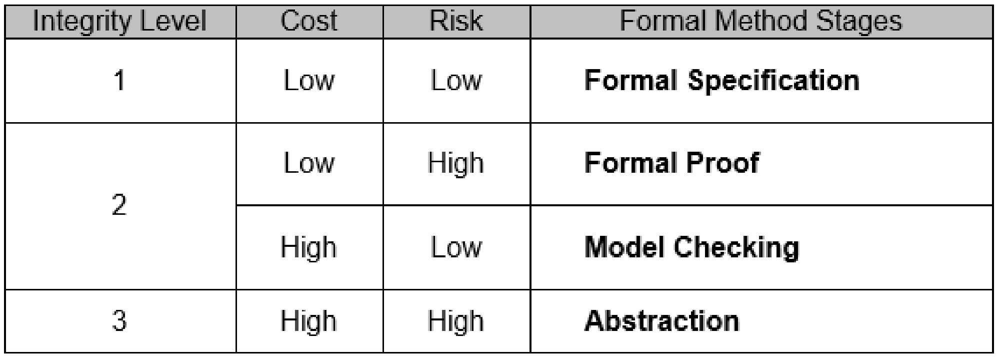
- Find a way to test for all the possible each exception. Only when you can test for an exception can you be able to stop that exception from happening.

2. Formal Proof
This level studies the formal specification and retrieves the goals of the formal specific. Then fixed rules are created and with these rules step by step instructions are listed to achieve the specified goals. This is relatively cheaper but there are more task steps.

3. Model Checking
This level studies the formal specification and formal proof deliverables to make sure that the system or software contains ALL possible properties to be able to handle all possible scenarios that could happen for a given specification. This stage is beginning to be more expensive.

4. Abstraction
This level uses mathematical and physical models to create a prototype of the entire system for simulation. This prototype is use to focus on the properties and characteristic of the system. This is the most expensive formal method.

Integrity Level and Formal Method Stages
The integrity level decided by the organization will determine how deep to go into the Formal Method stage.
Remember that the deeper into the formal method means more time and resources thus more cost will be incurred.



Integrity Level	Cost	Risk	Formal Method Stages
1	Low	Low	**Formal Specification**
2	Low	High	**Formal Proof**
	High	Low	**Model Checking**
3	High	High	**Abstraction**

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Briefly discuss the background of formal methods
2. Enumerate the stages of formal methods
3. Briefly describe each of the stage of SDLC
4. Define the phrase formal methods
5. State some advantages and disadvantages of formal methods

5.0 Conclusion

Formal methods can be applied at different stage of the SDLC. It can be applied at any stage but the earlier it is applied the better. It is very useful, most especially at the specification and design stages. Its advantages overweigh its disadvantages.

6.0 Summary

Formal Method forces the System Analyst and Designer to think carefully about the specification as it enforces proper engineering approach using discrete mathematics. In this unit we discussed the following:

Background
Formal Method
Advantages
Disadvantages
Critical Software
Integrity Level
Stages in Formal Methods

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 2: Proposition

1.0 Introduction

Proposition is a declarative statement that can result in either true or false. The statement must be a constant thus the value cannot change.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Define proposition
- Identify proposition operators
- Construct and interpret propositions
- Construct truth tables

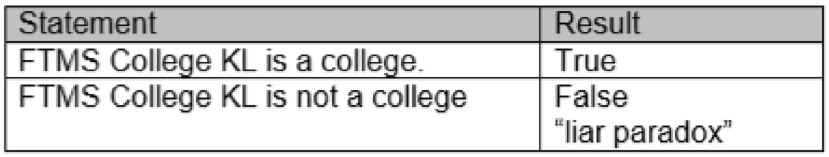
3.0 Main Content

3.1 Introduction to Proposition

Proposition is a declarative statement that can result in either true or false. The statement must be a constant thus the value cannot change.

In formal methods, the natural language is scan for propositions. Each proposition (either or false) will be translated into an expression usually joined using operators, and all the possible value for each proposition will be listed in a truth table to cover all possible value.

For example:



Statement	Result
FTMS College KL is a college.	True
FTMS College KL is not a college	False “liar paradox”

If two statements have the same meaning, that statement or proposition is considered to the equal even if they are spoken in different format or language.

When most people say ‘logic’, they mean either propositional logic or first-order predicate logic. However, the precise definition is quite broad, and literally hundreds of logics have been studied by philosophers, computer scientists and mathematicians.

Any ‘formal system’ can be considered a logic if it has:

- a well-defined syntax;
- a well-defined semantics; and
- a well-defined proof-theory.

The syntax of a logic defines the syntactically acceptable objects of the language, which are properly called well-formed formulae (wff). (We shall just call them formulae.). The semantics of a logic associate each formula with a meaning. The proof theory is concerned with manipulating formulae according to certain rules.
The simplest, and most abstract logic we can study is called propositional logic.
Definition: A proposition is a statement that can be either true or false; it must be one or the other, and it cannot be both.
EXAMPLES. The following are propositions:

- the reactor is on;
- the wing-flaps are up;
- John Major is prime minister.

whereas the following are not:

- are you going out somewhere?
- 2+3

It is possible to determine whether any given statement is a proposition by prefixing it with:
It is true that ...
and seeing whether the result makes grammatical sense.
We now define atomic propositions. Intuitively, these are the set of smallest propositions.
Definition: An atomic proposition is one whose truth or falsity does not depend on the truth or falsity of any other proposition. So, all the above propositions are atomic.

Now, rather than write out propositions in full, we will abbreviate them by using propositional variables. It is standard practice to use the lower-case roman letters p,q,r,... to stand for propositions. If we do this, we must define what we mean by writing something like:
Let p be John Major is prime Minister.
Another alternative is to write something like reactor is on, so that the interpretation of the propositional variable becomes obvious.

3.2 **Connectives**

The study of atomic propositions is pretty boring. We therefore now introduce a number of connectives which will allow us to build up complex propositions.

The connectives we introduce are:

∧ and (& or .)

∨ or (|or +)

¬ not (~)

⇒ implies (⊃or→)

⇔ iff

Some books use other notations; these are given in parentheses.

3.2.1 AND Connective

Any two propositions can be combined to form a third proposition called the conjunction of the original propositions.

Definition: If p and q are arbitrary propositions, then the conjunction of p and q is written p∧q and will be true iff both p and q are true

We can summarise the operation of ∧ in a truth table. The idea of a truth table for some formula is that it describes the behaviour of a formula under all possible interpretations of the primitive propositions the are included in the formula. If there are n different atomic propositions in some formula, then there are 2n different lines in the truth table for that formula. (This is because each proposition can take one 1 of 2 values — true or false.) Let us write T for truth, and F for falsity.

Then the truthtable for p∧q is:

p	q	$p \wedge q$
F	F	F
F	T	F
T	F	F
T	T	T

3.2.1 OR Connective

Any two propositions can be combined by the word 'OR' to form a third proposition called the disjunction of the originals.

Definition: If p and q are arbitrary propositions, then the disjunction of p and q is written p∨q and will be true iff either p is true, or q is true, or both p and q are true.

The operation of∨is summarised in the following truthtable:

p	q	$p \vee q$
F	F	F
F	T	T
T	F	T
T	T	T

3.2.3 Implies Connective (i.e., If... Then...)

Many statements, particularly in mathematics, are of the form: if p is true then q is true. Another way of saying the same thing is to write:

p implies q.

In propositional logic, we have a connective that combines two propositions into a new proposition called the conditional, or implication of the originals, that attempts to capture the sense of such a statement.

Definition: If p and q are arbitrary propositions, thenthe conditional of p and q is written p⇒q and will be trueiff either p is false or q is true.

The truth table for⇒is:

p	q	$p \Rightarrow q$
F	F	T
F	T	T
T	F	F
T	T	T

The ⇒operator is the hardest to understand of the operators we have considered so far, and yet it is extremely important. If you find it difficult to understand, just remember that the p⇒q means ‘if p is true, then q is true’. If p is false, then we don’t care about q, and by default, make p⇒q evaluate to T in this case.
Terminology: if φ is the formula p⇒q, then p is the antecedent of φ and q is the consequent.

3.2.4 Iff

•Another common form of statement in maths is: p is true if, and only if, q is true. The sense of such statements is captured using the biconditional operator.

Definition: If p and q are arbitrary propositions, then the biconditional of p and q is written:

$p \Leftrightarrow q$

and will be true iff either:

1. p and q are both true; or
2. p and q are both false.

The truthtable for⇔is:

p	q	$p \Leftrightarrow q$
F	F	T
F	T	F
T	F	F
T	T	T

If p⇔q is true, then p and q are said to be logically equivalent. They will be true under exactly the same circumstances.

3.2.5 Not Connective

All of the connectives we have considered so far have been binary: they have taken two arguments. The final connective we consider here is unary. It only takes one argument. Any proposition can be prefixed by the word 'not' to form a second proposition called the negation of the original.

Definition: If p is an arbitrary proposition, then the negation of p is written

$\neg p$

and will be true iff p is false.

Truth table for¬:

p	$\neg p$
F	T
T	F

3.2.6 Comments

We can nest complex formulae as deeply as we want.

We can use parentheses i.e.,), (, to disambiguate formulae.

EXAMPLES. If p,q,r,s and t are atomic propositions, then all of the following are formulae:

$p \wedge q \Rightarrow r$

$p \wedge (q \Rightarrow r)$

$(p \wedge (q \Rightarrow r)) \vee s$

$((p \wedge (q \Rightarrow r)) \vee s) \wedge t$

whereas none of the following is:

$p \wedge$

$p \wedge q)$

$p \neg$

3.2.7 Tautologies and Consistency

Given a particular formula, can you tell if it is true or not? No, you usually need to know the truth values of the component atomic propositions in order to be able to tell whether a formula is true.

Definition: A valuation is a function which assigns a truth value to each primitive proposition.

In Modula-2, we might write:

PROCEDURE Val(p : AtomicProp): BOOLEAN;

Given a valuation, we can say for any formula whether it is true or false.

EXAMPLE. Suppose we have a valuation v, such that:

$v(p) = F$

$v(q) = T$

$v(r) = F$

Then the truth value of $(p \vee q) \Rightarrow r$ is evaluated by:

$(v(p) \vee v(q)) \Rightarrow v(r)$ (1)

$= (F \vee T) \Rightarrow F$ (2)

$= T \Rightarrow F$ (3)

$= F$ (4)

Line (3) is justified since we know that $F \vee T = T$.

Line (4) is justified since $T \Rightarrow F = F$. If you can’t see this, look at the truth tables for $\vee$and $\Rightarrow$.

When we consider formulae in terms of interpretations, it turns out that some have interesting properties.

Definition:

1. A formula is a tautology iff it is true under every valuation;
2. A formula is consistent iff it is true under at least one valuation;
3. A formula is inconsistent iff it is not made true under any valuation. Now, each line in the truth table of a formula corresponds to a valuation. So, we can use truth tables to determine whether or not formulae are tautologies.

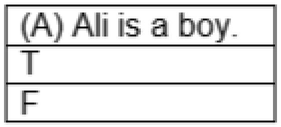
3.3 Truth Table

Truth tables is used to tell whether a propositional is true or false not only for one (1) instance but for all possible instance of the variable.

Since proposition is either true or false thus we can use a truth table to list down all the position state of that proposition.

Proposition: A = Ali is a boy.

Possible value: true or false therefore in a truth table.

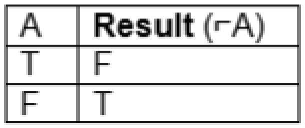


(A) Ali is a boy.
T
F

Proposition: A = Ali is NOT a boy or NOT (Ali is a boy)

Possible value: true or false therefore in a truth table.

With a NOT operator true becomes false and false become true.



A	**Result** (⌐A)
T	F
F	T

For every increase in proposition, there is a double increase in possible value.

Proposition: Ali is a boy and Mary is a girl. A = Ali is a boy M = Mary is a girl

Notice that this is actually two propositions join with the AND operator. We see it as A ^ M

Possible value: true or false for A and possible value: true or false for M

With an AND operator both statements must be true then the join statement will be true.

A	M	**Result** (A ^ M)
T	T	T
T	F	F
F	T	F
F	F	F

Once we understand the above explanation, we can use the same principal for the remaining proposition operators.

OR

A	M	**Result** (A v M)
T	T	T
T	F	T
F	T	T
F	F	F

Different

A	M	**Result** (A $\otimes$ M)
T	T	F
T	F	T
F	T	T
F	F	F

A more complex proposition

(P $\rightarrow$ Q) v (Q $\rightarrow$ P)

P	Q	P $\rightarrow$ Q	Q $\rightarrow$ P	**Result** (P $\rightarrow$ Q) v (Q $\rightarrow$ P)
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

Precedence of operation

Precedence	Symbol	Meaning	Example
1	( )		
2	¬ ~	NOT	Fail means NOT Pass
3	^	AND Conjunction	Hard work AND good attitude
4	v	OR Disjunction	Code in Java OR Code in C++
5	→	Conditional Implies	If you pass then you get reward
6	↔	Equals Bi-directional Bi-implication	Pass if and only if marks above 40
	⊗	Different Exclusive	Success is different from Failure

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. State the rules that governs the results of operation resulting from the joining of two propositions using OR, implies, and Difference
2. Illustrate tautology and consistency with an example.
3. Construct two prepositions using X, Y and Z. Determine the results using (X ^ Y) v (X ^ Z). Describe the meaning of your result(s)
4. State the importance of a truth table and connectives in prepositional logic.
5. Explain the following terms: preposition, connectives, formula disambiguation

5.0 Conclusion

Proposition is a formalism that can be used in software development to remove ambiguity in requirement specification. It useful in hunting and removing bugs and errors during software testing

A declarative sentence that is either true or false, but *not* both, is a proposition

6.0 Summary

In this unit propositional logic was introduced. Here we discussed several Connectives which include AND, OR, NOT, etc. We also illustrated proposition with the use of truth table.

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 3: Predicates

2.0 Introduction

A predicate is a relation among objects, and it consists of a condition part and an action part, IF (condition) and THEN (action). Predicates that have no conditional part are facts.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Give a background of formal methods
- State some advantages and disadvantages of formal methods
- Enumerate the stages of formal methods
- Enumerate the stages of SDLC
- Briefly describe each of the stage of SDLC

3.0 Main Content

3.1 Introduction to Predicate

In formal methods, the natural language is scan for predicates. Each functions and variables (bounded or free) will be translated into an expression also usually joined using operators. Then all possible qualifiers will be listed. Sometime a truth table is used to cover all possible value.

But this is not practical as most statement actually contains variables and changes in the variables will change the validity of the statement to true or false, because some statement refers to a set of different elements.

Therefore we use predicate to handle such statement. For this subject we will use First-Order Logic only.

To use predicate there must at least two (2) elements;

1. A variable or a constant
2. A function that will be performed on or bythe variable(s)

For example:

An ostrich has wing can fly, a eagle has wing can fly

The constant object here is “Wing”

The variable object here is either “Ostrich” or “Eagle”

The function here is “Fly”

Constant / Variable	Function	Result
Wing / Ostrich	Fly(Wing, Ostrich)	False
Wing / Eagle	Fly(Wing, Eagle)	True

Predicate quantifiers

The following are quantifiers that can be use with a predicate;

Symbol	Meaning	Example
∃	Existential	There exists some or For some the elements including itself
∀	universal	For every element For all the elements obviously including itself

Qualifier is normally place with an object (variable or constant)

Assuming a function call fly with only one (1) set of object Airplane, therefore using the above qualifier we can have two (2) different statements.

∀Airplane Fly (Airplane) = All airplane can fly

∃Airplane Fly (Airplane) = Some airplane can fly

If we have two sets of Airplane (Plane A and Plane B) using the above "Fly Faster" function we can have four (4) different statements. Plane A is denoted as A and Plane B is denoted as

B.

∀A ∀B	Fly Faster (A, B)	= All Plane A fly faster then All Plane B
∀A ∃B	Fly Faster (A, B)	= All Plane A fly faster then Some Plane B
∃A ∀B	Fly Faster (A, B)	= Some Plane A fly faster then All Plane B
∃A ∃B	Fly Faster (A, B)	= Some Plane A fly faster then Some Plane B

If we have two sets of different object Boys and Girls using the above "Run Faster" function we also have four (4) different statements. Boys is denoted as B and Girls is denoted as G.

∀B ∀G	Run Faster (B, G)	= All Boys run faster then All Girls
∀B ∃G	Run Faster (B, G)	= All Boys run faster then Some Girls
∃B ∀G	Run Faster (B, G)	= Some Boys run faster then All Girls
∃B ∃G	Run Faster (B, G)	= Some Boys run faster then Some Girls

Predicates and Operators

All the operators used with proposition can be use to join different predicates.

Predicate Example	Symbol	Meaning
⌐ Fly (Airplane)	⌐	Airplane cannot fly
Fly (Airplane) ^ Fly (Birds)	^	Airplane fly AND Bird fly
Fly (Airplane) v Fly (Birds)	v	Airplane fly OR Bird fly
Repair (Airplane) → Fly (Airplane)	→	Repair the airplane then airplane will fly.
Repair (Car) ↔ Repair(Bus)	↔	Repair the Car is the same as Repair Bus
Repair (Airplane) ⊗ Repair(Bus)	⊗	Repair the airplane is different from Repair Bus

3.1.1 Predicates and Truth Table

Because the result of a predict function can be true or false, Truth tables can also be used with predicates.

For example:

For a one function predict A = ⌐ Fly (Airplane)

A	**Result** (⌐ A)
T	F
F	T

For a two (2) function predict Fly (Airplane) ^ Fly (Birds)

A = ∀Airplane Fly (Airplane)

B = ∀Birds Fly (Birds)

A	B	**Result** (A ^ B)
T	T	T
T	F	F
F	T	F
F	F	F

3.1.2 Bound and Free (Bound) variables

This term is use in mathematics, in formal languages (mathematical logic and computer science).

A free variable is a notation that specifies places in an expression where substitution may take place.

A bound variable is a notation that specifies places in an expression no changes can take place.
The idea is related to a symbol that will later be replaced with strings or values. It can also be represented by a wildcard character that stands for an unspecified symbol.
Based on the example, below:

1. ∀x, function (x, y)
2. ∃x, function (x, y)

If symbol x in the function represents a bound variable because it is stated in the qualifier. The symbol y in the function represents a free variable because it is not stated in the qualifier. The symbol w (or any other value) is a neither bound nor free as it was never use in the function.

∀x on the left refers to an instance of x
function (x, y) on the right should also refers to an instance of x

But technically the left x and right x could mean something else but this will cause a lot of confusion, thus the symbol on the left is usually kept in consistent with the symbol on the right

4.0 Self-Assessment Exercise(s)
Answer the following questions:

1. Explain bound and free variables
2. What are predicate, operators and truth sets?
3. Illustrate a first-order logic with appropriate predicates
4. List the predicates quantifiers for each of universal and existential statements.
5. Explain Universal and Existential statements
6. Explain the different stages of SDLC

5.0 Conclusion
A predicate is a relation among objects, and it consists of a condition part and an action part, IF (condition) and THEN (action). Predicates that have no conditional part are facts.

6.0 Summary

An introduction to Predicates was discussed as well as bound and Free variables, predicates and Truth Table. Predicate quantifiers such existential and universal was highlighted'

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD 3049 Formal Methods in Software Engineering
Kuala Lumpur, Malaysia

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 4: Sets

1.0 Introduction

Set is very basic mathematical concept use to group objects. It is basically used to show the relationship between each type of objects. The Venn diagram is always used to picture the set theory graphically. A set is a group that may contain none or one (1) or more elements.

In formal methods, there are many ways to use set theory. One example will be to categories many types of objects available in a system mostly in the form of data. Base on the purpose of the particular system or software the objects are properly grouped and then related to one another.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Define a set
- Mention and illustrate the terminologies used to describe sets Relationship
- Differentiate between finite and infinite elements
- Discuss the operations on set with appropriate examples

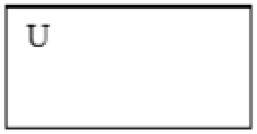
3.0 Main Content

Set is very basic mathematical concept use to group objects. It is basically used to show the relationship between each type of objects. The Venn diagram is always used to picture the set theory graphically. A set is a group that may contain none or one (1) or more elements.

In formal methods, there are many ways to use set theory. One example will be to categories many types of objects available in a system mostly in the form of data. Base on the purpose of the particular system or software the objects are properly grouped and then related to one another.

3.1 Universe (U)

The Universe represents the scope of the system. All elements that is within the universe is considered necessary and elements not mentioned in the universe is considered none existence. Thus, it is very important that we define the universe accurately.



3.3 Elements

All elements in a set must be unique. In the Venn Diagrams the individual small letter element name are prefix with a dot. Capital Letter names is use to group many duplicates elements (sets) do not have a dot. The sequence or arrangement of the elements is not important.

There are two (2) ways to describe elements in a set.

1. Using a rule or semantic description:

A is the set whose members are the first four positive integers.

B is the set of colours of the Malaysian flag.

2. Listing each member of the set or extensional definition. Elements in a list are enclosed inside curly brackets separated by commas.

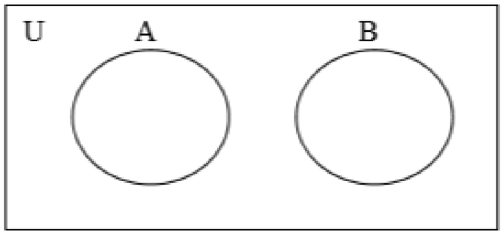
All elements in a set must be unique. In the Venn Diagrams the individual small letter element name are prefix with a dot. Capital Letter names is use to group many duplicates elements (sets) do not have a dot. The sequence or arrangement of the elements is not important.

There are two (2) ways to describe elements in a set.

1. Using a rule or semantic description:

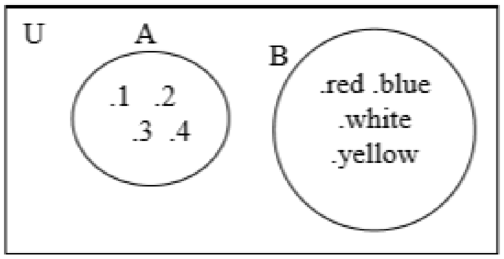
A is the set whose members are the first four positive integers.

B is the set of colors of the Malaysian flag.



2. Listing each member of the set or extensional definition. Elements in a list are enclosed inside curly brackets separated by commas.

A = {4, 2, 1, 3}
B = {blue, white, red, yellow}



3.3 Finite Elements

Some elements may be finite (with a starting and ending value) thus it can be representation as:

To show a value from 1 to 100 is represented as {1, 2, 3, ..., 100}
To show a value between 1 to 100 is represented as {2, 3, ..., 99}
F contains a number power by 2 minus 4 such that (: or |) all the numbers are integer starting from 0 to 19 is represented as
F = {n2 − 4 | n is an integer; and $0 \leq n \leq 19$}
Or
F = {n2 − 4 : n is an integer; and $0 \leq n \leq 19$}

3.4 Infinite Elements
Some elements may be infinite (no ending value) thus it can be representation as:
To show a integer value above 1 is represented as {1, 2, 3, ...}
F contains all the teachers in FTMS Global KL is represented as
F = { F | F all the teachers in FTMS Global KL }
Or
F = { F : F all the teachers in FTMS Global KL }

3.5 Cardinality
Cardinality means the number of elements in a set. Cardinality is denoted by vertical bars around the set.
For example
$|\{1, 3, 9, 15\}| = 4$
$|\{1, 2, 3, . . .\}| = \infty$

3.6 Reserve Letter used by Mathematician
P Set of all primes: P = {2, 3, 5, 7, 11, 13, 17, ...}
N Set of all natural numbers: N = {1, 2, 3, . . .}
Z Set of all integers (positive/ negative / zero): Z = {..., −2, −1, 0, 1, 2, ...}
Q Set of all rational numbers (that is, the set of all proper and improper fractions):
R Set of all real numbers (rational numbers, irrational numbers)
C Set of all complex numbers: C = {a + bi: a, b ∈ R}.
H Set of all Quaternions: For example, $1 + i + 2j - k \in H$.

3.7 Terminology used to describe sets Relationship
Membership (∈)
Membership happens when one element or a set is found inside another set. This symbol is normally used in describing a set for example

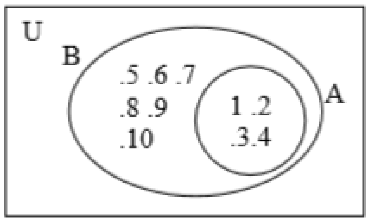
A = {A ∈ Color of the rainbow}
If A is a member of B, this is denoted A ∈ B.
If A is not a member of B then A ∉ B
A = {1, 2, 3, 4} therefore 4 ∈ A but 9 ∉ A
B = {blue, white, red} therefore “blue” ∈ B but “pink” ∉ B

Subsets (⊆)/ Supersets (⊇)

A = {4, 2, 1, 3}

B = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

A ⊆ B and A ⊂B



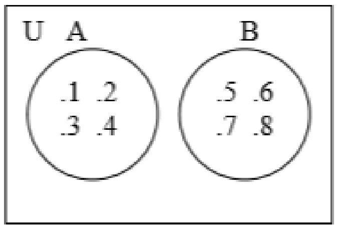
If every member of set A is found inside set B, then A is a subset of B (A ⊆ B). If set B has every member of A and more then B is a super set of A (B ⊇ A) This kind of relationship is also known as inclusion or containment.
If B is not a subset of A then we use the not a subset ⊊
If A is not a superset of B then we use the not a subset ⊋

We call “A“ a proper subset of “B” if A ⊆ B and A ≠ B
Let A = {1, 2, 3}, B = {1, 2, 3, 4}, then A ⊆ B and also A ⊂B
Let A = {1, 2, 3}, B = {3, 2, 1}, then A ⊆ B because A = B

Disjoint Sets

A = {4, 2, 1, 3}

B = {5, 6, 7, 8}



If every member of set A has no relation with set B and vice versa then we say that A disjoint B. There is no special symbol to show this relationship.

NULL Set (∅)

Every universe or set or subset contains a NULL set. A null set is an empty set ({ }) that carries no elements. We can say that the NULL set is a subset for every set.

Family Sets

There are times when a set does not contain individual elements but it contains many subsets. Conveniently this is called a family set and is it describes using the curly bracket within a curly bracket.

A = { {1, 2, 3, 4, 5} , {6, 7, 8, 9, 10} , {11, 12, 13, 14, 15} }

Power Sets (P(setName))

Remember that a set is a group that may contain none or one (1) or more elements. A power set means to show how many possible different ways to group all the elements in a set. In other words, power set is the set of all subsets of a given set.

A = {1, 2, 3}, A has 3 elements, there is 8 possible ways to arrange this 23 = 8.

P(A) = { ∅ , {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1, 2, 3} }

3.8 Terminology used to describe sets Operation

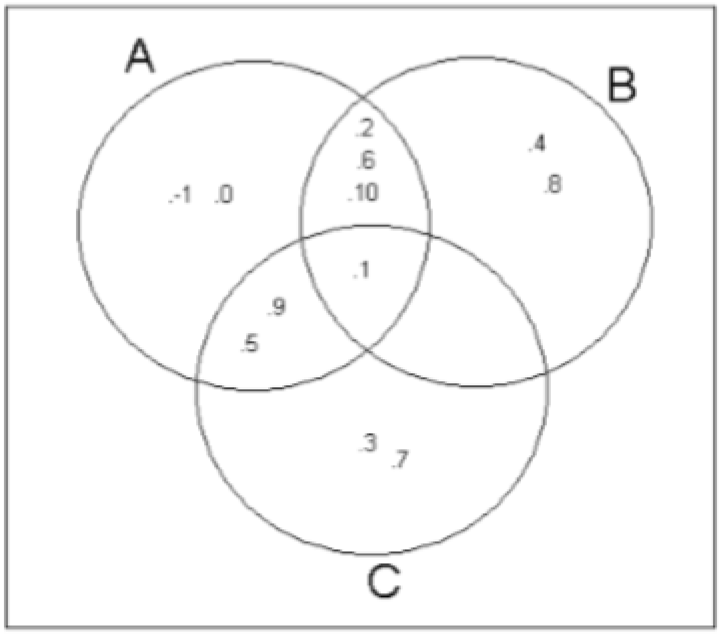
Given the following sets:

U = {-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

A = {-1, 0, 1, 2, 5, 6, 9, 10}

B = {1, 2, 4, 6, 8, 10}

C = {1, 3, 5, 7, 9}



Union (∪): Add in all elements that are found in both sets.
A ∪ B = {-1, 0, 1, 2, 4, 5, 6, 8, 9, 10}
A ∪ C = {-1, 0, 1, 2, 3, 5, 6, 7, 9, 10}
B ∪ C = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
A ∪ B ∪ C = U

Intersect (∩): Show only elements that is found only in both sets
A ∩ B = {1, 2, 6, 10}   A ∩ C = {1, 5, 9}   B ∩ C = {1}

Difference (-): Also known as subtract, this show only elements that is found in this set but NOT found in another sets
A - B = {-1, 0, 5, 9}   A - C = {-1, 0, 2, 6, 10}   B ∩ C = {2, 4, 6, 8, 10}
B - A = {4, 8}   C - A = {3, 7}

Complement (‘): Show only elements that is found NOT found this sets
A’ = {3, 4, 7, 8}   B’ = {-1, 0, 3, 5, 7, 9}   C’ = {-1, 0, 2, 4, 6, 8, 10}
(A ∪ B)’ = {3, 7}   (A ∩ B)’ = {-1, 0, 3, 4, 5, 7, 8, 9}
Difference can be seen as the same as complement.

Equality: Both sets must have exactly the same number of elements with exactly the same value. Take note that sequence and duplication does not affect the set.

A = {3, 4, 7, 8} Z = {4, 3, 7, 8} therefore A = Z
B = {3, 4, 7, 8, 4} Y = {3, 4, 7, 8, 7} therefore B = Y
A = B = Y = Z

Compatible: Two sets are compatible if all element in one of the set can fit nicely inside another set.
A = {x, b} Z = {x, b, c} therefore A is compatible to Z Because elements in A (x and b) can fit inside element in Z also have (x, b)
A = {x, b} Y = {b, c} therefore A is not compatible to Y Because elements in Y cannot contain (x) and A cannot contain (c).

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. How is set theory useful in formal methods?
2. List the terminologies used to describe sets relationship
3. State the differences between a finite and infinite element of set
4. Discuss three operations on set with appropriate examples
5. Illustrate the difference between a null set and a singleton using set notation only.

5.0 Conclusion

Set is a mathematical concept used in grouping objects. It could also be used to model relation between two sets or among several sets.

6.0 Summary

A set is a group of elements. In this unit we have examined Universality of set, Cardinality of set, elements of set (ie Finite Elements and Infinite Elements), relationships, set operations/operators

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 5: Series or Sequence

1.0 Introduction

It is sometimes necessary to record the order in which objects are arranged: for example, data may be indexed by an ordered collection of keys; messages may be stored in order of arrival; tasks may be performed in order of importance. In this chapter, we introduce the notion of a sequence: an ordered collection of objects. We examine the ways in which sequences may be combined, and how the information contained within a sequence may be extracted. We show that the resulting theory of sequences falls within our existing theory of sets,and provide formal definitions for all of the operators used. The chapter ends with a proof method for universal statements about sequences.:

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Define a sequence using natural and notation
2. Mention different types of sequence
3. Find a term in a given sequence
4. Find a sum in sequence

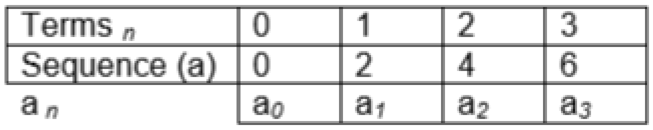
3.0 Main Content

3.1 Background

A sequence is simply a list, such as 2, 4, 6, ... where the numbers 2, 4, etc. are the terms of the sequence.

Please take time to understand this terminology:

Terms (usually represented with a subscript italic letter "n") refers to the index for a given sequence starting from 0 to infinite. The sequence (usually represented with any small letter "a") refers to the value for a specific term. For example:



Terms $_n$	0	1	2	3
Sequence (a)	0	2	4	6
a_n	a_0	a_1	a_2	a_3

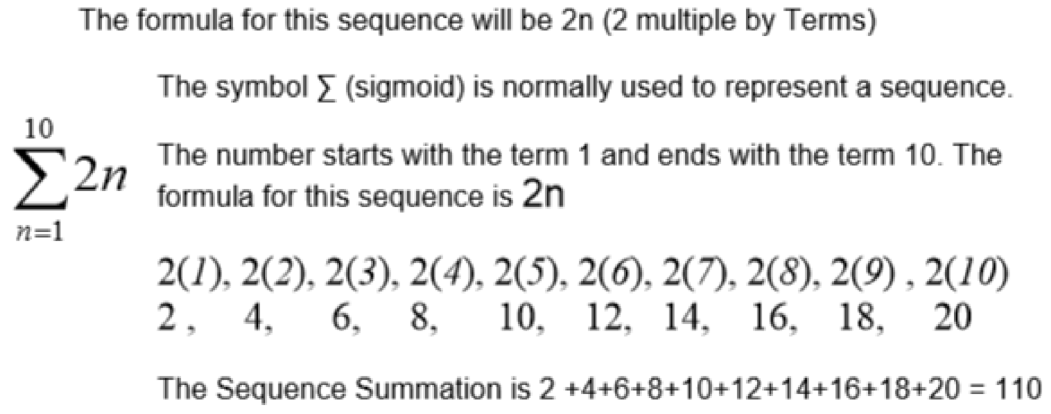
The term 0 has the number 0

The term 1 has the number 2

The term 2 has the number 4

The term 3 has the number 6

The formula for this sequence will be 2n (2 multiple by Terms)



The formula for this sequence will be 2n (2 multiple by Terms)

The symbol ∑ (sigmoid) is normally used to represent a sequence.

$$\sum_{n=1}^{10} 2n$$

The number starts with the term 1 and ends with the term 10. The formula for this sequence is 2n

$2(1), 2(2), 2(3), 2(4), 2(5), 2(6), 2(7), 2(8), 2(9), 2(10)$

$2, 4, 6, 8, 10, 12, 14, 16, 18, 20$

The Sequence Summation is 2 +4+6+8+10+12+14+16+18+20 = 110

3.2 Type of sequence

There are two type of sequence;

Finite sequence A finite sequence has both a starting value and an ending value.

E.g. 1, 2, 3, 4, 5 and 6

Infinite sequence An infinite sequence has both a starting value but no ending value

E.g. 1, 2, 3, 4, 5, 6 …

Sequences can be applied in two areas;

Arithmetic sequence

Arithmetic sequence a.k.a. Arithmetic progression or is a sequence of numbers that goes from one term to the next by always adding (or subtracting) the same value.

Geometric sequence

Geometric sequence a.k.a. geometric progression is a sequence of numbers that goes from one term to the next by always multiplying (or dividing) by the same value. Value multiple by the same value, Value divided by the same value.

3.3 How to find a SEQUENCE for a given term

Arithmetic sequence

Finite or Infinite Sequence

a = the number for the first term in the sequence

d = the common difference (first term - second term)

n = the number of terms in the sequence needed

$$a_n = d(n-1) + a$$

Example:

Term	1	2	3	4	5	6	7	8	9	10
	1	3	5	7	9	11	13	15	17	**19**

Given X = 1, 3, 57, 9, 11, what is the 10 terms?

X_{10} = +2(10 – 1) + 1 = +2 (9) + 1 = 18 + 1 = **19**

Geometric sequence

Finite or Infinite Sequence

a = the number for the first term in the sequence

r = ratio for the sequence

n = the number of terms in the sequence needed

$$a_n = r^n$$

Example:

Term	1	2	3	4	5	6	7	8	9	10
	4	16	64	256	1024	4096	16384	65536	262144	**1048576**

Given X = 4, 16, 64, 256, 1024, what is the 10 terms?

$X_{10} = 4^{10}$ = **1048576**

How to find a SUM for a given term

Arithmetic sequence

Finite or Infinite Sequence

a	= the first term in the sequence
a_n	= the last term in the sequence
d	= the common difference (first term - second term)
n	= the number of terms in the sequence needed

$$S_n = \left(\frac{a + a_n}{2} \right)(n)$$

Remember:
Average the first and last then multiply by the number of terms

Example:

Term	1	2	3	4	5	6	7	8	9	10
	1	3	5	7	9	11	13	15	17	19

SUM = 1+ 3+5+7+ 9+11+13+15+17+19 = 100

Given X = 1, 3, 57, 9, 11, 13, 15, 17, 19 calculate the sum of X_{10}?

X_{10} = ((1+ 19)/2)10 = (20/2)10 = (10)10 = 100

Geometric Sequence

a	= the number for the first term in the sequence
m	= start terms for the given sequence
n	= stop terms for the given sequence
r	= ratio for the sequence

Finite sequence (**the first term value must above 0**)

$$S_n = \frac{a(1-r^n)}{1-r}$$

Example:

Term	1	2	3	4	5	6	7	8	9	10
	4	16	64	256	1024	4096	16384	65536	262144	1048576

SUM = 4 + 16 + 64 + 256 + 1024 + 4096 + 16384 + 65536 + 262144 + 1048576 = $\underline{\underline{1398100}}$

Given X = = 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, Calculate the sum of X?

Sum X_{10} = 4 (1 - 4^{10}) / 1 – 4
= 4 (1-1048576) / -3 = 4 (-1048575) / -3
= -4194300 / -3 = $\underline{\underline{1398100}}$

Infinite sequence that start (**the first term value must above 0**)

$$S\infty = \frac{a}{1-r}$$

Example:

Term	0	1	2	3	4	5	6	…
	4	16	64	256	1024	4096	16384	…

SUM = 4 + 16 + 64 + 256 + 1024 + 4096 + 16384 + …

Given X = = 4, 16, 64, 256, 1024, 4096, 16384 … Calculate the sum of X?

Sum X_{∞} = 4 / 1 – 4 = 4 / -3 = $\underline{\underline{-1.3333333}}$

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Mention the different types of sequence
2. Describe Arithmetic and Geometrical Sequences with appropriate examples.
3. Find a sum in the following sequence 1, 4, 9…10000.

5.0 Conclusion

A sequence is list of numbers or terms generated used a particular expression or construct. There are basically two types of sequence namely: arithmetic and geometric.

6.0 Summary

The concept of sequence was introduced. Different types of sequence were elaborated upon, i.e., finite and infinite. Examples of Arithmetic and Geometric sequencewere illustrated. Arithmetic and Geometric sequence associated with addition (subtraction)and multiplication (division) respectively.

77.0 Further Reading

Formal methods - Wikipedia, the free encyclopaedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

MODULE 6: FORMAL METHODS CONTINUES

1.0 Introduction

Proof simply means to be able to show that a statement is correct or true. No matter how the statement is twisted and turned or set against many different scenario, that statement comes up with the constant answer.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Discuss formal proof
2. Mention some terminologies used in mathematical proof
3. Briefly explain the four proofing methods

3.0 Main Content

3.1 What is proof?

Proof simply means to be able to show that a statement is correct or true. No matter how the statement is twisted and turned or set against many different scenario, that statement comes up with the constant answer.

Before a statement can be proof it can have two (2); the conditions followed by the result. For example, if it rains then I will be wet. This can then express in a using proposition symbols as;

Rain →I am wet (if it rains then I will be wet)

3.2 Terminology

1) Conjecture/ Hypothesis

This is a statement that is believe to be true but has yet toC be proven.

2) Axiom/ Postulate

If the statement is taken for granted to be true even though it was never tested, but base on logic it is assume to be true.

3) Paradox/ Antinomy

This is a statement which appears to contradict itself or contrary to expectations

4) Theorem
This is a statement that has been proven to be true.
5) Un-decidable
This is a statement that cannot be proven right or wrong.
6) Lemma
A proven theorem that is used to prove other statements
7) Converse
Theorem that is reversed or turned upside down or inward out thus a converse of a theorem need not be always true.

3.3 Proofing Methods

1. Direct Proof
In direct proof, the conclusion is established by logically combining the axioms, definitions, and earlier theorems. From the expression one can directly see the answer.
For example: Rain →I am wet
2. Contradiction Proof
In proof by contradiction, if that statement is true and we logically contradict it then it will not be true anymore.
For example: Rain → I am wet
No Rain →I am Dry
3. Contra-positive/ Transposition Proof
Proof by transposition or contra-positive turns the statement inside out and upside down. This method swaps the result into the condition and negates both the result and condition.
For example: Rain →I am wet
I am Not wet →No Rain
4. Induction Proof
This proof method insists that if the statement is true for one instance, it should be true for every instance.
For example: Rain →I am wet
On Monday (Rain →I am wet)
On Tuesday (Rain →I am wet)
On Wednesday (Rain →I am wet)
On Thursday (Rain →I am wet)
On Friday (Rain →I am wet)

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. State at least four proofing methods.
2. Illustrate induction proof with example.
3. Explain the following terms: Conjecture/ Hypothesis, Axiom/ postulate, Lemma
4. Explain the concept of formal proof.
5. Describe the contradiction proof.

5.0 Conclusion

Proofs are the heart and soul of mathematics, no matter how simple or complicated they are. They play a central role in the development of mathematics and guarantee the correctness of mathematical results and algorithms. No mathematical results or computer algorithms are accepted as correct unless they are proved using logical reasoning.

6.0 Summary

Proofs are meant to show the correctness or otherwise of a statement. In software development, it could be used correctness of program statement or algorithm. We have examined some terminologies used in proof and the various proofing methods.

7.0 Further

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 2: Testing

1.0 Introduction

Test is a way of validating and verifying software. This ensures the removal and/ or reduction of errors to the barest minimum. It further ensures that the right product is crafted and that it meets user’s requirement specifications. Formal methods can be used to achieve this to a higher extent.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Outline stages in SDLC
2. Identify the focus of both validation and verification during software testing
3. Discuss test plan
4. Produce a sample of test plan
5. State the content of a test case
6. Discuss test in terms of size

3.0 Main Content

3.1 Software Development Life Cycle

The SDLC can be divided into seven (7) stages;

1. Initial Study:Team collects information regarding the problem.
2. Analysis:Team discover the source of the problem.
3. Design: Team creates the specification for the solution.
4. Development: Team built the solution base on the given specification.
5. Testing: Test the software to make sure it solves the problem.
6. Implementation: Team prepare the environment to accept software.
7. Review: Team and client review the software.

3.1.1 Revise Formal Method Process

The formal specification generally does the following process.

1. Get user requirement usually from the specification written in the natural language.
2. Clarify the requirement using mathematical approach. This is to remove all ambiguous, incomplete and inconsistent statement.

3. After statements are clearly identified. Then find all assumptions (Things that must be in place before something can happen) that is state or not stated within the clarified requirement.
4. Then expose every possible logic defect (fault) or omission in the clarified requirement.
5. Identify what are the exceptions (bad things) that will arise if the defects are not corrected.
7. Find a way to test for all the possible each exception. Only when you can test for an exception can you be able to stop that exception from happening.

3.2 Testing stage

A test stage has only one (1) important purpose, that is to ensure that they software solution built solves the problem as specified in the analyst report and the specification.
Validation focuses on building the right solution and Verification focuses on building the product correctly are two terms used a lot during testing.
No software is 100% bugs free and testing cannot guarantee that there are no bugs, it can only ensure to a certain reasonable level that the system is able to perform the task it was created to do. It does not mean there are no more bugs in the system.
Once the testing is done, the tester will write a letter to give their opinion on the testing and the test result.
This letter will be given to the SA who then decides the following action, which may take the form of returning to:
1) The development stage where the programmer will debug the problem.
2) The design stage to redesign the specification then later continue to the development stage.
3) Worst case scenario, to return to the Analyst stage to redo the analyst for that given module which will then continue into the design and development stage again.
The same test document is reuse when the software returns to the testing stage. The SA may add in new test item in the test plan and new test case but the previous test document must remain intact. The tester will then repeat the testing for the failed modules and the new modules.
After a successful test, if there are future changes, the same test document is also reuse, thus the test plan can be use to audit the changes to ensure changes do not introduce new problem.

The test document consist of a test plan that list down all the test item, each test item will then be reflected in one (1) or more test case.

For example; Test plan will have many test items. In one (1) of the test item there is a test for the customer name. There may be three (3) test cases for that test item;

1) To test if the customer name can be save.
2) To test if the customer name can be numeric.
3) To test if the customer name can be blank.

3.3 Test plan

A test plan is a document that state down clearly every step (test item) that will be taken during testing, basically a systematic approach to testing a system.

The Test Plan is created first by the System Analyst (SA) using the Analyst and Design deliverables. This is done during the development stage when the work load has been transferred to the software programmers.

In order to create the Test Plan, the SA must understand the testing concept, because the strategy applied by the SA can easily be seen in the test plan.

The test plan will normally contain:

1. A sequence number call the test plan no.
2. The general description of the test item.
3. The date for the completion of each test plan no. One (1) test item can hold many test cases and each test case has a different purpose.

The tested date is inserted only after ALL the test cases for one (1) test item is successfully tested. If after the testing is done and the test plan date remains blank means that the software fails the testing.

This is sample of a test plan

No	Description	Tested Date
1	System Installation into a Windows XP Professional Operating System	
2	Login Program.	
3	Main Menu	
4	Customer Module	
5	Customer Particular	
6	Customer Name	
7	Customer Search	
. . .	. . .	. . .

3.4 Test Case

After completing the test plan the SA will then create a test case. There will be at least one (1) test case for each of the test in the test plan. Each test case can contain only one (1) set of instruction and one (1) outcome.

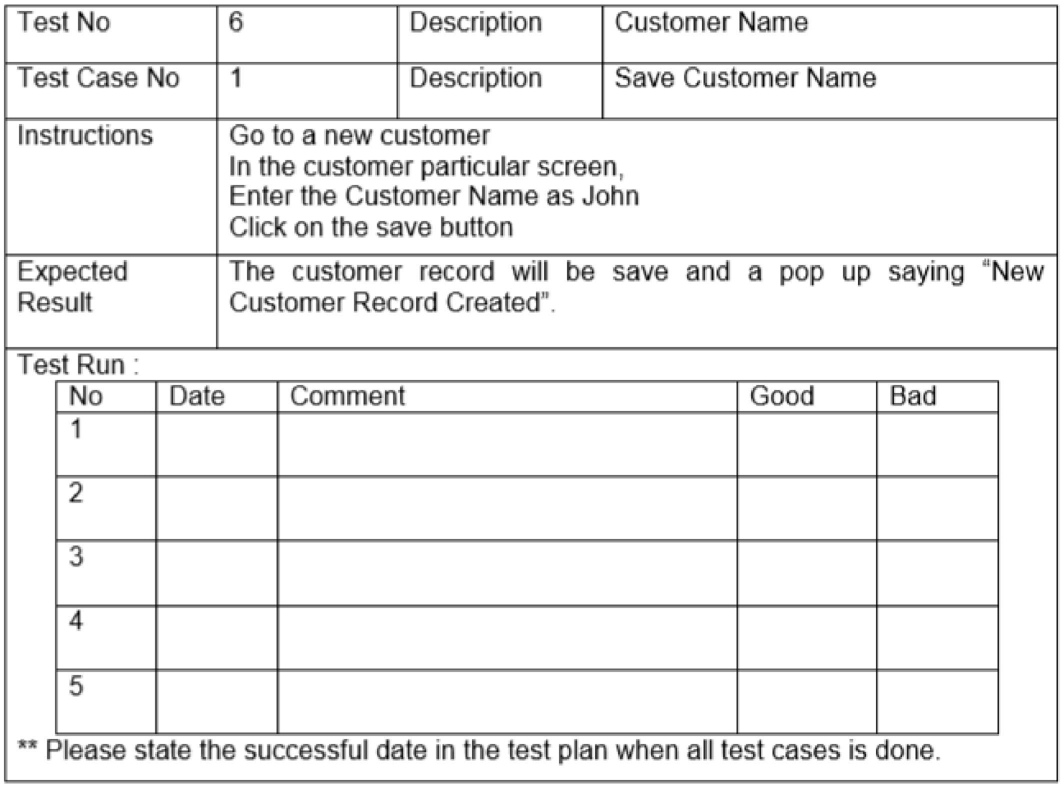
There will always be a BLANK table inserted in the test case to be use by the tester to fill in the result of that particular testing.

The test case will normally contain:

1. The test plan no to tally back to the test plan and a test case number for that given test case.
2. The test instruction explains to the tester exactly how to run that particular test.
3. The expected result for that given test usually with a simple screen design or simple sentences to describe the result.

Please remember, SA creates the test cases for the tester.

This is a sample of a test case



Test No	6	Description	Customer Name
Test Case No	1	Description	Save Customer Name
Instructions	Go to a new customer In the customer particular screen, Enter the Customer Name as John Click on the save button		
Expected Result	The customer record will be save and a pop up saying "New Customer Record Created".		

Test Run :

No	Date	Comment	Good	Bad
1				
2				
3				
4				
5				

** Please state the successful date in the test plan when all test cases is done.

3.5 Testing Concept

3.5.1 Test Flow

1. Top Down

This is a test flow that starts from a general level down to the specific detail level. Example for an inventory system will be to start from a main menu and slowly make the way down to the product module.

2. Bottom Up

This is a test flow that starts from a detail specific level up to the general level. Example for an inventory system will be to start from the products and slowly make the way up to the main menu.

3.5.2 Test Size

1. Unit Testing

This is a test that focuses on an individual specific independent module. Example for an inventory system will be to start test the products module alone and then the customer module alone.

2. Integration Testing

This is a test that starts to join individual module. Example for an inventory system will be after testing out the product and customer module to test out the sales invoice module.

3. System Testing

This is a test that starts to studies the system environment surrounding the software. Example for an inventory system will be to test if the bar code reader at the POS can read the barcode label on the product.

4. User Acceptance Testing

This is the final a test where the end user will physically tested out the system themselves using real life data but still in a control testing environment. Example for an inventory system will be to ask the POS staff to test out the POS system and enter 100 products and produce the correct balance on the receipt.

3.6 Test Depth

1. Black box testing

This is commonly known as a validation testing. This is an effectiveness test that is result base, input is given and output is produce and compared. Example for an inventory system will be to scan a barcode label on the product and see it appear on the POS interface with the correct total.

2. White box testing

This is commonly known as a verification testing. This is an efficiency test, to test out how much resources and time is required to complete a process. Example for an inventory system will be to see how much time and processing resources to list and print a sales report for 1000 products.

3. Grey box testing

This is partial effective and an efficiency test. Basic processing information is needed to discover how a process works. This test is normally use to create the test case.

3.7 Other Test

1. Boundary Testing

This is a test done usually with a black box that can be done at the unit testing or integration testing stage. The main objective of this test is to make sure that the software can make the correct decision. All the possible result and alternative value is determined for the condition. Then extreme test data are generated to see if the software can produce a correct result.

2. Stress Testing

This is a test done usually with a black box, unit testing approach. The main objective will be to break the system. Thus, this test will take along time as it will continue until the system breaks down. From the break down, a safe level can be reach for contingency planning.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Identify the stages in SDLC
2. Explain the importance of validation and verification during software testing
3. Discuss at least three test plans in software development.
4. State the content of a test plan and test case
5. Discuss test in terms of size

5.0 Conclusion

Testing is carried out in software to eliminate errors or at least to reduce it to the barest minimum. Different proofing methods can be used to achieve this.

6.0 Summary

In this unit we have explain different types of testing including unt testing, integration testing, system testing etc.

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

Unit 3: Application to Formal Specification

1.0 Introduction

Formal methods are used at various stages of software development to improve the quality of the software.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Discuss the various stages to apply formal methods
2. Discuss what to do at various stages

3.0 Main Content

3.1 Formal method

Formal methods are a way to apply mathematically-based techniques to the specification, development and verification of software for computer science or software engineering. When Formal method is applied it is likely to produce a more reliability and robustness specification design.

Thus, it is important to stress that Formal Method in itself cannot guarantee perfect software. It depends very much on how the formal methods are interpreted and applied into the specification.

Formal Method stages consist of; Formal Specification, Formal Proof, Model Checking and Abstraction.

3.2 Formal Specification

Formal Specification is the initial part of formal method that describes what the system must do without saying how it is to be done. It is totally language independent and focuses only on the abstract rather than detail logic.

A formal specification can serve as a single, reliable reference point for those who investigate the customer's needs, those who implement programs to satisfy those needs, those who test the results, and those who write instruction manuals for the system.

3.3 Formal Specification in the SDLC

Two (2) things are very important during requirement gathering; the data and the process done on the data. Formal specification will be applied directly on these things.
If formal Specification is used during the Analysis and Design stage, there is no need to use them again. If formal specification is not used by the SA and SD then the only window of opportunity to apply it will be during the pre-development stage.
If no formal specification was ever applied, and it is applied in the testing stage, this will expose a lot of possible bugs not catered for and the problem will loop back to the design and possibly the analysis stage.

3.3.1 Analyst Stage

The main purpose of Analyze is to find the source of the problem. During this stage the System Analyst (SA) would collect all the data and processes (observation, document inspection, interview, etc…). The SA would then express the requirements into some form of diagrams such as DFD or Rich Picture, Use Cases and Case Diagram, etc…
Then the SA will write a report (in English – common natural language) to indicate the source of the problem and some alternative solution.

When studying the current system, the SA could apply proposition and predicate to every client’s statement thus translating them into mathematical equivalent. This will remove ambiguity and expose all possible hidden state of the process and data. Proof is then use to be sure if the statement given by the client is correct. The SA can also apply set theories and series to categories and view data from different perspective. This is very helpful when SA needs to understand how reports are generated.
By doing this the SA can be to a certain degree confident to cover all possible alternative to a given statement.

3.3.2 Design Stage

The main purpose of Design is to create a specification for the selected solution. It should be stressed that, the specification will be use to built the solution, thus a good specification will create a good software and a bad specification will create a bad software.
During this stage the System Designer (SD) uses the analyze report to create the specification. The SD also expresses their specification into some form of diagrams sometimes similar to those used by the SA.

Before creating the specification, the SD could translate all the natural language found in the analyze report into mathematical equivalent (proposition and predicates) that will remove all ambiguity and uncertainty. Proof can be use here to ensure that every statement given is logically correct. Set theories and series are use to categories and view data from different perspective and to create relevant reports.
Then studying the mathematical form, the SD will be able to create the new system environment and also the solution that can cater for all possible scenario and state for each data and processes.

3.3.3 Development Stage

The main purpose of Development is to find built the solution base on the specification. During this stage the Senior Programmer (SP) will study the specification, create the relevant data structure, study the modules and delegate the programming team to develop the solution.
To apply formal specification at this stage will be a bit late but a small window of opportunity still exists.
During this stage the Senior Programmer (SP) will study the specification. The SP could translate all the natural language found in the specification into mathematical equivalent (proposition and predicates) that will remove all ambiguity and uncertainty. Proof can be use here to ensure that every statement given is logically correct. Set theories and series are use to categories and view data from different perspective and to create relevant reports.

The SP can then verify that the specification is complete before starting out the development.
If there is any problem with the design, the SP will stop the development and return the specification to the SD for correction. Worst case scenario, the entire specification is drop and the system is reanalyzed.

3.3.4 Testing Stage

The main purpose of Testing is to make sure that the solution solves the problem found in the analysis and created base on the specification. Before this stage the (SA) will have already created the test plan and test case. In this stage the tester will then use the test plan and test case to execute the testing.

To apply formal specification at this stage is really very late.

During this stage the SA will revise the specification built during by the SD. The SA could then translate all the natural language found in the specification into mathematical equivalent (proposition and predicates) that will remove all ambiguity and uncertainty. Proof can be use here to ensure that every statement given is logically correct. Set theories and series are use to categories and view data from different perspective and to create relevant reports.

After studying the specification, then the SA can create a Test Plan that will cover all aspect of the system. Using what is learnt from the Formal specification, the SA will be able to create a test case for each test item to test out all the different exception.
If there is any problem with the testing, the SA will stop the testing and return the specification to the SD for correction. Worst case scenario, the entire specification is drop and the system is reanalyzed.
Notice that this will not return to the development, because development only follows the specification created during the design stage.
Formal Specification should not be use during the Implementation stage

4.0 Self-Assessment Exercise(s)
Answer the following questions:

1. Explain the various stages in software development that formal methods is applied.
2. State the deliverables at various stages of software development life cycle
3. Describe how formal method is implemented in system development.
4. List the available tools used by the systems analysts during software development.
5. Explain why formal specification is considered to be late at the testing phase of system development.

5.0 Conclusion
Formal methods are applied at various stages of software development in order to precisely specify the requirement of the system being developed and to find and remove errors. The application of formal methods assistsin crafting error free, safe and reliable software.

6.0 Summary

The application of formal methods at several stages of software development are discussed.

7.0 Further Reading

Formal methods - Wikipedia, the free encyclopedia [online] Available at http://en.wikipedia.org/wiki/Formal_methods

FTMS Consultants (M) Sdn Bhd (2011)SD3049 Formal Methods in Software Engineering Kuala Lumpur, Malaysia

L. M. Barroca, J. A. McDermid (1997) Formal Methods: Use and Relevance for the Development of Safety-Critical Systems, THE COMPUTER JOURNAL, VOL. 35, NO. 6

Michael Jackson, Patrick Cousot, Jonathan Peter Bowen, Margaria Tiziana (2008) Software engineering and formal methods, ACM

Mona Batra, Amit Malik, Dr. Meenu Dave (2020) Formal Methods: Benefits, Challenges And Future Direction, *Journal of Global Research in Computer Science*

Zoltán Istenes (2014) FormalMethods in Software Engineering

MODULE 6:SOFTWARE DEVELOPMENT OVERVIEW

1.0 Introduction

Much of our endeavour in software development is the design and construction of software to meet some recognised need – of people, organisations or society at large – with tangible effect on the real world. **Software Development process** is the practice of organising the design and construction of software and its deployment in context. Note that **Software development and Software Engineering can be used interchangeably. In this unit, we shall be give some definitions and shall be discussing**Software Evolution, Software Paradigms, Need of Software Engineering, and Characteristics of good software

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Relate software development with engineering process
2. State some software evolution laws
3. Discuss E-Type software evolution
4. Discuss the need of Software Engineering
5. Outline the characteristics of good software

3.0 Main Content

3.1 Definitions

3.1.1 Software

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

3.1.2 Engineering

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

3.1.3 Software Development

Software development refers to a set of computer science activities dedicated to the process of creating, designing, deploying and supporting software. Software itself is the set of instructions or programs that tell a computer what to do. It is independent of hardware and makes computers programmable. There are three basic types:

System software to provide core functions such as operating systems, disk management, utilities, hardware management and other operational necessities.

Programming software to give programmers tools such as text editors, compilers, linkers, debuggers and other tools to create code.

Application software (applications or apps) to help users perform tasks. Office productivity suites, data management software, media players and security programs are examples. Applications also refers to web and mobile applications like those used to shop on Amazon.com, socialize with Facebook or post pictures to Instagram.[1]

3.1.4 Software Developer

Software developers have a less formal role than engineers and can be closely involved with specific project areas — including writing code. At the same time, they drive the overall software development lifecycle — including working across functional teams to transform requirements into features, managing development teams and processes, and conducting software testing and maintenance.[3]

3.1.5 Software Engineering

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Definitions

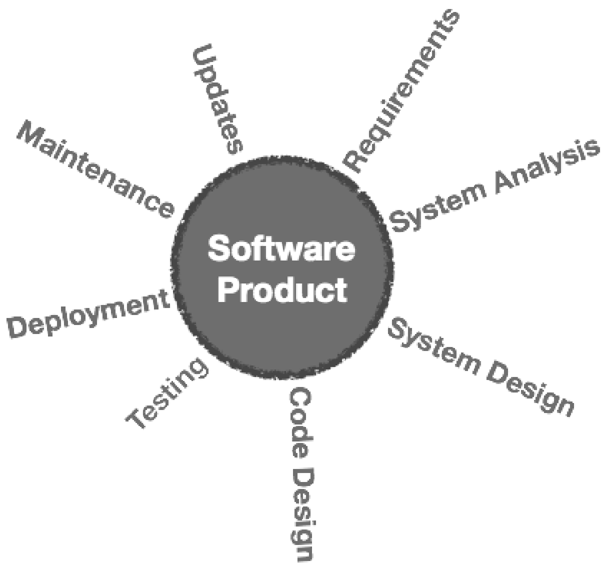
IEEE defines software engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in the above statement.

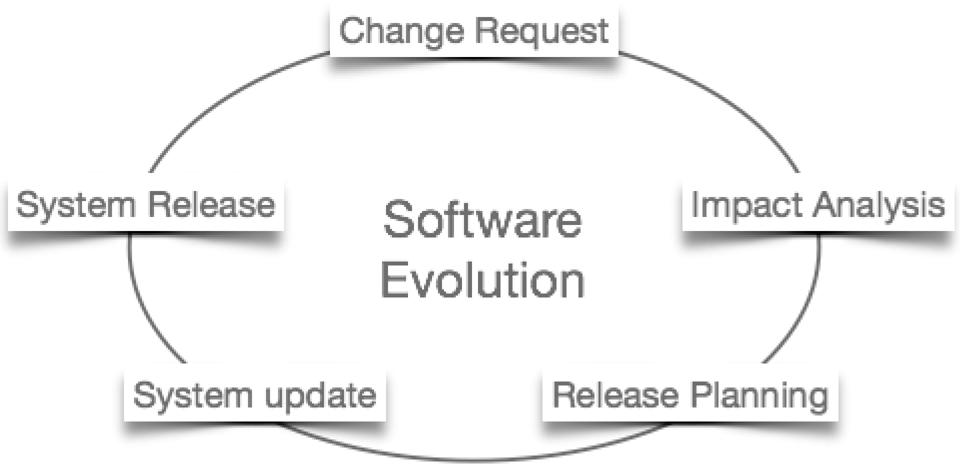
Fritz Bauer, a German computer scientist, defines software engineering as:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically viable software that is reliable and work efficiently on real machines.



3.2 Software Evolution

The process of developing a software product using software engineering principles and methods is referred to as **software evolution.** This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements.



Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and to go one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

3.2.1 Software Evolution Laws

Lehman has given laws for software evolution. He divided the software into three different categories:

1. **S-type (static-type) -** This is a software, which works strictly according to defined specifications and solutions. The solution and the method to achieve it, both are immediately understood before coding. The s-type software is least subjected to changes hence this is the simplest of all. For example, calculator program for mathematical computation.
2. **P-type (practical-type) -** This is a software with a collection of procedures. This is defined by exactly what procedures can do. In this software, the specifications can be described but the solution is not obvious instantly. For example, gaming software.
3. **E-type (embedded-type) -** This software works closely as the requirement of real-world environment. This software has a high degree of evolution as there are various changes in laws, taxes etc. in the real world situations. For example, Online trading software.

3.2.2 E-Type software evolution

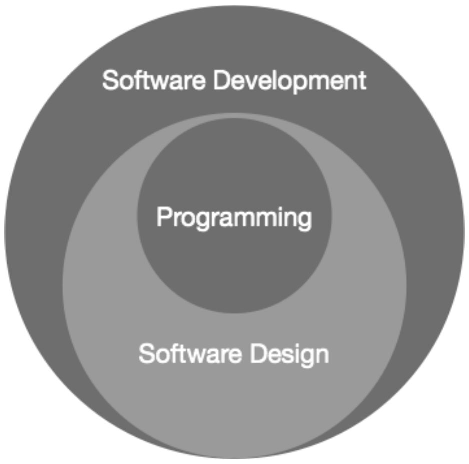
Lehman has given eight laws for E-Type software evolution -

- **Continuing change -** An E-type software system must continue to adapt to the real-world changes, else it becomes progressively less useful.
- **Increasing complexity -** As an E-type software system evolves, its complexity tends to increase unless work is done to maintain or reduce it.
- **Conservation of familiarity -** The familiarity with the software or the knowledge about how it was developed, why was it developed in that particular manner etc. must be retained at any cost, to implement the changes in the system.
- **Continuing growth-** In order for an E-type system intended to resolve some business problem, its size of implementing the changes grows according to the lifestyle changes of the business.
- **Reducing quality -** An E-type software system declines in quality unless rigorously maintained and adapted to a changing operational environment.

- **Feedback systems-** The E-type software systems constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.
- **Self-regulation -** E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
- **Organizational stability -** The average effective global activity rate in an evolving E-type system is invariant over the lifetime of the product.

3.3 Software Paradigms

Software paradigms refer to the methods and steps, which are taken while designing the software. There are many methods proposed and are in work today, but we need to see where in the software engineering these paradigms stand. These can be combined into various categories, though each of them is contained in one another:



Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

3.3.1 Software Development Paradigm

This Paradigm is known as software engineering paradigms where all the engineering concepts pertaining to the development of software are applied. It includes various researches and requirement gathering which helps the software product to build. It consists of –

- Requirement gathering
- Software design
- Programming

3.3.2 Software Design Paradigm

This paradigm is a part of Software Development and includes –

- Design
- Maintenance
- Programming

3.3.4 Programming Paradigm

This paradigm is related closely to programming aspect of software development. This includes –

- Coding
- Testing
- Integration

3.4 Need of Software development

The need of software development arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.
- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down he price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

3.5 Characteristics of good software

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

3.5.1 Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

3.5.2 Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

3.5.3 Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. List and explain the basic concepts in Z notation.
2. Explain the following terms: formal specification, Formal Verification and Theorem Proves

3. Explain maintainability, scalability and modularity
4. Compare and contrast interoperability and reusability
5. State the characteristics of a good software
6. Explain the basic concepts of software paradigm.

5.0 Conclusion

Definitions of some software development concepts are given. Software has gone through evolutionary processes which has been highlighted

6.0 Summary

In this unit software evolution, some paradigms, characteristics of good software etc have been discussed. The need of Software Development is also examined.

7.0 Further

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 2: Software Development Life Cycle

2.0 Introduction

Software Development Life Cycle, SDLC allows the software developer or engineer to follow well-defined phases or stages to achieve quality in the design and construction of software product that will meet user's need. That is, in terms of functionality, reliability, maintainability, availability etc. SDLC comes in different flavours. This includes among others – waterfall model, iterative model, spiral mmodel, V-model etc. The stages and various models will be discussed in this unit.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

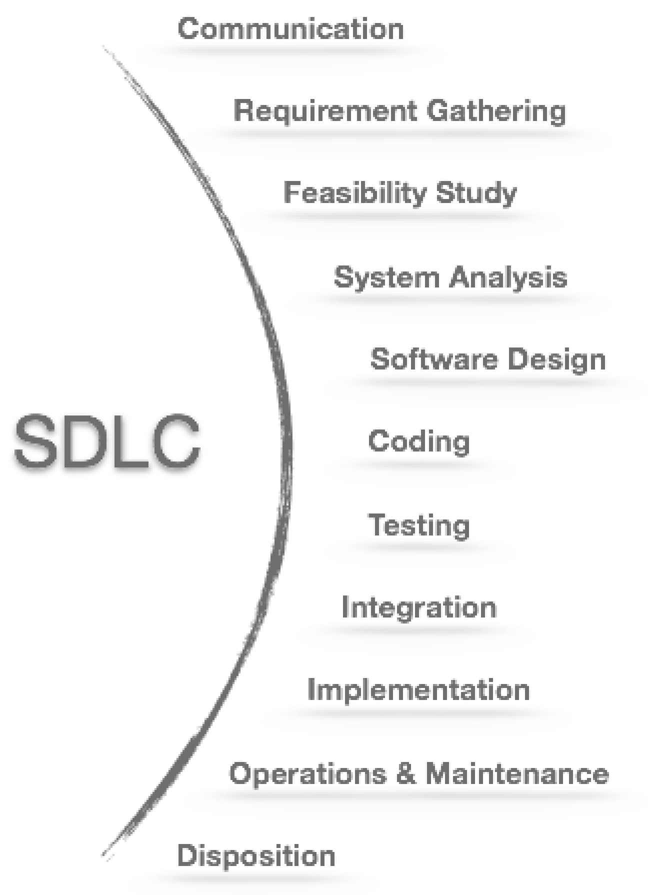
6. List the SDLC activities
7. Ecplain the SDLC activities with aid of a diagram
8. List and explain the Software Development Paradigm

3.0 Main Content

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software development to develop the intended software product.

3.1 SDLC Activities

SDLC provides a series of steps to be followed to design and develop a software product efficiently. SDLC framework includes the following steps:



1. Communication: This is the first step where the user initiates the request for a desired software product. He contacts the service provider and tries to negotiate the terms. He submits his request to the service providing organization in writing.
2. Requirement Gathering: This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given -
   - studying the existing or obsolete system and software,
   - conducting interviews of users and developers,
   - referring to the database or
   - collecting answers from the questionnaires.
3. Feasibility Study: After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if a software can be made to fulfil all requirements of the user and if there is any possibility of software being no more useful. It is found out, if the project is financially, practically and

technologically feasible for the organization to take up. There are many algorithms available, which help the developers to conclude the feasibility of a software project.

4. System Analysis: At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes Understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyzes the scope of the project and plans the schedule and resources accordingly.
5. Software Design: Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams and in some cases pseudo codes.
6. Coding: This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.
7. Testing: An estimate says that 50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing and testing the product at user's end. Early discovery of errors and their remedy is the key to reliable software.

7.Integration: Software may need to be integrated with the libraries, databases and other program(s). This stage of SDLC is involved in the integration of software with outer world entities.

8. Implementation: This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.
9. Operation and Maintenance: This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the

changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

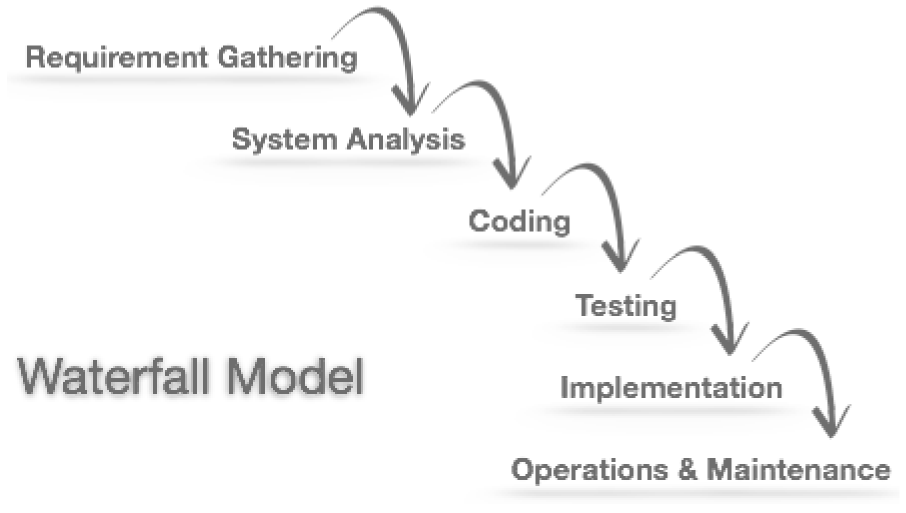
10 Disposition: As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense up gradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software components, closing down the system, planning disposition activity and terminating system at appropriate end-of-system time.

3.2 Software Development Paradigm

The software development paradigm helps developer to select a strategy to develop the software. A software development paradigm has its own set of tools, methods and procedures, which are expressed clearly and defines software development life cycle. A few of software development paradigms or process models are defined as follows:

3.2.1 Waterfall Model

Waterfall model is the simplest model of software development paradigm. It says the all the phases of SDLC will function one after another in linear manner. That is, when the first phase is finished then only the second phase will start and so on.

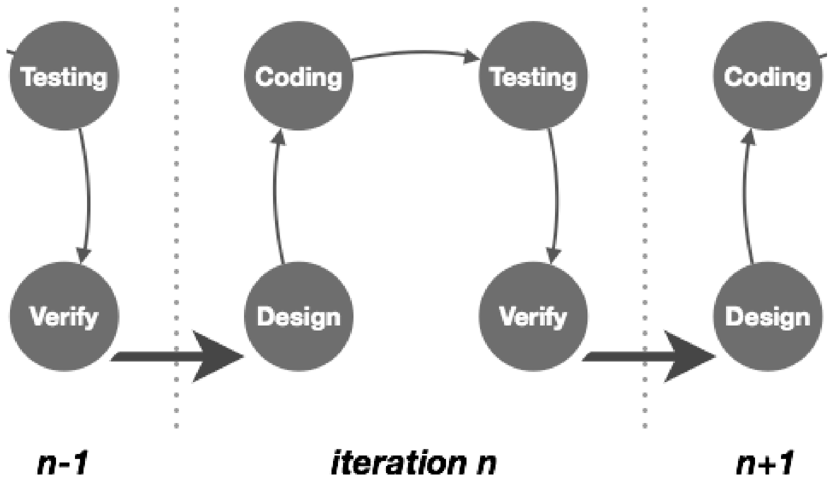


This model assumes that everything is carried out and taken place perfectly as planned in the previous stage and there is no need to think about the past issues that may arise in the next phase. This model does not work smoothly if there are some issues left at the previous step. The sequential nature of model does not allow us go back and undo or redo our actions.

This model is best suited when developers already have designed and developed similar software in the past and are aware of all its domains.

3.2.2 Iterative Model

This model leads the software development process in iterations. It projects the process of development in cyclic manner repeating every step after every cycle of SDLC process.

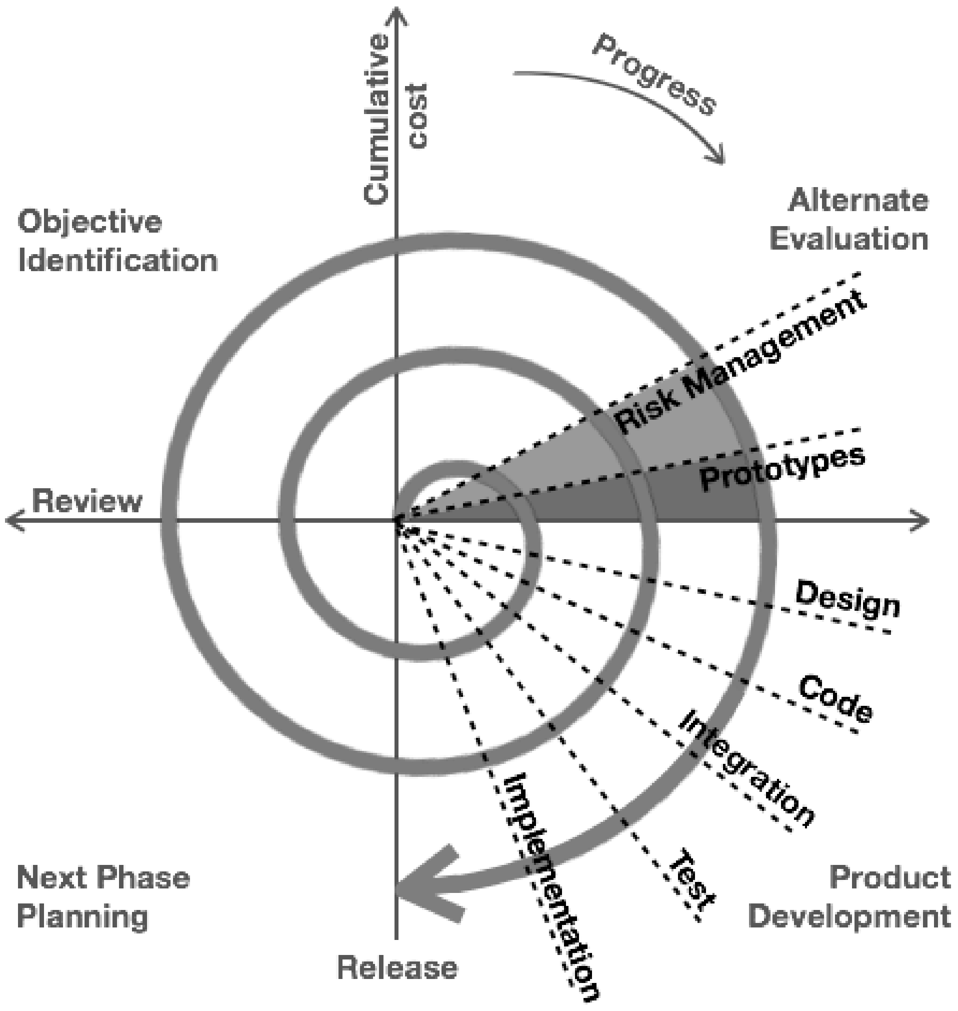


The software is first developed on very small scale and all the steps are followed which are taken into consideration. Then, on every next iteration, more features and modules are designed, coded, tested and added to the software. Every cycle produces a software, which is complete in itself and has more features and capabilities than that of the previous one.

After each iteration, the management team can do work on risk management and prepare for the next iteration. Because a cycle includes small portion of whole software process, it is easier to manage the development process but it consumes more resources.

3.2.3 Spiral Model

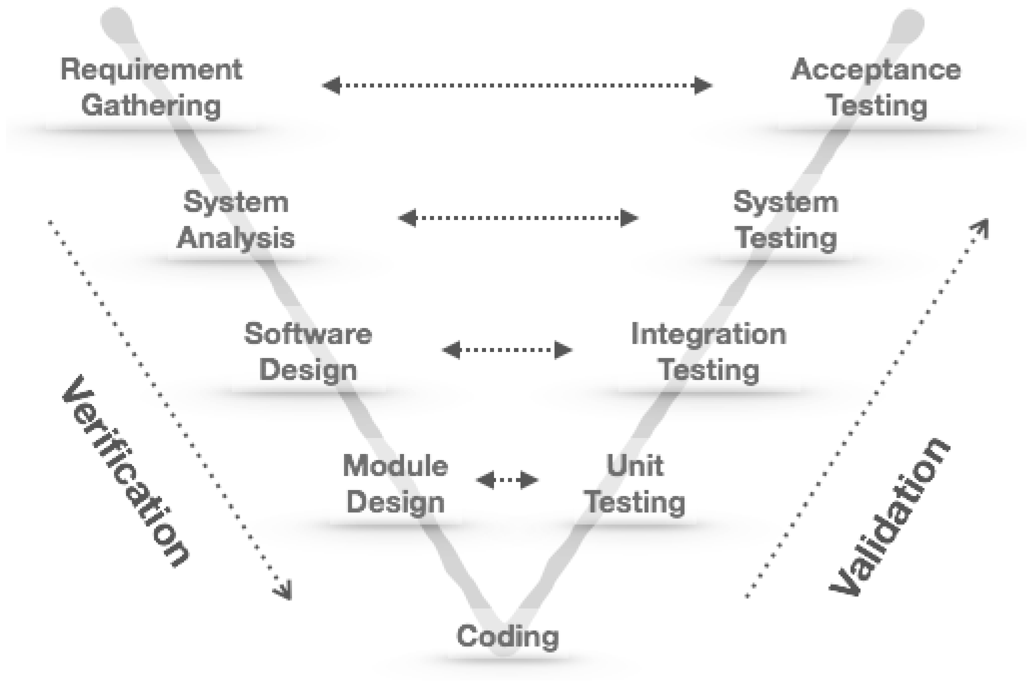
Spiral model is a combination of both, iterative model and one of the SDLC model. It can be seen as if you choose one SDLC model and combine it with cyclic process (iterative model).



This model considers risk, which often goes un-noticed by most other models. The model starts with determining objectives and constraints of the software at the start of one iteration. Next phase is of prototyping the software. This includes risk analysis. Then one standard SDLC model is used to build the software. In the fourth phase of the plan of next iteration is prepared.

3.2.4 V – model

The major drawback of waterfall model is we move to the next stage only when the previous one is finished and there was no chance to go back if something is found wrong in later stages. V-Model provides means of testing of software at each stage in reverse manner.

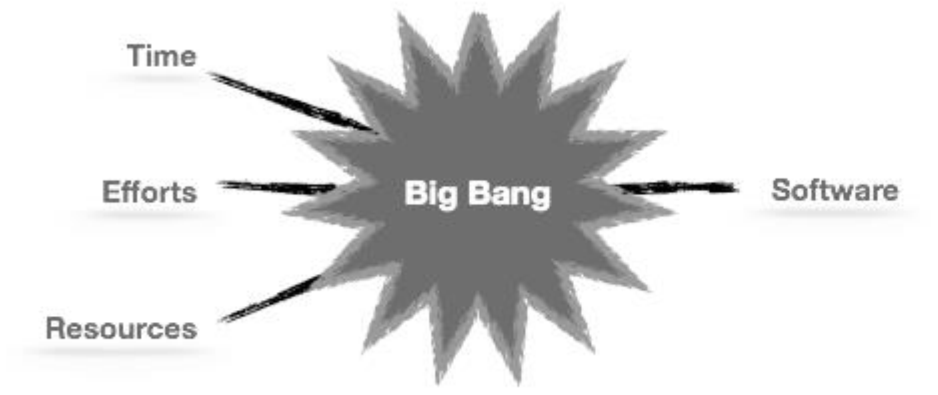


At every stage, test plans and test cases are created to verify and validate the product according to the requirement of that stage. For example, in requirement gathering stage the test team prepares all the test cases in correspondence to the requirements. Later, when the product is developed and is ready for testing, test cases of this stage verify the software against its validity towards requirements at this stage.

This makes both verification and validation go in parallel. This model is also known as verification and validation model.

3.2.5 Big Bang Model

This model is the simplest model in its form. It requires little planning, lots of programming and lots of funds. This model is conceptualized around the big bang of universe. As scientists say that after big bang lots of galaxies, planets and stars evolved just as an event. Likewise, if we put together lots of programming and funds, you may achieve the best software product.



For this model, very small amount of planning is required. It does not follow any process, or at times the customer is not sure about the requirements and future needs. So, the input requirements are arbitrary.
This model is not suitable for large software projects but good one for learning and experimenting.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. List at least four types ofsoftware development models.
2. State the weaknesses and strengths of each of the named model in (1) above.
3. Describe the iterative model of software development.
4. Explain the importance of model verification and validation in software development.
5. With the aid of a diagram, illustrate the stages involved in waterfall model.

5.0 Conclusion

Software Development Life Cycle consist of steps or phases in developing a software. The steps are as follows:Communication, requirement gathering, feasibility study, system analysis, system design, coding, iteration, implementation, operation and maintenance and disposition. There are quite a number of paradigms used in software development. This includes among others: water fall model, spiral model, V-model Iterative model etc.

6.0 Summary

The Software Development Life Cycle (SDLC) has been discussed. Also discussed are the various software paradigms, among which are: water fall model, spiral model, V-model Iterative model etc.

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 3: Software Project Management

1.0 Introduction

The job pattern of an IT company engaged in software development can be seen split in two parts:

- Software Creation
- Software Project Management

A project is well-defined task, which is a collection of several operations done in order to achieve a goal (for example, software development and delivery). A Project can be characterized as:

- Every project may have a unique and distinct goal.
- Project is not routine activity or day-to-day operations.
- Project comes with a start time and end time.
- Project ends when its goal is achieved hence it is a temporary phase in the lifetime of an organization.
- Project needs adequate resources in terms of ti
- me, manpower, finance, material and knowledge-bank.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Identify the characteristics of a software project
2. Describe a software project
3. Justify the need for software project management
4. Identify the job of a software project manager
5. Explain the following: project planning, scope management and project estimation
6. Mention at least 3 project management tools

3.0 Main Content

3.1 Software Project

A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

3.1.1 Need for software project management

Software is said to be an intangible product. Software development is a kind of all new stream in world business and there’s very little experience in building software products. Most software products are tailor made to fit client’s requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one. All such business and environmental constraints bring risk in software development hence it is essential to manage software projects efficiently.

The image above shows triple constraints for software projects. It is an essential part of software organization to deliver quality product, keeping the cost within client’s budget constrain and deliver the project as per scheduled. There are several factors, both internal and external, which may impact this triple constrain triangle. Any of three factors can severely impact the other two.

Therefore, software project management is essential to incorporate user requirements along with budget and time constraints.

3.2 Software Project Manager

A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

A project manager closely monitors the development process, prepares and executes various plans, arranges necessary and adequate resources, maintains communication among all team members in order to address issues of cost, budget, resources, time, quality and customer satisfaction.

Let us see few responsibilities that a project manager shoulders:

3.2.1 Managing People

- Act as project leader
- Liaison with stakeholders
- Managing human resources
- Setting up reporting hierarchy etc

3.2.2 Managing Project

- Defining and setting up project scope
- Managing project management activities
- Monitoring progress and performance
- Risk analysis at every phase
- Take necessary step to avoid or come out of problems
- Act as project spokesperson

3.3 Software Management Activities

Software project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management. Project management activities may include:

- Project Planning
- Scope Management
- Project Estimation

3.3.1 Project Planning

Software project planning is task, which is performed before the production of software actually starts. It is there for the software production but involves no concrete activity that has any direction connection with software production; rather it is a set of multiple processes, which facilitates software production. Project planning may include the following:

3.3.2 Scope Management

It defines the scope of project; this includes all the activities; process need to be done in order to make a deliverable software product. Scope management is essential because it creates boundaries of the project by clearly defining what would be done in the project and what would not be done. This makes project to contain limited and quantifiable tasks, which can easily be documented and in turn avoids cost and time overrun.

During Project Scope management, it is necessary to -

- Define the scope
- Decide its verification and control
- Divide the project into various smaller parts for ease of management.

- Verify the scope
- Control the scope by incorporating changes to the scope

3.3.3 Project Estimation

For an effective management accurate estimation of various measures is a must. With correct estimation managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

- **Software size estimation**

  Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices and Function points vary according to the user or software requirement.
- **Effort estimation**

  The managers estimate efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by managers' experience, organization's historical data or software size can be converted into efforts by using some standard formulae.
- **Time estimation**

  Once size and efforts are estimated, the time required to produce the software can be estimated. Efforts required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months.

  The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.
- **Cost estimation**

  This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider -
  - Size of software
  - Software quality
  - Hardware
  - Additional software or tools, licenses etc.

  - Skilled personnel with task-specific skills
  - Travel involved
  - Communication
  - Training and support

3.4 Project Estimation Techniques

We discussed various parameters involving project estimation such as size, effort, time and cost.

Project manager can estimate the listed factors using two broadly recognized techniques:

3.4 .1 Decomposition Technique

This technique assumes the software as a product of various compositions.

There are two main models -

- **Line of Code** Estimation is done on behalf of number of line of codes in the software product.
- **Function Points** Estimation is done on behalf of number of function points in the software product.

3.4.2 Empirical Estimation Technique

This technique uses empirically derived formulae to make estimation.These formulae are based on LOC or FPs.

- **Putnam Model**

  This model is made by Lawrence H. Putnam, which is based on Norden's frequency distribution (Rayleigh curve). Putnam model maps time and efforts required with software size.

- **COCOMO**

  COCOMO stands for COnstructive COst MOdel, developed by Barry W. Boehm. It divides the software product into three categories of software: organic, semi-detached and embedded.

3.5 Project Scheduling

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and arrange them keeping various

factors in mind. They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.
For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

3.6 Resource management

All elements used to develop a software product may be assumed as resource for that project. This may include human resource, productive tools and software libraries.
The resources are available in limited quantity and stay in the organization as a pool of assets. The shortage of resources hampers the development of project and it can lag behind the schedule. Allocating extra resources increases development cost in the end. It is therefore necessary to estimate and allocate adequate resources for the project.
Resource management includes -

- Defining proper organization project by creating a project team and allocating responsibilities to each team member
- Determining resources required at a particular stage and their availability
- Manage Resources by generating resource request when they are required and de-allocating them when they are no more needed.

3.7 Project Risk Management

Risk management involves all activities pertaining to identification, analyzing and making provision for predictable and non-predictable risks in the project. Risk may include the following:

- Experienced staff leaving the project and new staff coming in.
- Change in organizational management.
- Requirement change or misinterpreting requirement.

- Under-estimation of required time and resources.
- Technological changes, environmental changes, business competition.

3.7.1 Risk Management Process

There are following activities involved in risk management process:

- **Identification -** Make note of all possible risks, which may occur in the project.
- **Categorize -** Categorize known risks into high, medium and low risk intensity as per their possible impact on the project.
- **Manage -** Analyze the probability of occurrence of risks at various phases. Make plan to avoid or face risks. Attempt to minimize their side-effects.
- **Monitor -** Closely monitor the potential risks and their early symptoms. Also monitor the effects of steps taken to mitigate or avoid them.

3.8 Project Execution & Monitoring

In this phase, the tasks described in project plans are executed according to their schedules.

Execution needs monitoring in order to check whether everything is going according to the plan. Monitoring is observing to check the probability of risk and taking measures to address the risk or report the status of various tasks.

These measures include -

- **Activity Monitoring -** All activities scheduled within some task can be monitored on day-to-day basis. When all activities in a task are completed, it is considered as complete.
- **Status Reports -** The reports contain status of activities and tasks completed within a given time frame, generally a week. Status can be marked as finished, pending or work-in-progress etc.
- **Milestones Checklist -** Every project is divided into multiple phases where major tasks are performed (milestones) based on the phases of SDLC. This milestone checklist is prepared once every few weeks and reports the status of milestones.

3.9 Project Communication Management

Effective communication plays vital role in the success of a project. It bridges gaps between client and the organization, among the team members as well as other stake holders in the project such as hardware suppliers.

Communication can be oral or written. Communication management process may have the following steps:

- **Planning -** This step includes the identifications of all the stakeholders in the project and the mode of communication among them. It also considers if any additional communication facilities are required.
- **Sharing -** After determining various aspects of planning, manager focuses on sharing correct information with the correct person on correct time. This keeps everyone involved the project up to date with project progress and its status.
- **Feedback -** Project managers use various measures and feedback mechanism and create status and performance reports. This mechanism ensures that input from various stakeholders is coming to the project manager as their feedback.
- **Closure -** At the end of each major event, end of a phase of SDLC or end of the project itself, administrative closure is formally announced to update every stakeholder by sending email, by distributing a hardcopy of document or by other mean of effective communication.

After closure, the team moves to next phase or project.

3.10 Configuration Management

Configuration management is a process of tracking and controlling the changes in software in terms of the requirements, design, functions and development of the product.

IEEE defines it as “the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items”.

Generally, once the SRS is finalized there is less chance of requirement of changes from user. If they occur, the changes are addressed only with prior approval of higher management, as there is a possibility of cost and time overrun.

3.10.1 Baseline

A phase of SDLC is assumed over if it is baselined, i.e. baseline is a measurement that defines completeness of a phase. A phase is baselined when all activities pertaining to it are finished and well documented. If it was not the final phase, its output would be used in next immediate phase.

Configuration management is a discipline of organization administration, which takes care of occurrence of any change (process, requirement, technological, strategical etc.) after a phase is baselined. CM keeps check on any changes done in software.

3.10.2 Change Control

Change control is function of configuration management, which ensures that all changes made to software system are consistent and made as per organizational rules and regulations.

A change in the configuration of product goes through following steps -

- **Identification -** A change request arrives from either internal or external source. When change request is identified formally, it is properly documented.
- **Validation -** Validity of the change request is checked and its handling procedure is confirmed.
- **Analysis -** The impact of change request is analyzed in terms of schedule, cost and required efforts. Overall impact of the prospective change on system is analyzed.
- **Control -** If the prospective change either impacts too many entities in the system or it is unavoidable, it is mandatory to take approval of high authorities before change is incorporated into the system. It is decided if the change is worth incorporation or not. If it is not, change request is refused formally.
- **Execution -** If the previous phase determines to execute the change request, this phase take appropriate actions to execute the change, does a thorough revision if necessary.
- **Close request -** The change is verified for correct implementation and merging with the rest of the system. This newly incorporated change in the software is documented properly and the request is formally is closed.

3.11 Project Management Tools

The risk and uncertainty rises multifold with respect to the size of the project, even when the project is developed according to set methodologies.

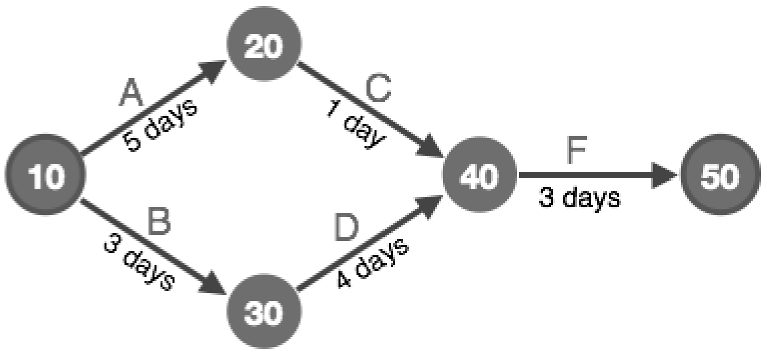
There are tools available, which aid for effective project management. A few are described:

3.11.1 Gantt Chart

Gantt charts was devised by Henry Gantt (1917). It represents project schedule with respect to time periods. It is a horizontal bar chart with bars representing activities and time scheduled for the project activities.

3.11.2 PERT Chart

PERT (Program Evaluation & Review Technique) chart is a tool that depicts project as network diagram. It is capable of graphically representing main events of project in both parallel and consecutive way. Events, which occur one after another, show dependency of the later event over the previous one.



Events are shown as numbered nodes. They are connected by labeled arrows depicting sequence of tasks in the project.

3.11.3 Resource Histogram

This is a graphical tool that contains bar or chart representing number of resources (usually skilled staff) required over time for a project event (or phase). Resource Histogram is an effective tool for staff planning and coordination.

3.11.4 Critical Path Analysis

This tool is useful in recognizing interdependent tasks in the project. It also helps to find out the shortest path or critical path to complete the project successfully. Like PERT diagram, each event is allotted a specific time frame. This tool shows dependency of event assuming an event can proceed to next only if the previous one is completed.

The events are arranged according to their earliest possible start time. Path between start and end node is critical path which cannot be further reduced and all events require to be executed in same order.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. List the characteristics of a software project
2. Explain software project
3. Explain the need for software project management
4 Give the detail description of the job of a software project manager.
5. Explain the terms: project planning, scope management and project estimation
6. Mention at least 3 project management tools

5.0 Conclusion

Software project management involves both software development skills and managerial skills. It is therefore imperative for project managers to acquire technical skills in software development such communication skill, requirement elicitation skill, specification writing skill, analysis skill, design skill, coding skill etc. And managerial skills such as leadership skill, cost estimation skill, scheduling skill etc.

6.0 Summary

This unit has highlighted need for software project management, the duties of Software Project Manager, Software Management Activities(i.e. Project Planning, Scope Management, Project Estimation) and Project Estimation Techniques. We have also discussed Project Scheduling, Resource management, Project Risk Management,

Execution and Monitoring, Project Communication Management, Configuration Management, Project Management Tools etc

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 4 Software Requirements

1.0 Introduction

The software requirements are description of features and functionalities of the target system. Requirements convey the expectations of users from the software product. The requirements can be obvious or hidden, known or unknown, expected or unexpected from client's point of view.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. List and explain the four steps in requirement engineering process
2. Depict the requirement elicitation process with a diagram

1. Mention at least 6 requirement elicitation techniques
2. List at least 10 software requirement characteristics
3. Differentiate between functional and non-functional software requirements
4. Mention at 10 user interface requirements
5. Outline the responsibility of a system analyst
6. Differentiate between software metric and software measures

3.0 Main Content

3.1 Requirement Engineering

The process to gather the software requirements from client, analyze and document them is known as requirement engineering.

The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

3.1.1 Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

Let us see the process briefly -

Feasibility study

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

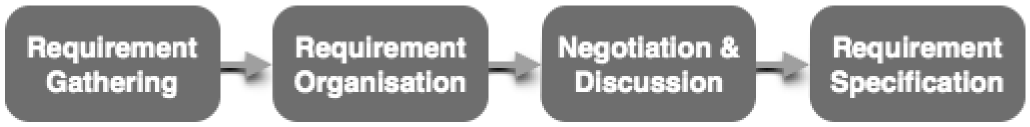
3.2 Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

3.3 Requirement Elicitation Process

Requirement elicitation process can be depicted using the following diagram:



- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.
- **Organizing Requirements -** The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.
  The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.
- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

3.4 Requirement Elicitation Techniques

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.

There are various ways to discover requirements

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- Oral interviews
- Written interviews
- One-to-one interviews which are held between two persons across the table.
- Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.
A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

Task analysis

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

Domain Analysis

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps in giving better idea of requirements. If there is no software installed at client’s end for developer’s reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.

Observation

Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

3.5 Software Requirements Characteristics

Gathering software requirements is the foundation of the entire software development project. Hence, they must be clear, correct and well-defined.
A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent
- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

3.6 Software Requirements

We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.
Broadly software requirements should be categorized in two categories:

3.6.1 Functional Requirements

Requirements, which are related to functional aspect of software fall into this category.
They define functions and functionality within and from the software system.

Examples -

- Search option given to user to search from various invoices.
- User should be able to mail any report to management.

- Users can be divided into groups and groups can be given separate rights.
- Should comply business rules and administrative functions.
- Software is developed keeping downward compatibility intact.

3.6.2 Non-Functional Requirements

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software, which users make assumption of.

Non-functional requirements include -

- Security
- Logging
- Storage
- Configuration
- Performance
- Cost
- Interoperability
- Flexibility
- Disaster recovery
- Accessibility

Requirements are categorized logically as

- **Must Have**: Software cannot be said operational without them.
- **Should have**: Enhancing the functionality of software.
- **Could have**: Software can still properly function with these requirements.
- **Wish list**: These requirements do not map to any objectives of software.

While developing software, 'Must have' must be implemented, 'Should have' is a matter of debate with stakeholders and negation, whereas 'could have' and 'wish list' can be kept for software updates.

3.7 User Interface requirements

UI is an important part of any software or hardware or hybrid system. A software is widely accepted if it is -

- easy to operate
- quick in response
- effectively handling operational errors
- providing simple yet consistent user interface

User acceptance majorly depends upon how user can use the software. UI is the only way for users to perceive the system. A well performing software system must also be equipped with attractive, clear, consistent and responsive user interface. Otherwise, the functionalities of software system cannot be used in convenient way. A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below:

- Content presentation
- Easy Navigation
- Simple interface
- Responsive
- Consistent UI elements
- Feedback mechanism
- Default settings
- Purposeful layout
- Strategical use of colour and texture.
- Provide help information
- User centric approach
- Group based view settings.

3.8 Software System Analyst

System analyst in an IT organization is a person, who analyzes the requirement of proposed system and ensures that requirements are conceived and documented properly & correctly. Role of an analyst starts during Software Analysis Phase of SDLC. It is the responsibility of analyst to make sure that the developed software meets the requirements of the client.

System Analysts have the following responsibilities:

- Analyzing and understanding requirements of intended software
- Understanding how the project will contribute in the organization objectives
- Identify sources of requirement
- Validation of requirement
- Develop and implement requirement management plan
- Documentation of business, technical, process and product requirements
- Coordination with clients to prioritize requirements and remove and ambiguity
- Finalizing acceptance criteria with client and other stakeholders

3.9 Software Metrics and Measures

Software Measures can be understood as a process of quantifying and symbolizing various attributes and aspects of software.

Software Metrics provide measures for various aspects of software process and software product.

Software measures are fundamental requirement of software engineering. They not only help to control the software development process but also aid to keep quality of ultimate product excellent.

According to Tom DeMarco, a (Software Engineer), “You cannot control what you cannot measure.” By his saying, it is very clear how important software measures are.

Let us see some software metrics:

- **Size Metrics -** LOC (Lines of Code), mostly calculated in thousands of delivered source code lines, denoted as KLOC.

  Function Point Count is measure of the functionality provided by the software. Function Point count defines the size of functional aspect of software.
- **Complexity Metrics -** McCabe’s Cyclomatic complexity quantifies the upper bound of the number of independent paths in a program, which is perceived as complexity of the program or its modules. It is represented in terms of graph theory concepts by using control flow graph.
- **Quality Metrics -** Defects, their types and causes, consequence, intensity of severity and their implications define the quality of product.

  The number of defects found in development process and number of defects reported by the client after the product is installed or delivered at client-end, define quality of product.
- **Process Metrics -** In various phases of SDLC, the methods and tools used, the company standards and the performance of development are software process metrics.
- **Resource Metrics -** Effort, time and various resources used, represents metrics for resource measurement.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Explain the four steps in requirement engineering process
2. Illustrate requirement elicitation process with a diagram only.

3. List at least 6 requirement elicitation techniques
4. Name at least 10 software requirement characteristics
5. Differentiate between functional and non-functional software requirements
6. Explain the user interface requirements
7. Outline the responsibility of a system analyst
8. Differentiate between software metric and software measures

5.0 Conclusion

Software requirements specify the needs or expectation of the user or client. They are captured through the process of elicitation. Both functional and non-functional requirements are captured or elicited. This involves the interaction between the user or client and the System Analyst and/or the development team.

6.0 Summary

In this unit we discussed the following:

- *Requirement Engineering Process*
- Software Requirement Validation
- Requirement Elicitation Process
- Requirement Elicitation Techniques

- *Software Requirements Characteristics*
- *Software Requirements*
- *User Interface requirements*
- *Software System Analyst*
- *Software Metrics and Measures*

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: “Software Engineering”- A Practitioner’s Approach”, McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

MODULE 7: OVERVIEW OF SOFTWARE DESIGN, ANALYSIS AND DESIGN TOOLS, DESIGN STRATEGIES AND USER INTERFACE BASICS

1.0 Introduction

Software design involves describing, conceptually, a software solution that meets the requirements of the problem. Before proffering solution, the problem must be analysed adequately to have good understanding of the problem. The intent is to solve the problem, that is, the requirement in context, with validation as the means to check that understanding.

Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Software design yields three levels of results. Mention and briefly describe them
2. Discuss modularization and state its advantages in software development
3. Differentiate between cohesion and coupling in software
4. List and explain any 5 types of cohesion

3.0 Main Content

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfil the requirements mentioned in SRS.

3.1 Software Design Levels

Software design yields three levels of results:

- **Architectural Design -** The architectural design is the highest abstract version of the system. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.

- **High-level Design-** The high-level design breaks the ‘single entity-multiple component’ concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
- **Detailed Design-** Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

3.2 Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

3.2.1 Advantages of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

3.3 Concurrency

Back in time, all software are meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs along side the word processor itself.

3.4 Coupling and Cohesion

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

3.4.1 Cohesion

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

- **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
- **Logical cohesion -** When logically categorized elements are put together into a module, it is called logical cohesion.
- **Temporal Cohesion -** When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.
- **Procedural cohesion -** When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

- **Communicational cohesion -** When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.
- **Sequential cohesion -** When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.
- **Functional cohesion -** It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

3.4.2 Coupling

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely:

- **Content coupling -** When a module can directly access or modify or refer to the content of another module, it is called content level coupling.
- **Common coupling-** When multiple modules have read and write access to some global data, it is called common or global coupling.
- **Control coupling-** Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.
- **Stamp coupling-** When multiple modules share common data structure and work on different part of it, it is called stamp coupling.
- **Data coupling-** Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

3.5 Design Verification

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements.

The next phase, which is the implementation of software, depends on all outputs mentioned above.

It then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.
By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Mention and briefly describe the result of software development.
2. Discuss modularization and state its advantages in software development
3. Differentiate between cohesion and coupling in software
4. List and explain any five types of cohesion
5. Explain the activities involved in software design verification.

5.0 Conclusion

Software design is the art of finding solution to business problem(s). This in three different levels, namely: architectural design, high level design and detailed design. The design is carried out in modules which performs simple function. The interactions between and within modules are design with coupling and cohesion in mind.

6.0 Summary

In this unit we discussed the following:

- Software Design Levels
- Modularization
- Concurrency
- Coupling and Cohesion
- Design Verification

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 2: Analysis and Design tools

1.0 Introduction

Analysis **involves understanding the problem which the software is intended to solve** it while design is the solution to problem. Software analysis and design tools are tools used to convert requirement specifications into a software product. As the name implies, they are used for both analysis and design. We shall be discussing some of these tools in this unit.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Explain what data flow is
2. Describe the following: Logical DFD, Physical DFD
3. Describe the components of DFD with their corresponding symbols
4. Differentiate between a data flow and control flow in a structure chart
5. Compare and contrast between HIPO and IPO
6. State the steps needed to create a decision table
7. List the content of a data dictionary

3.0 Main Content

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.

Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.

Let us see few analysis and design tools used by software designers:

3.1 Data Flow Diagram

Data flow diagram is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

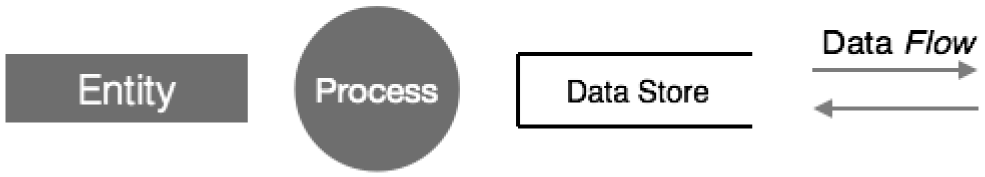
3.1.1 Types of DFD

Data Flow Diagrams are either Logical or Physical.

- **Logical DFD -** This type of DFD concentrates on the system process, and flow of data in the system. For example, in a Banking software system, how data is moved between different entities.
- **Physical DFD -** This type of DFD shows how the data flow is actually implemented in the system. It is more specific and closer to the implementation.

DFD Components

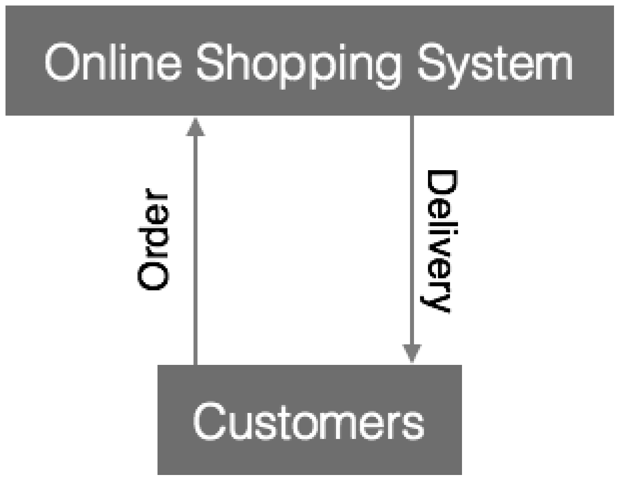
DFD can represent Source, destination, storage and flow of data using the following set of components -



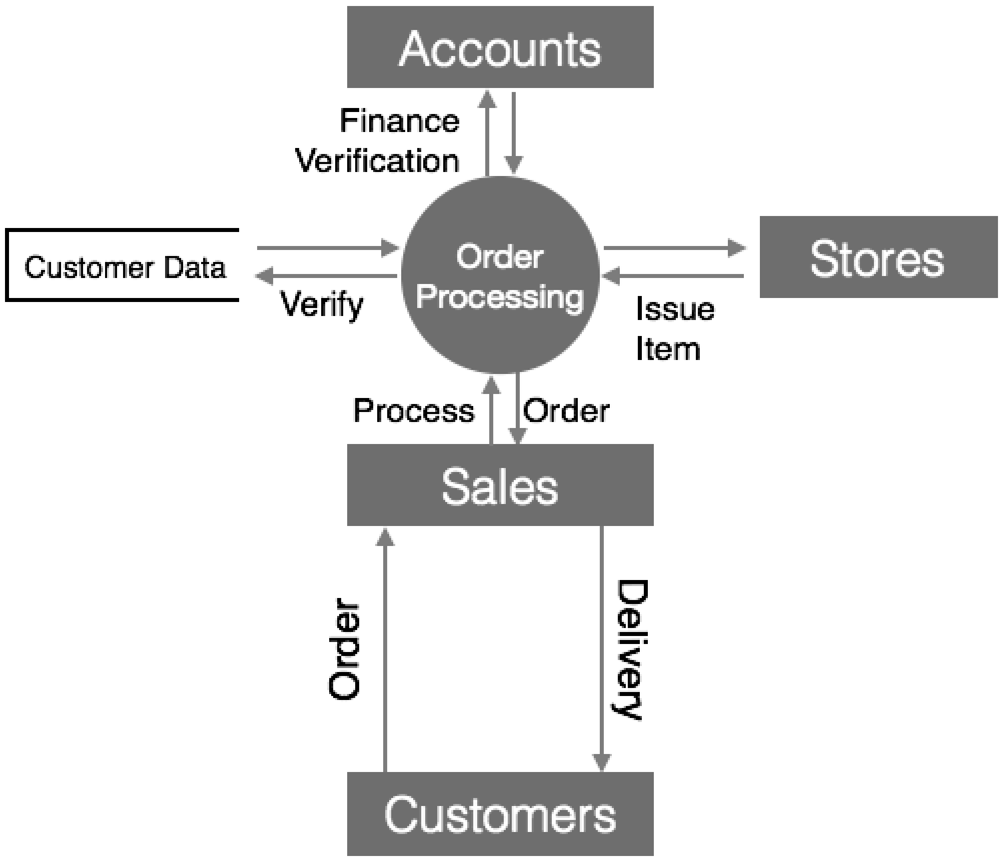
- **Entities -** Entities are source and destination of information data. Entities are represented by rectangles with their respective names.
- **Process -** Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage -** There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **Data Flow -** Movement of data is shown by pointed arrows. Data movement is shown from the base of arrow as its source towards head of the arrow as destination.

3.1.2 Levels of DFD

- **Level 0 -** Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.



- **Level 1 -** The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1 DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.



- **Level 2 -** At this level, DFD shows how data flows inside the modules mentioned in Level 1.

Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

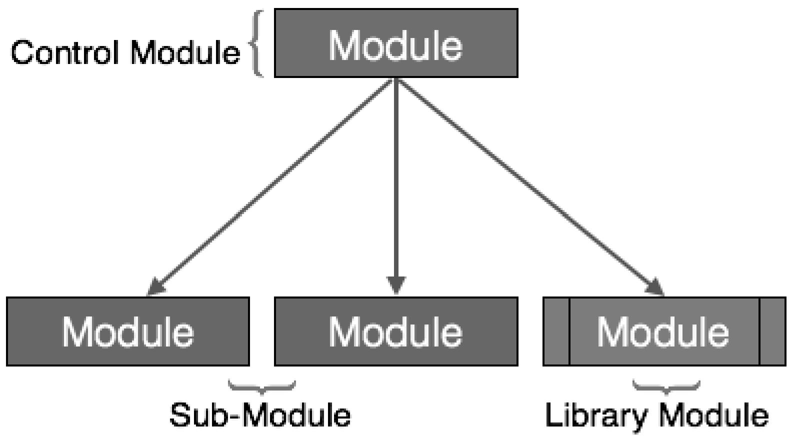
3.2 Structure Charts

Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.

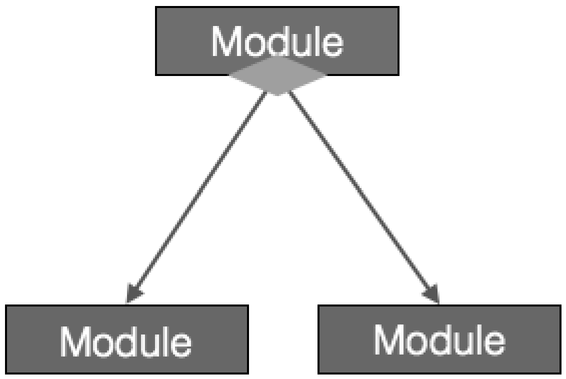
Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

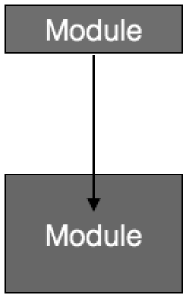
- **Module -** It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invokable from any module.



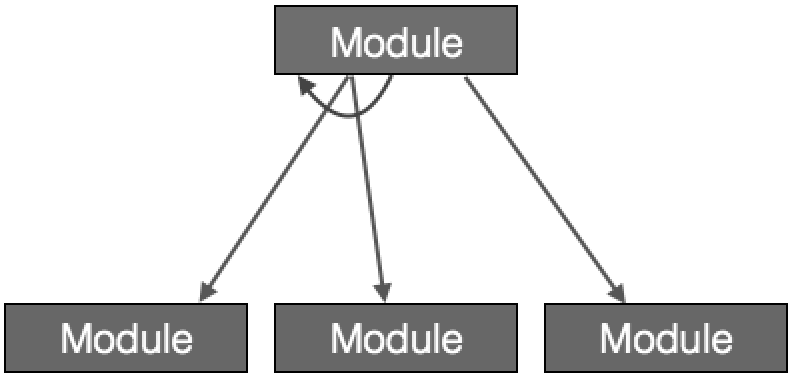
- **Condition -** It is represented by small diamond at the base of module. It depicts that control module can select any of sub-routine based on some condition.



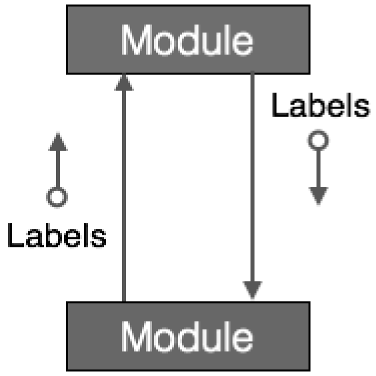
- **Jump -** An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.



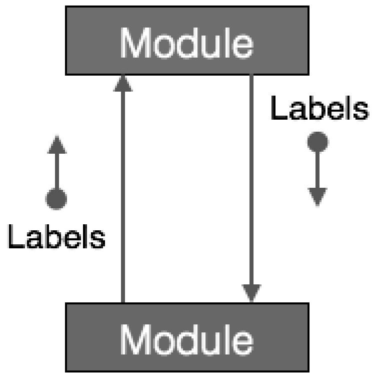
- **Loop -** A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



- **Data flow -** A directed arrow with empty circle at the end represents data flow.



- **Control flow -** A directed arrow with filled circle at the end represents control flow.

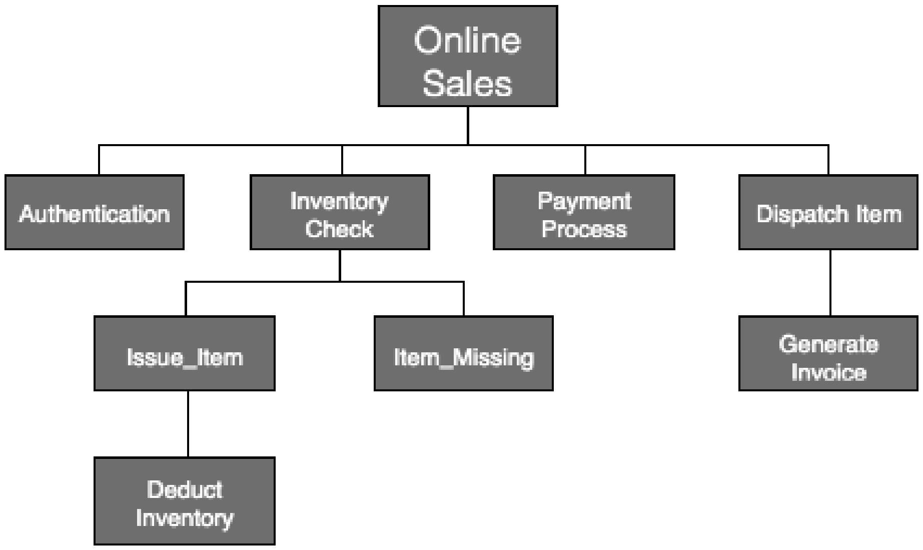


3.3 HIPO Diagram

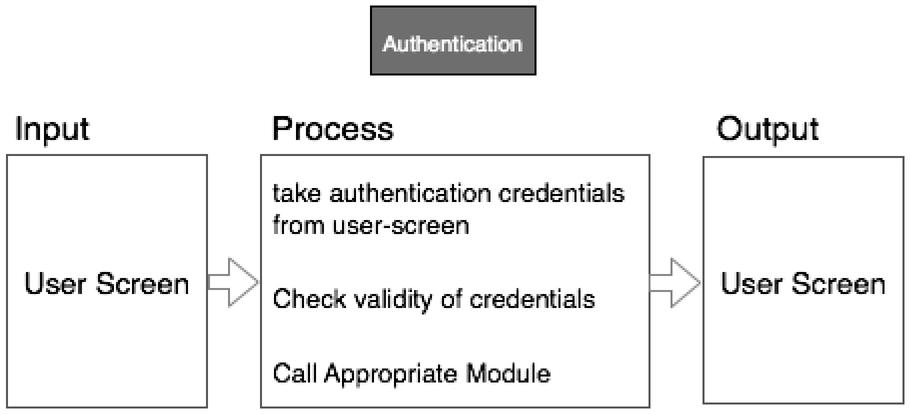
HIPO (Hierarchical Input Process Output) diagram is a combination of two organized method to analyze the system and provide the means of documentation. HIPO model was developed by IBM in year 1970.

HIPO diagram represents the hierarchy of modules in the software system. Analyst uses HIPO diagram in order to obtain high-level view of system functions. It decomposes functions into sub-functions in a hierarchical manner. It depicts the functions performed by system.

HIPO diagrams are good for documentation purpose. Their graphical representation makes it easier for designers and managers to get the pictorial idea of the system structure.



In contrast to IPO (Input Process Output) diagram, which depicts the flow of control and data in a module, HIPO does not provide any information about data flow or control flow.


Authentication
Input
Process
Output
User Screen
take authentication credentials from user-screen
Check validity of credentials
Call Appropriate Module
User Screen

Example

Both parts of HIPO diagram, Hierarchical presentation and IPO Chart are used for structure design of software program as well as documentation of the same.

3.4 Structured English

Most programmers are unaware of the large picture of software so they only rely on what their managers tell them to do. It is the responsibility of higher software management to provide accurate information to the programmers to develop accurate yet fast code.

Other forms of methods, which use graphs or diagrams, may are sometimes interpreted differently by different people.

Hence, analysts and designers of the software come up with tools such as Structured English. It is nothing but the description of what is required to code and how to code it. Structured English helps the programmer to write error-free code.

Other form of methods, which use graphs or diagrams, may are sometimes interpreted differently by different people. Here, both Structured English and Pseudo-Code tries to mitigate that understanding gap.

Structured English uses plain English words in structured programming paradigm. It is not the ultimate code but a kind of description what is required to code and how to code it. The following are some tokens of structured programming.

```
IF-THEN-ELSE,
DO-WHILE-UNTIL
```

Analyst uses the same variable and data name, which are stored in Data Dictionary, making it much simpler to write and understand the code.

Example

We take the same example of Customer Authentication in the online shopping environment. This procedure to authenticate customer can be written in Structured English as:

```
Enter Customer_Name
SEEK Customer_Name in Customer_Name_DB file
IF Customer_Name found THEN
   Call procedure USER_PASSWORD_AUTHENTICATE()
ELSE
   PRINT error message
   Call procedure NEW_CUSTOMER_REQUEST()
ENDIF
```

The code written in Structured English is more like day-to-day spoken English. It cannot be implemented directly as a code of software. Structured English is independent of programming language.

3.5 Pseudo-Code

Pseudo code is written closer to programming language. It may be considered as augmented programming language, full of comments and descriptions.

Pseudo code avoids variable declaration but they are written using some actual programming language's constructs, like C, Fortran, Pascal etc.

Pseudo code contains more programming details than Structured English. It provides a method to perform the task, as if a computer is executing the code.

Example

Program to print Fibonacci up to n numbers.

```
void function Fibonacci
Get value of n;
Set value of a to 1;
Set value of b to 1;
Initialize I to 0
for (i=0; i< n; i++)
{
   if a greater than b
```

```
  {
    Increase b by a;
    Print b;
  }
  else if b greater than a
  {
    increase a by b;
    print a;
  }
}
```

3.6 Decision Tables

A Decision table represents conditions and the respective actions to be taken to address them, in a structured tabular format.

It is a powerful tool to debug and prevent errors. It helps group similar information into a single table and then by combining tables it delivers easy and convenient decision-making.

3.6.1 Creating Decision Table

To create the decision table, the developer must follow basic four steps:

1. Identify all possible conditions to be addressed
2. Determine actions for all identified conditions
3. Create Maximum possible rules
4. Define action for each rule

Decision Tables should be verified by end-users and can lately be simplified by eliminating duplicate rules and actions.

Example

Let us take a simple example of day-to-day problem with our Internet connectivity. We begin by identifying all problems that can arise while starting the internet and their respective possible solutions.

We list all possible problems under column conditions and the prospective actions under column Actions.

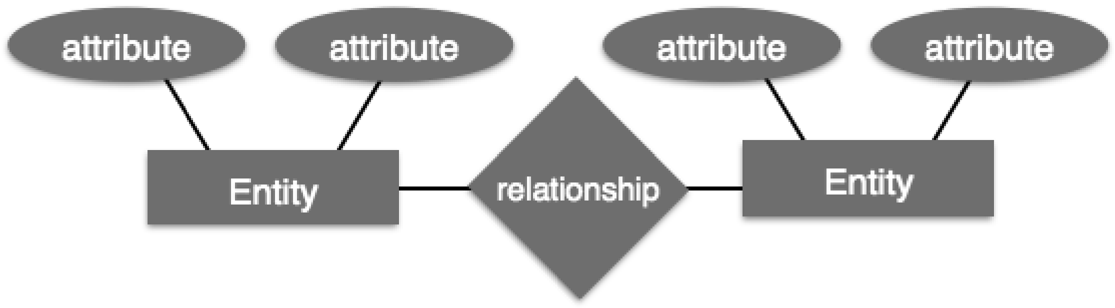
Table: Decision Table – In-house Internet Troubleshooting

	Conditions/Actions	**Rules**

Conditions	Shows Connected	N	N	N	N	Y	Y	Y	Y
	Ping is Working	N	N	Y	Y	N	N	Y	Y
	Opens Website	Y	N	Y	N	Y	N	Y	N
Actions	Check network cable	X							
	Check internet router	X				X	X	X	
	Restart Web Browser							X	
	Contact Service provider		X	X	X	X	X	X	
	Do no action								

3.8 Entity-Relationship Model

Entity-Relationship model is a type of database model based on the notion of real-world entities and relationship among them. We can map real world scenario onto ER database model. ER Model creates a set of entities with their attributes, a set of constraints and relation among them.



ER Model is best used for the conceptual design of database. ER Model can be represented as follows:

- **Entity -** An entity in ER Model is a real world being, which has some properties called ***attributes***. Every attribute is defined by its corresponding set of values, called ***domain***.
  For example, Consider a school database. Here, a student is an entity. Student has various attributes like name, id, age and class etc.

- **Relationship -** The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of associations between two entities.

  Mapping cardinalities:
    - one to one
    - one to many
    - many to one
    - many to many

3.7 Data Dictionary

Data dictionary is the centralized collection of information about data. It stores meaning and origin of data, its relationship with other data, data format for usage etc. Data dictionary has rigorous definitions of all names in order to facilitate user and software designers.
Data dictionary is often referenced as meta-data (data about data) repository. It is created along with DFD (Data Flow Diagram) model of software program and is expected to be updated whenever DFD is changed or updated.

3.7.1 Requirement of Data Dictionary

The data is referenced via data dictionary while designing and implementing software. Data dictionary removes any chances of ambiguity. It helps keeping work of programmers and designers synchronized while using same object reference everywhere in the program.
Data dictionary provides a way of documentation for the complete database system in one place. Validation of DFD is carried out using data dictionary.

3.7.2 Contents

Data dictionary should contain information about the following

- Data Flow
- Data Structure
- Data Elements
- Data Stores
- Data Processing

Data Flow is described by means of DFDs as studied earlier and represented in algebraic form as described.

=	**Composed of**
{}	Repetition
()	Optional
+	And
[/]	Or

Example

Address = House No + (Street / Area) + City + State

Course ID = Course Number + Course Name + Course Level + Course Grades

3.8 Data Elements

Data elements consist of Name and descriptions of Data and Control Items, Internal or External data stores etc. with the following details:

- Primary Name
- Secondary Name (Alias)
- Use-case (How and where to use)
- Content Description (Notation etc.)
- Supplementary Information (preset values, constraints etc.)

3.8 Data Store

It stores the information from where the data enters into the system and exists out of the system. The Data Store may include -

- **Files**
  - Internal to software.
  - External to software but on the same machine.
  - External to software and system, located on different machine.
- **Tables**
  - Naming convention
  - Indexing property

3.9 Data Processing

There are two types of Data Processing:

- **Logical:** As user sees it
- **Physical:** As software sees it

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Illustrate data flow diagram using diagram only.
2. Explain the following: Logical DFD, Physical DFD
3. Describe the components of DFD with their corresponding symbols
4. Differentiate between a data flow and control flow in a structure chart
5. Compare and contrast between HIPO and IPO
8. State the steps needed to create a decision table
9. List the content of a data dictionary

5.0 Conclusion

As stated in the previous module software design is concerned with finding or proffering solution to business problem(s). To achieve this feat, the designer will need some software design tools. These tools include among others: Data Flow Diagram, Structure Charts, HIPO Diagram, Structured English, Pseudo-Code

6.0 Summary

In this unit we discussed the following:

- Data Flow Diagram
- Structure Charts
- HIPO Diagram
- Structured English
- Pseudo-Code
- Decision Tables
- Data Dictionary

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: “Software Engineering”- A Practitioner’s Approach”, McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 3: Software Design Strategies

1.0 Introduction

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Mention and discuss difference types of software design
2. List and explain the different concepts of object-oriented design
3. Mention and discuss two generic approaches for software design

3.0 Main Content

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

3.1 Structured Design

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

Cohesion - grouping of all functionally related elements.

Coupling - communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

3.2 Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

3.3 Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object-Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together and this is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

3.4 Design Process

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet it may have the following steps involved:

- A solution design is created from requirement or previously used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- Application framework is defined.

3.5 Software Design Approaches

Here are two generic approaches for software designing:

3.5.1 Top-Down Design

We know that a system is composed of more than one sub-system and it contains a number of components. Further, these sub-systems and components may have their onset of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

3.5.2 Bottom-up Design

The bottom-up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower-level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Explain the types of software design
2. Explain the different concepts of object-oriented design
3. Mention and discuss two generic approaches for software design
4. Describe the bottom-up design approach.
5. Describe the fundamental concepts of object-oriented design.

5.0 Conclusion

It is pertinent to note that in the design of software certain strategies need to be applied. Some of these strategies include: Structured Design, Function Oriented Design, Object Oriented Design, Design Process, Software Design Approaches (Top-down Design, Bottom-up Design).

6.0 Summary

In this unit we discussed the following:

- Structured Design
- Function Oriented Design
- Object Oriented Design
- Design Process
- Software Design Approaches

7.0 Further

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: “Software Engineering”- A Practitioner’s Approach”, McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 4: Software User Interface Design

1.0 Introduction

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Mention some qualities of a user interface that make a software more popular
2. Give a broad classification of user interface
3. Mention and explain 3 elements of a text-based command line interface
4. Briefly describe graphical user interface
5. Mention at least 5 Application specific GUI components
6. State at least 5 user Interface Golden rules

3.0 Main Content

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

3.1 Broad Classification of User Interface

UI is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

3.1.1 Command Line Interface (CLI)

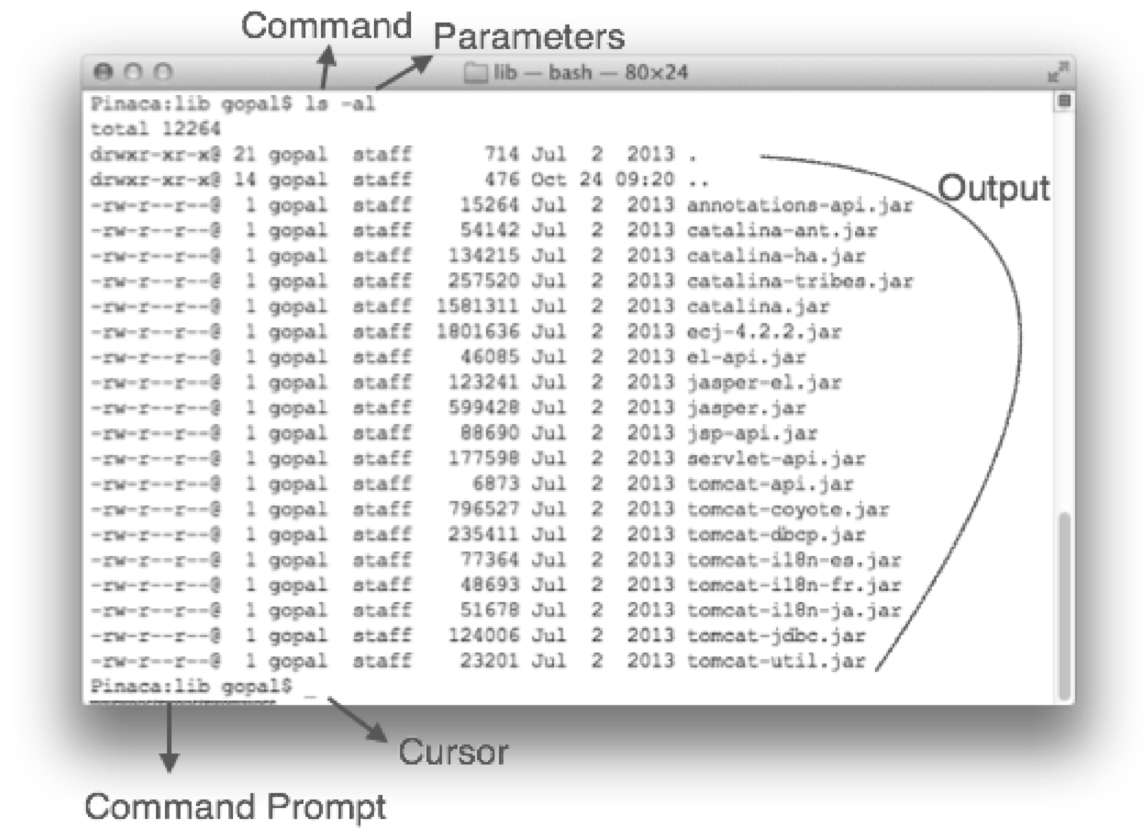
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

CLI Elements



```
Pinaca:lib gopal$ ls -al
total 12264
drwxr-xr-x@ 21 gopal  staff      714 Jul  2  2013 .
drwxr-xr-x@ 14 gopal  staff      476 Oct 24 09:20 ..
-rw-r--r--@  1 gopal  staff    15264 Jul  2  2013 annotations-api.jar
-rw-r--r--@  1 gopal  staff    54142 Jul  2  2013 catalina-ant.jar
-rw-r--r--@  1 gopal  staff   134215 Jul  2  2013 catalina-ha.jar
-rw-r--r--@  1 gopal  staff   257520 Jul  2  2013 catalina-tribes.jar
-rw-r--r--@  1 gopal  staff  1581311 Jul  2  2013 catalina.jar
-rw-r--r--@  1 gopal  staff  1801636 Jul  2  2013 ecj-4.2.2.jar
-rw-r--r--@  1 gopal  staff    46085 Jul  2  2013 el-api.jar
-rw-r--r--@  1 gopal  staff   123241 Jul  2  2013 jasper-el.jar
-rw-r--r--@  1 gopal  staff   599428 Jul  2  2013 jasper.jar
-rw-r--r--@  1 gopal  staff    88690 Jul  2  2013 jsp-api.jar
-rw-r--r--@  1 gopal  staff   177598 Jul  2  2013 servlet-api.jar
-rw-r--r--@  1 gopal  staff     6873 Jul  2  2013 tomcat-api.jar
-rw-r--r--@  1 gopal  staff   796527 Jul  2  2013 tomcat-coyote.jar
-rw-r--r--@  1 gopal  staff   235411 Jul  2  2013 tomcat-dbcp.jar
-rw-r--r--@  1 gopal  staff    77364 Jul  2  2013 tomcat-i18n-es.jar
-rw-r--r--@  1 gopal  staff    48693 Jul  2  2013 tomcat-i18n-fr.jar
-rw-r--r--@  1 gopal  staff    51678 Jul  2  2013 tomcat-i18n-ja.jar
-rw-r--r--@  1 gopal  staff   124006 Jul  2  2013 tomcat-jdbc.jar
-rw-r--r--@  1 gopal  staff    23201 Jul  2  2013 tomcat-util.jar
Pinaca:lib gopal$ _
```

A text-based command line interface can have the following elements:

- **Command Prompt -** It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.

- **Cursor -** It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- **Command -** A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

3.1.2 Graphical User Interface

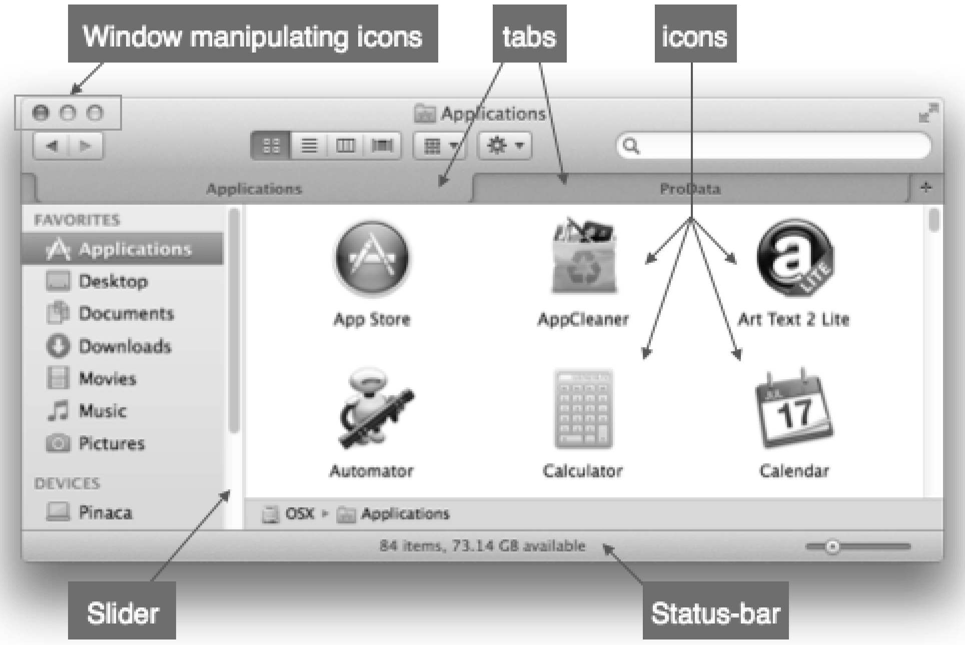
Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

3.2 GUI Elements

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:



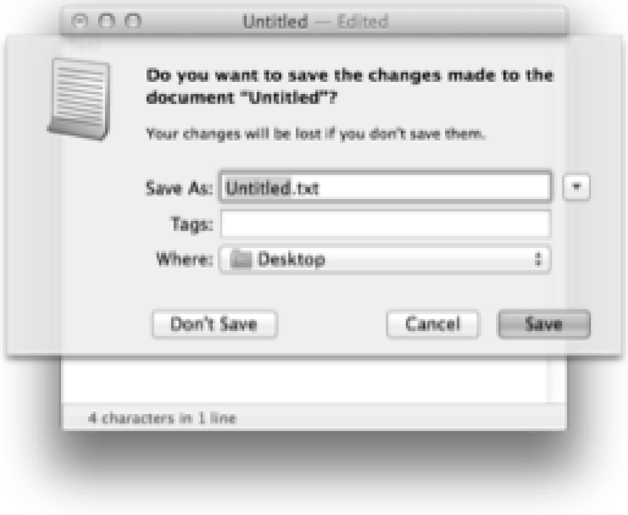
- **Window -** An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.

- **Tabs -** If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.
- **Menu -** Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.
- **Icon -** An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.
- **Cursor -** Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

3.3 Application specific GUI components

A GUI of an application contains one or more of the listed GUI elements:

- **Application Window -** Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
- **Dialogue Box -** It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.


Untitled — Edited

Do you want to save the changes made to the document "Untitled"?

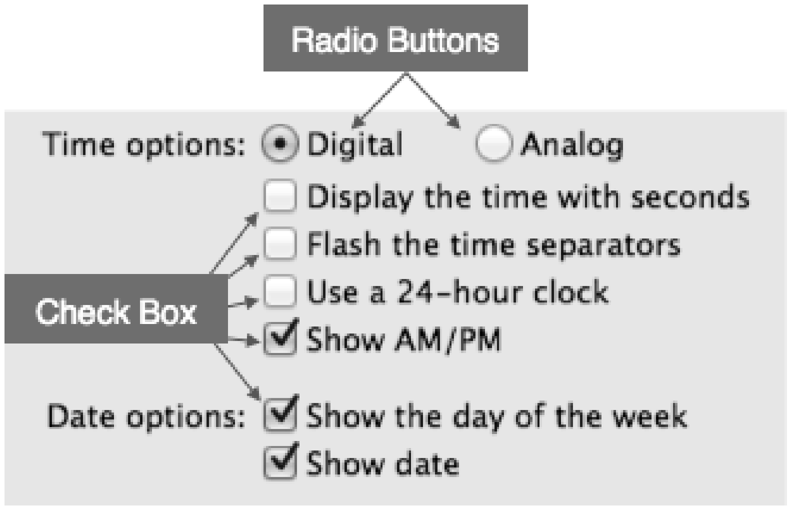
Your changes will be lost if you don't save them.

Save As: Untitled.txt

Tags:

Where: Desktop

- **Text-Box -** Provides an area for user to type and enter text-based data.
- **Buttons -** They imitate real life buttons and are used to submit inputs to the software.



Time options: [✓ Digital] Analog

Display the time with seconds

Flash the time separators

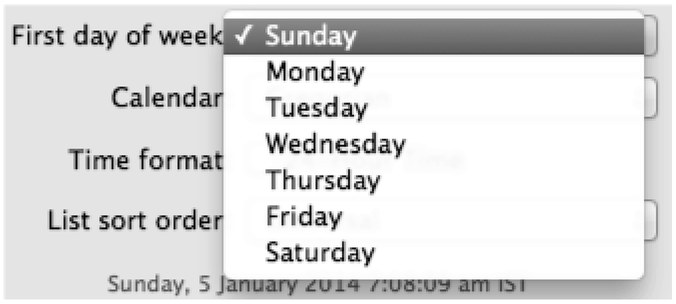
Use a 24-hour clock

[✓ Show AM/PM]

Date options: [✓ Show the day of the week]

[✓ Show date]

- **Radio-button -** Displays available options for selection. Only one can be selected among all offered.
- **Check-box -** Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
- **List-box -** Provides list of available items for selection. More than one item can be selected.

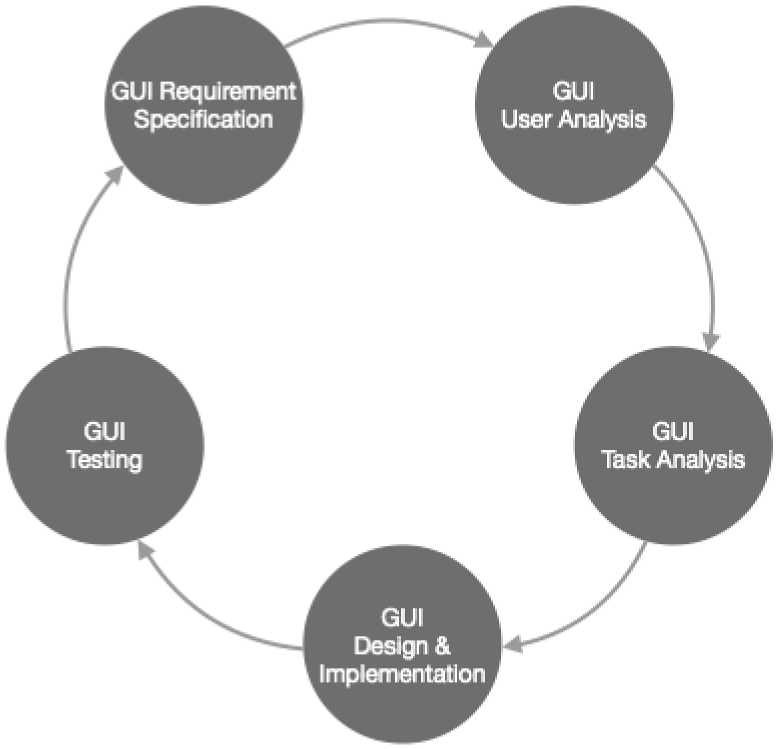


3.4 Other impressive GUI components are:

- Sliders
- Combo-box
- Data-grid
- Drop-down list

3.5 User Interface Design Activities

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.



A model used for GUI design and development should fulfil these GUI specific steps.

- **GUI Requirement Gathering -** The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
- **User Analysis -** The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
- **Task Analysis -** Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
- **GUI Design & implementation -** Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.
- **Testing -** GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

3.6 GUI Implementation Tools

There are several tools available using which the designers can create entire GUI on a mouse click. Some tools can be embedded into the software environment (IDE).

GUI implementation tools provide powerful array of GUI controls. For software customization, designers can change the code accordingly.

There are different segments of GUI tools according to their different use and platform.

Example

Mobile GUI, Computer GUI, Touch-Screen GUI etc. Here is a list of few tools which come handy to build GUI:

- FLUID
- AppInventor (Android)
- LucidChart

- Wavemaker
- Visual Studio

3.7 User Interface Golden rules

The following rules are mentioned to be the golden rules for GUI design, described by Shneiderman and Plaisant in their book (Designing the User Interface).

- **Strive for consistency -** Consistent sequences of actions should be required in similar situations. Identical terminology should be used in prompts, menus, and help screens. Consistent commands should be employed throughout.
- **Enable frequent users to use short-cuts -** The user's desire to reduce the number of interactions increases with the frequency of use. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.
- **Offer informative feedback -** For every operator action, there should be some system feedback. For frequent and minor actions, the response must be modest, while for infrequent and major actions, the response must be more substantial.
- **Design dialog to yield closure -** Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and this indicates that the way ahead is clear to prepare for the next group of actions.
- **Offer simple error handling -** As much as possible, design the system so the user will not make a serious error. If an error is made, the system should be able to detect it and offer simple, comprehensible mechanisms for handling the error.
- **Permit easy reversal of actions -** This feature relieves anxiety, since the user knows that errors can be undone. Easy reversal of actions encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.
- **Support internal locus of control -** Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

- **Reduce short-term memory load -** The limitation of human information processing in short-term memory requires the displays to be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Mention some qualities of a user interface that make a software more popular
2. Give a broad classification of user interface
3. Mention and explain 3 elements of a text-based command line interface
4. Briefly describe the tools used for development of graphical user interface
5. Mention at least 5 Application specific GUI components

5.0 Conclusion

User interface is the means through which a user (Operator) interact with the computer (software). UI provides a platform for the user to manipulate the software. It becomes imperative that while developing a software, the user interface must of necessity be design and incorporated into the greater whole. The user interface could be command line based or graphical.

6.0 Summary

In this unit we discussed the following:

- Broad Classification of User Interface
- GUI Elements
- Application specific GUI components
- Other impressive GUI components are
- User Interface Design Activities
- GUI Implementation Tools
- User Interface Golden rules

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

MODULE8:OVERVIEW OF DESIGN COMPLEXITY, SOFTWARE IMPLEMENTATION, TESTING, MAINTENANCE AND CASE TOOLS

1.0 Introduction

The term complexity stands for state of events or things, which have multiple interconnected links and highly complicated structures. In software programming, as the design of software is realized, the number of elements and their interconnections gradually emerge to be huge, which becomes too difficult to understand at once.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- *Discuss*
    - *Halstead's Complexity Measures*
    - *Cyclomatic Complexity Measures*
    - *Function Point*
- State the formula for each of the following meaning of each parameter:
    - *Halstead's Complexity Measures*
    - *Cyclomatic Complexity Measures*
    - *Function Point*
- Mention any 3 parameter of function point
- Mention any 10 characteristics for system Description

3.0 Main Content

The term complexity stands for state of events or things, which have multiple interconnected links and highly complicated structures. In software programming, as the design of software is realized, the number of elements and their interconnections gradually emerge to be huge, which becomes too difficult to understand at once.

Software design complexity is difficult to assess without using complexity metrics and measures. Let us see three important software complexity measures.

3.1 Halstead's Complexity Measures

In 1977, Mr. Maurice Howard Halstead introduced metrics to measure software complexity. Halstead's metrics depends upon the actual implementation of program and its measures, which are computed directly from the operators and operands from source code, in static manner. It allows to evaluate testing time, vocabulary, size, difficulty, errors, and efforts for C/C++/Java source code.

According to Halstead, "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operands". Halstead metrics think a program as sequence of operators and their associated operands.

He defines various indicators to check complexity of module.

Parameter	**Meaning**
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

When we select source file to view its complexity details in Metric Viewer, the following result is seen in Metric Report:

Metric	**Meaning**	**Mathematical Representation**
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume
B	Errors	Volume / 3000
T	Testing time	Time = Efforts / S, where S=18 seconds.

3.2 Cyclomatic Complexity Measures

Every program encompasses statements to execute in order to perform some task and other decision-making statements that decide, what statements need to be executed. These decision-making constructs change the flow of the program.

If we compare two programs of same size, the one with more decision-making statements will be more complex as the control of program jumps frequently.

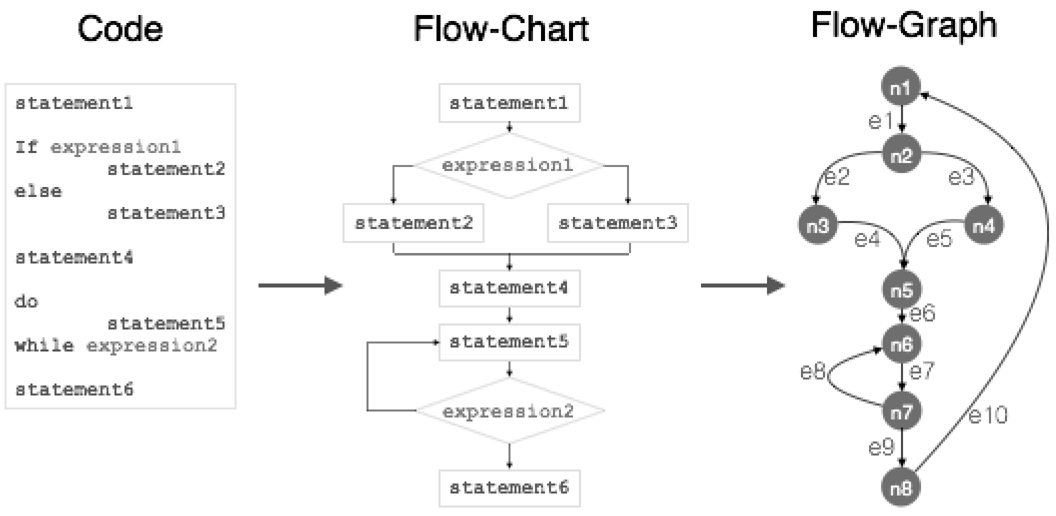
McCabe, in 1976, proposed Cyclomatic Complexity Measure to quantify complexity of a given software. It is graph driven model that is based on decision-making constructs of program such as if-else, do-while, repeat-until, switch-case and goto statements.

Process to make flow control graph:

- Break program in smaller blocks, delimited by decision-making constructs.
- Create nodes representing each of these nodes.
- Connect nodes as follows:
    - If control can branch from block i to block j
      Draw an arc
    - From exit node to entry node
      Draw an arc.

To calculate Cyclomatic complexity of a program module, we use the formula

V(G)= e – n +2 Where e is total number of edges n is total number of nodes



The Cyclomatic complexity of the above module is

e =10

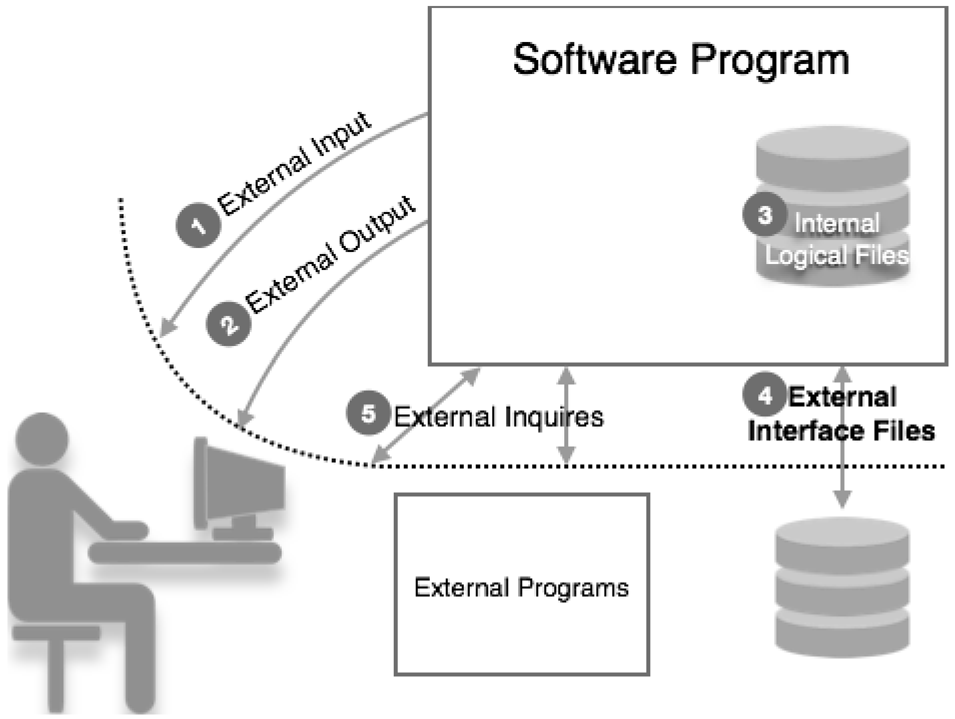
n =8
CyclomaticComplexity=10-8+2
=4

According to P. Jorgensen, Cyclomatic Complexity of a module should not exceed 10.

3.3 Function Point

It is widely used to measure the size of software. Function Point concentrates on functionality provided by the system. Features and functionality of the system are used to measure the software complexity.

Function point counts on five parameters, named as External Input, External Output, Logical Internal Files, External Interface Files, and External Inquiry. To consider the complexity of software each parameter is further categorized as simple, average or complex.



Let us see parameters of function point:

3.3.1 Parameters of function point

External Input

Every unique input to the system, from outside, is considered as external input. Uniqueness of input is measured, as no two inputs should have same formats. These inputs can either be data or control parameters.

- **Simple -** if input count is low and affects less internal files
- **Complex -** if input count is high and affects more internal files
- **Average -** in-between simple and complex.

External Output

All output types provided by the system are counted in this category. Output is considered unique if their output format and/or processing are unique.

- **Simple -** if output count is low
- **Complex -** if output count is high
- **Average -** in between simple and complex.

Logical Internal Files

Every software system maintains internal files in order to maintain its functional information and to function properly. These files hold logical data of the system. This logical data may contain both functional data and control data.

- **Simple -** if number of record types are low
- **Complex -** if number of record types are high
- **Average -** in between simple and complex.

External Interface Files

Software system may need to share its files with some external software or it may need to pass the file for processing or as parameter to some function. All these files are counted as external interface files.

- **Simple -** if number of record types in shared file are low
- **Complex -** if number of record types in shared file are high
- **Average -** in between simple and complex.

External Inquiry

An inquiry is a combination of input and output, where user sends some data to inquire about as input and the system responds to the user with the output of inquiry processed. The complexity of a query is more than External Input and External Output. Query is said to be unique if its input and output are unique in terms of format and data.

- **Simple -** if query needs low processing and yields small amount of output data
- **Complex -** if query needs high process and yields large amount of output data
- **Average -** in between simple and complex.

Each of these parameters in the system is given weight according to their class and complexity. The table below mentions the weight given to each parameter:

Parameter	Simple	Average	Complex
Inputs	3	4	6
Outputs	4	5	7
Enquiry	3	4	6
Files	7	10	15
Interfaces	5	7	10

The table above yields raw Function Points. These function points are adjusted according to the environment complexity. System is described using fourteen different characteristics.

3.3.2 Characteristics for system Description

- Data communications
- Distributed processing
- Performance objectives
- Operation configuration load
- Transaction rate
- Online data entry,
- End user efficiency
- Online update
- Complex processing logic
- Re-usability
- Installation ease
- Operational ease
- Multiple sites
- Desire to facilitate changes

These characteristics factors are then rated from 0 to 5, as mentioned below:

- No influence
- Incidental
- Moderate
- Average
- Significant
- Essential

All ratings are then summed up as N. The value of N ranges from 0 to 70 (14 types of characteristics x 5 types of ratings). It is used to calculate Complexity Adjustment Factors (CAF), using the following formulae:

CAF =0.65+0.01N

Then,

DeliveredFunctionPoints(FP)= CAF x Raw FP

This FP can then be used in various metrics, such as:

Cost = $ / FP

Quality = Errors / FP

Productivity = FP / person-month

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Explain the following terms:

(i) Halstead's Complexity Measures

(ii) Cyclomatic Complexity Measures

(iii) Function Point

2. Describe the formula for each of the 1(i), (ii), and (iii) above.

3. Mention any 10 characteristics for system description

4. Explain any three parameters of function point

5.0 Conclusion

Any enterprise software of value has some level of complexity. As components or modules are developed and incorporated into the system, the complexity increases. Software design complexity is difficult to assess without using complexity metrics and measures. These metrics and measures are discussed in this unit.

6.0 Summary

In this unit we discussed the following:

- *Halstead's Complexity Measures*
- *Cyclomatic Complexity Measures*
- *Function Point*
- Characteristics for system Description

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 2: Software Implementation

1.0 Introduction

The implementation phase plays a very important role in the software development process. It is at this stage that the physical source code of the system being built is created. Programmer’s code the IT system on the basis of the collected requirements and the developed project documentation. They are based on experience and proven software development techniques. Implementation is the process of realizing the design as a program.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

1. Discuss the 3 main concepts used in structured programming
2. Discuss the concepts used in functional programming
3. State any five-coding guideline

3.0 Main Content

In this unit, we will study about programming methods, documentation and challenges in software implementation.

3.1 Structured Programming

In the process of coding, the lines of code keep multiplying, thus, size of the software increases. Gradually, it becomes next to impossible to remember the flow of program. If one forgets how software and its underlying programs, files, procedures are constructed it then becomes very difficult to share, debug and modify the program. The solution to this is structured programming. It encourages the developer to use subroutines and loops instead of using simple jumps in the code, thereby bringing clarity in the code and improving its efficiency Structured programming also helps programmer to reduce coding time and organize code properly.

Structured programming states how the program shall be coded. Structured programming uses three main concepts:

- **Top-down analysis -** A software is always made to perform some rational work. This rational work is known as problem in the software parlance.

Thus, it is very important that we understand how to solve the problem. Under top-down analysis, the problem is broken down into small pieces where each one has some significance. Each problem is individually solved and steps are clearly stated about how to solve the problem.

- **Modular Programming -** While programming, the code is broken down into smaller group of instructions. These groups are known as modules, subprograms or subroutines. Modular programming based on the understanding of top-down analysis. It discourages jumps using 'goto' statements in the program, which often makes the program flow non-traceable. Jumps are prohibited and modular format is encouraged in structured programming.
- **Structured Coding -** In reference with top-down analysis, structured coding sub-divides the modules into further smaller units of code in the order of their execution. Structured programming uses control structure, which controls the flow of the program, whereas structured coding uses control structure to organize its instructions in definable patterns.

3.2 Functional Programming

Functional programming is style of programming language, which uses the concepts of mathematical functions. A function in mathematics should always produce the same result on receiving the same argument. In procedural languages, the flow of the program runs through procedures, i.e. the control of program is transferred to the called procedure. While control flow is transferring from one procedure to another, the program changes its state.

In procedural programming, it is possible for a procedure to produce different results when it is called with the same argument, as the program itself can be in different state while calling it. This is a property as well as a drawback of procedural programming, in which the sequence or timing of the procedure execution becomes important.

Functional programming provides means of computation as mathematical functions, which produces results irrespective of program state. This makes it possible to predict the behaviour of the program.

Functional programming uses the following concepts:

- **First class and High-order functions -** These functions have capability to accept another function as argument or they return other functions as results.

- **Pure functions -** These functions do not include destructive updates, that is, they do not affect any I/O or memory and if they are not in use, they can easily be removed without hampering the rest of the program.
- **Recursion -** Recursion is a programming technique where a function calls itself and repeats the program code in it unless some pre-defined condition matches. Recursion is the way of creating loops in functional programming.
- **Strict evaluation -** It is a method of evaluating the expression passed to a function as an argument. Functional programming has two types of evaluation methods, strict (eager) or non-strict (lazy). Strict evaluation always evaluates the expression before invoking the function. Non-strict evaluation does not evaluate the expression unless it is needed.
- **λ-calculus -** Most functional programming languages use λ-calculus as their type systems. λ-expressions are executed by evaluating them as they occur.

Common Lisp, Scala, Haskell, Erlang and F# are some examples of functional programming languages.

3.3 Programming style

Programming style is set of coding rules followed by all the programmers to write the code. When multiple programmers work on the same software project, they frequently need to work with the program code written by some other developer. This becomes tedious or at times impossible, if all developers do not follow some standard programming style to code the program.

An appropriate programming style includes using function and variable names relevant to the intended task, using well-placed indentation, commenting code for the convenience of reader and overall presentation of code. This makes the program code readable and understandable by all, which in turn makes debugging and error solving easier. Also, proper coding style helps ease the documentation and updating.

3.4 Coding Guidelines

Practice of coding style varies with organizations, operating systems and language of coding itself.

The following coding elements may be defined under coding guidelines of an organization:

- **Naming conventions -** This section defines how to name functions, variables, constants and global variables.

- **Indenting -** This is the space left at the beginning of line, usually 2-8 whitespace or single tab.
- **Whitespace -** It is generally omitted at the end of line.
- **Operators -** Defines the rules of writing mathematical, assignment and logical operators. For example, assignment operator '=' should have space before and after it, as in "x = 2".
- **Control Structures -** The rules of writing if-then-else, case-switch, while-until and for control flow statements solely and in nested fashion.
- **Line length and wrapping -** Defines how many characters should be there in one line, mostly a line is 80 characters long. Wrapping defines how a line should be wrapped, if is too long.
- **Functions -** This defines how functions should be declared and invoked, with and without parameters.
- **Variables -** This mentions how variables of different data types are declared and defined.
- **Comments -** This is one of the important coding components, as the comments included in the code describe what the code actually does and all other associated descriptions. This section also helps creating help documentations for other developers.

3.5 Software Implementation Challenges

There are some challenges faced by the development team while implementing the software. Some of them are mentioned below:

- **Code-reuse -** Programming interfaces of present-day languages are very sophisticated and are equipped huge library functions. Still, to bring the cost down of end product, the organization management prefers to re-use the code, which was created earlier for some other software. There are huge issues faced by programmers for compatibility checks and deciding how much code to re-use.
- **Version Management -** Every time a new software is issued to the customer, developers have to maintain version and configuration related documentation. This documentation needs to be highly accurate and available on time.
- **Target-Host -** The software program, which is being developed in the organization, needs to be designed for host machines at the customers end.

But at times, it is impossible to design a software that works on the target machines.

3.6 Software Documentation

Software documentation is an important part of software process. A well written document provides a great tool and means of information repository necessary to know about software process. Software documentation also provides information about how to use the product.

A well-maintained documentation should involve the following documents:

- **Requirement documentation -** This documentation works as key tool for software designer, developer and the test team to carry out their respective tasks. This document contains all the functional, non-functional and behavioural description of the intended software.

  Source of this document can be previously stored data about the software, already running software at the client's end, client's interview, questionnaires and research. Generally, it is stored in the form of spreadsheet or word processing document with the high-end software management team.

  This documentation works as foundation for the software to be developed and is majorly used in verification and validation phases. Most test-cases are built directly from requirement documentation.
- **Software Design documentation -** These documentations contain all the necessary information, which are needed to build the software. It contains: **(a)** High-level software architecture, **(b)** Software design details, **(c)** Data flow diagrams, **(d)** Database design

  These documents work as repository for developers to implement the software. Though these documents do not give any details on how to code the program, they give all necessary information that is required for coding and implementation.
- **Technical documentation -** These documentations are maintained by the developers and actual coders. These documents, as a whole, represent information about the code. While writing the code, the programmers also mention objective of the code, who wrote it, where will it be required, what it does and how it does, what other resources the code uses, etc.

The technical documentation increases the understanding between various programmers working on the same code. It enhances re-use capability of the code. It makes debugging easy and traceable.

There are various automated tools available and some comes with the programming language itself. For example, java comes JavaDoc tool to generate technical documentation of code.

- **User documentation -** This documentation is different from all the above explained. All previous documentations are maintained to provide information about the software and its development process. But user documentation explains how the software product should work and how it should be used to get the desired results.

  These documentations may include, software installation procedures, how-to guide, user-guides, un-installation method and special references to get more information like license updating etc.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Discuss the three main concepts used in structured programming
2. Discuss the basic concepts used in functional programming
3. State any five coding guidelines
4. Describe the importance of software documentation
5. Explain the challenges associated with software implementation

5.0 Conclusion

Implementation phase is a very important phase of the SDLC. This is when and where the actual coding is carried out. That is, after the requirements have been elicited and specified, analysis and design has been done. We have examined some waynthrough which this can be accomplished.

6.0 Summary

In this unit we discussed the following:

- *Structured Programming*
- *Functional Programming*
- *Programming style*
- Coding Guidelines
- *Software Implementation Challenges*
- Software Documentation

7.0 Further

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 3: Software Testing

1.0 Introduction

Software testing is a critical element of software development life cycles which is called software quality control or software quality assurance. It basic goals are for validation and verification. Validation helps us to know whether we are building the right product. Verification helps us to know whether our product meet its specification. The product could be code, a model, a design diagram, a requirement etc. At each stage, we need to verify that the thing we produce accurately represents its specification

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Explain software testing
- Differentiate between validation and verification
- Identify the importance of software testing
- Differentiate between manual and automated testing
- Identify the basis of software testing
- Differentiate between Black-box testing **and** White-box testing
- Mention the various level of testing

3.0 Main Content

Software Testing is evaluation of the software against requirements gathered from users and system specifications. Testing is conducted at the phase level in software development life cycle or at module level in program code. Software testing comprises of Validation and Verification.

3.1 Software Validation

Validation is process of examining whether or not the software satisfies the user requirements. It is carried out at the end of the SDLC. If the software matches requirements for which it was made, it is validated.

- Validation ensures the product under development is as per the user requirements.
- Validation answers the question – "Are we developing the product which attempts all that user needs from this software?".
- Validation emphasizes on user requirements.

3.2 Software Verification

Verification is the process of confirming if the software is meeting the business requirements, and is developed adhering to the proper specifications and methodologies.

- Verification ensures the product being developed is according to design specifications.
- Verification answers the question– "Are we developing this product by firmly following all design specifications?"
- Verifications concentrates on the design and system specifications.

Target of the test are -

- **Errors -** These are actual coding mistakes made by developers. In addition, there is a difference in output of software and desired output, is considered as an error.
- **Fault -** When error exists fault occurs. A fault, also known as a bug, is a result of an error which can cause system to fail.
- **Failure -** failure is said to be the inability of the system to perform the desired task. Failure occurs when fault exists in the system.

3.3 Manual Vs Automated Testing

Testing can either be done manually or using an automated testing tool:

- **Manual -** This testing is performed without taking help of automated testing tools. The software tester prepares test cases for different sections and levels of the code, executes the tests and reports the result to the manager.
  Manual testing is time and resource consuming. The tester needs to confirm whether or not right test cases are used. Major portion of testing involves manual testing.
- **Automated** This testing is a testing procedure done with aid of automated testing tools. The limitations with manual testing can be overcome using automated test tools.

A test needs to check if a webpage can be opened in Internet Explorer. This can be easily done with manual testing. But to check if the web-server can take the load of 1 million users, it is quite impossible to test manually.

There are software and hardware tools which helps tester in conducting load testing, stress testing, regression testing.

3.4 Testing Approaches

Tests can be conducted based on two approaches –

- Functionality testing
- Implementation testing

When functionality is being tested without taking the actual implementation in concern it is known as black-box testing. The other side is known as white-box testing where not only functionality is tested but the way it is implemented is also analyzed.

Exhaustive tests are the best-desired method for a perfect testing. Every single possible value in the range of the input and output values is tested. It is not possible to test each and every value in real world scenario if the range of values is large.

3.4.1 Black-box testing

It is carried out to test functionality of the program. It is also called 'Behavioural' testing. The tester in this case, has a set of input values and respective desired results. On providing input, if the output matches with the desired results, the program is tested 'ok', and problematic otherwise.

input
output

In this testing method, the design and structure of the code are not known to the tester, and testing engineers and end users conduct this test on the software.

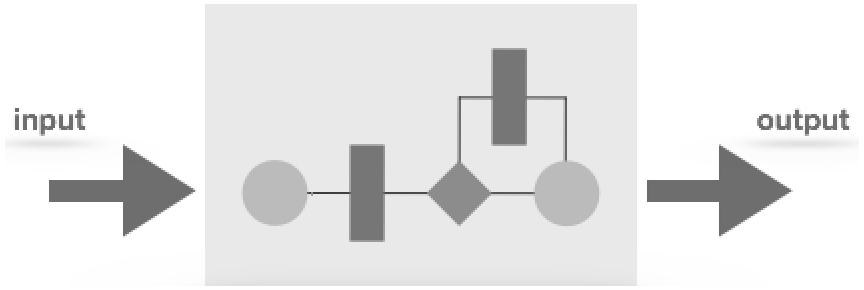
3.4.2 Black-box testing techniques:

- **Equivalence class -** The input is divided into similar classes. If one element of a class passes the test, it is assumed that all the class is passed.
- **Boundary values -** The input is divided into higher and lower end values. If these values pass the test, it is assumed that all values in between may pass too.

- **Cause-effect graphing -** In both previous methods, only one input value at a time is tested. Cause (input) – Effect (output) is a testing technique where combinations of input values are tested in a systematic way.
- **Pair-wise Testing -** The behaviour of software depends on multiple parameters. In pairwise testing, the multiple parameters are tested pair-wise for their different values.
- **State-based testing -** The system changes state on provision of input. These systems are tested based on their states and input.

3.4.2 White-box testing

It is conducted to test program and its implementation, in order to improve code efficiency or structure. It is also known as 'Structural' testing.



In this testing method, the design and structure of the code are known to the tester. Programmers of the code conduct this test on the code.

The below are some White-box testing techniques:

- **Control-flow testing -** The purpose of the control-flow testing to set up test cases which covers all statements and branch conditions. The branch conditions are tested for both being true and false, so that all statements can be covered.
- **Data-flow testing -** This testing technique emphasis to cover all the data variables included in the program. It tests where the variables were declared and defined and where they were used or changed.

3.5 Testing Levels

Testing itself may be defined at various levels of SDLC. The testing process runs parallel to software development. Before jumping on the next stage, a stage is tested, validated and verified.

Testing separately is done just to make sure that there are no hidden bugs or issues left in the software. Software is tested on various levels -

3.5.1 Unit Testing

While coding, the programmer performs some tests on that unit of program to know if it is error free. Testing is performed under white-box testing approach. Unit testing helps developers decide that individual units of the program are working as per requirement and are error free.

3.5.2 Integration Testing

Even if the units of software are working fine individually, there is a need to find out if the units if integrated together would also work without errors. For example, argument passing and data updating, etc.

3.5.3 System Testing

The software is compiled as product and then it is tested as a whole. This can be accomplished using one or more of the following tests:

- **Functionality testing -** Tests all functionalities of the software against the requirement.
- **Performance testing -** This test proves how efficient the software is. It tests the effectiveness and average time taken by the software to do desired task. Performance testing is done by means of load testing and stress testing where the software is put under high user and data load under various environment conditions.
- **Security & Portability -** These tests are done when the software is meant to work on various platforms and accessed by number of persons.

3.6 Acceptance Testing

When the software is ready to hand over to the customer it has to go through last phase of testing where it is tested for user-interaction and response. This is important because even if the software matches all user requirements and if user does not like the way it appears or works, it may be rejected.

- **Alpha testing -** The team of developer themselves perform alpha testing by using the system as if it is being used in work environment. They try to find out how user would react to some action in software and how the system should respond to inputs.

- **Beta testing -** After the software is tested internally, it is handed over to the users to use it under their production environment only for testing purpose. This is not as yet the delivered product. Developers expect that users at this stage will bring minute problems, which were skipped to attend.

3.7 Regression Testing

Whenever a software product is updated with new code, feature or functionality, it is tested thoroughly to detect if there is any negative impact of the added code. This is known as regression testing.

3.8 Testing Documentation

Testing documents are prepared at different stages **-**

Before Testing

Testing starts with test cases generation. Following documents are needed for reference –

- **SRS document -** Functional Requirements document
- **Test Policy document -** This describes how far testing should take place before releasing the product.
- **Test Strategy document -** This mentions detail aspects of test team, responsibility matrix and rights/responsibility of test manager and test engineer.
- **Traceability Matrix document -** This is SDLC document, which is related to requirement gathering process. As new requirements come, they are added to this matrix. These matrices help testers know the source of requirement. They can be traced forward and backward.

While Being Tested

The following documents may be required while testing is started and is being done:

- **Test Case document -** This document contains list of tests required to be conducted. It includes Unit test plan, Integration test plan, System test plan and Acceptance test plan.
- **Test description -** This document is a detailed description of all test cases and procedures to execute them.
- **Test case report -** This document contains test case report as a result of the test.

- **Test logs -** This document contains test logs for every test case report.

After Testing

The following documents may be generated after testing:

- **Test summary -** This test summary is collective analysis of all test reports and logs. It summarizes and concludes if the software is ready to be launched. The software is released under version control system if it is ready to launch.

3.9 Testing vs. Quality Control, Quality Assurance and Audit

We need to understand that software testing is different from software quality assurance, software quality control and software auditing.

- **Software quality assurance -** These are software development process monitoring means, by which it is assured that all the measures are taken as per the standards of organization. This monitoring is done to make sure that proper software development methods were followed.
- **Software quality control -** This is a system to maintain the quality of software product. It may include functional and non-functional aspects of software product, which enhance the goodwill of the organization. This system makes sure that the customer is receiving quality product for their requirement and the product certified as 'fit for use'.
- **Software audit -** This is a review of procedure used by the organization to develop the software. A team of auditors, independent of development team examines the software process, procedure, requirements and other aspects of SDLC. The purpose of software audit is to check that software and its development process, both conform standards, rules and regulations.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Explain software testing
2. Compare validation and verification
3. State the importance of software testing
4. Differentiate between manual and automated testing
5. Identify the basis of software testing
6. Differentiate between Black-box testing andWhite-box testing
7. Mention the various level of testing in software development

8. Explain the following terms: software quality assurance, software quality control and system audit

5.0 Conclusion

Software testing is basically carried out fish out errors and bugs in the software before it is delivered or deployed. It is aimed producing functional, reliable and maintainable software. It helps in the production of quality and cost-effective software. Software testing is concerned with the validation and verification of software.

6.0 Summary

In this unit we discussed the following:

- *Software Validation*
- *Software Verification*
- *Manual Vs Automated Testing*
- *Testing Approaches*
- *Testing Levels*
- *Acceptance Testing*
- Regression Testing
- *Testing Documentation*
- *Testing vs. Quality Control, Quality Assurance and Audit*

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 4: Software Maintenance

1.0 Introduction

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updates done after the delivery of software product.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Mention and discuss different types of software maintenance
- Outline real world factors affecting software maintenance cost
- List software-end factors affecting maintenance cost
- Mention at least 5 factors

3.0 Main Content

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updates done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions -** Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements -** Over the time, customer may ask for new features or functions in the software.

- **Host Modifications -** If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes -** If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

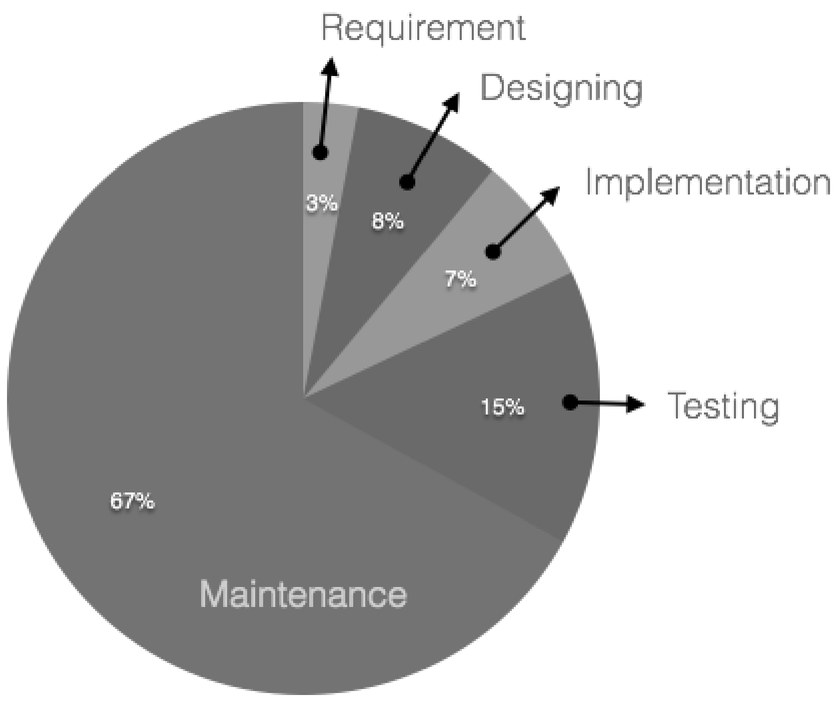
3.1 Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- **Corrective Maintenance -** This includes modifications and updates done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance -** This includes modifications and updates applied to keep the software product up-to date and tuned to the ever-changing world of technology and business environment.
- **Perfective Maintenance -** This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance -** This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

3.2 Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.



On an average, the cost of software maintenance is more than 50% of all SDLC phases.

3.3 Factors Influencing Software Maintenance Cost

There are various factors, which trigger maintenance cost go high, such as:

3.3.1 Real-world factors affecting Maintenance Cost

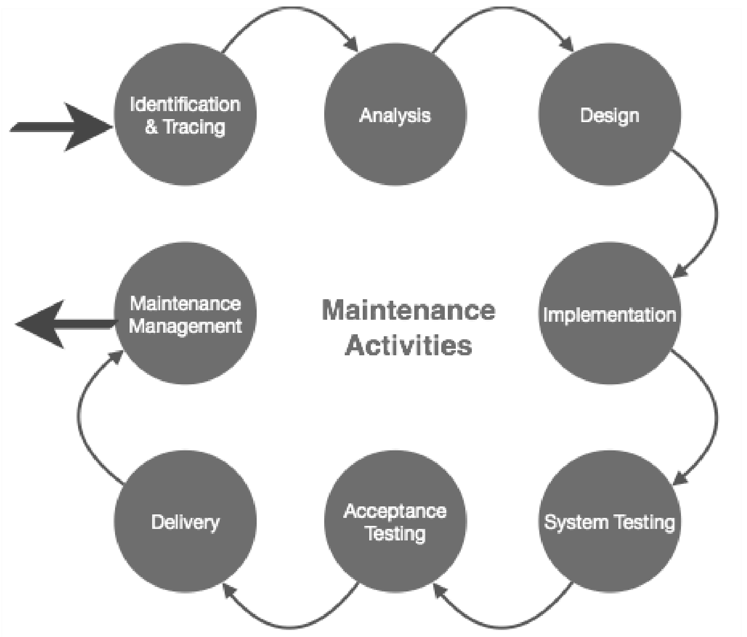
- The standard age of any software is considered up to 10 to 15 years.
- Older software, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced software on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

3.4 Software-end factors affecting Maintenance Cost

- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability

3.5 Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be included.



These activities go hand-in-hand with each of the following phase:

- **Identification & Tracing -** It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
- **Analysis -** The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.

- **Design -** New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.
- **Implementation -** The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.
- **System Testing -** Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.
- **Acceptance Testing -** After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.
- **Delivery -** After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

  Training facility is provided if required, in addition to the hard copy of user manual.
- **Maintenance management -** Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

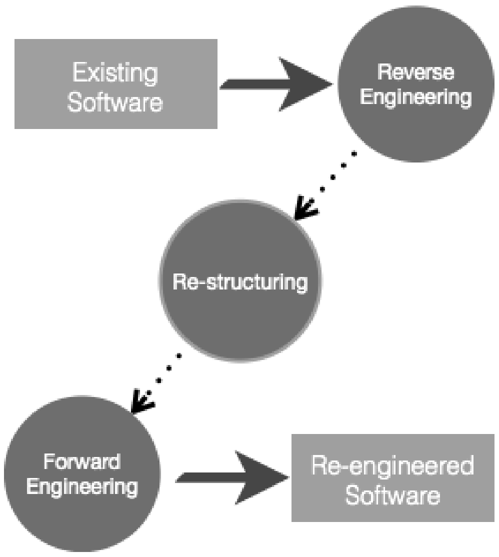
3.6 Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.



3.6.1 Re-Engineering Process

- Decide what to re-engineer. Is it whole software or a part of it?
- Perform Reverse Engineering, in order to obtain specifications of existing software.
- Restructure Program if required. For example, changing function-oriented programs into object-oriented programs.
- Re-structure data as required.
- Apply Forward engineering concepts in order to get re-engineered software.

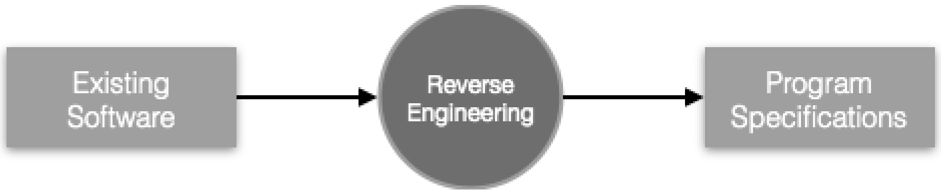
3.6.2 Terminologies inSoftware Re-Engineering

There are few important terms used in Software re-engineering

3.7 Reverse Engineering

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.



3.8 Program Restructuring

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

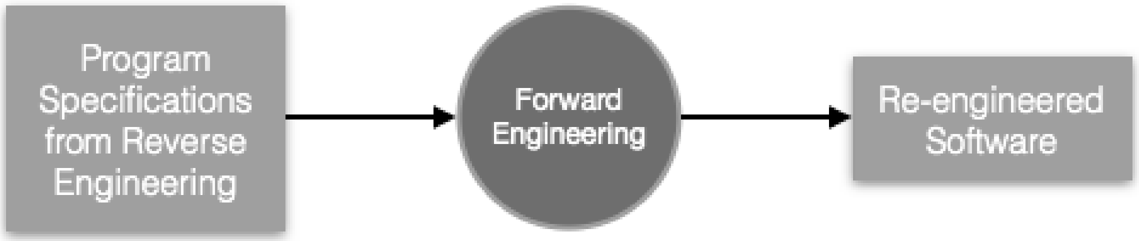
Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

3.9 Forward Engineering

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.

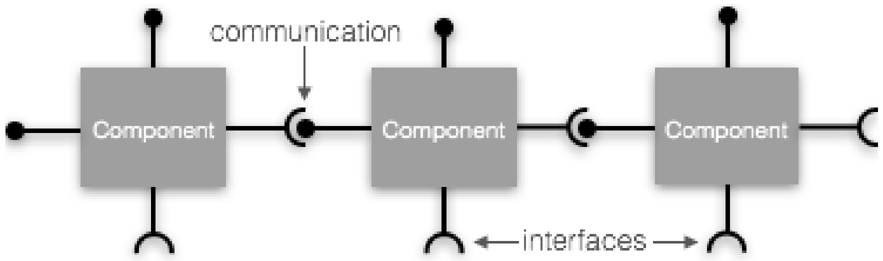


3.10 Component reusability

A component is a part of software program code, which executes an independent task in the system. It can be a small module or sub-system itself.

Example

The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.
Components have high cohesion of functionality and lower rate of coupling, i.e. they work independently and can perform tasks without depending on other modules.
In OOP, the objects are designed are very specific to their concern and have fewer chances to be used in some other software.
In modular programming, the modules are coded to perform specific tasks which can be used across number of other software programs.
There is a whole new vertical, which is based on re-use of software component, and is known as Component Based Software Engineering (CBSE).



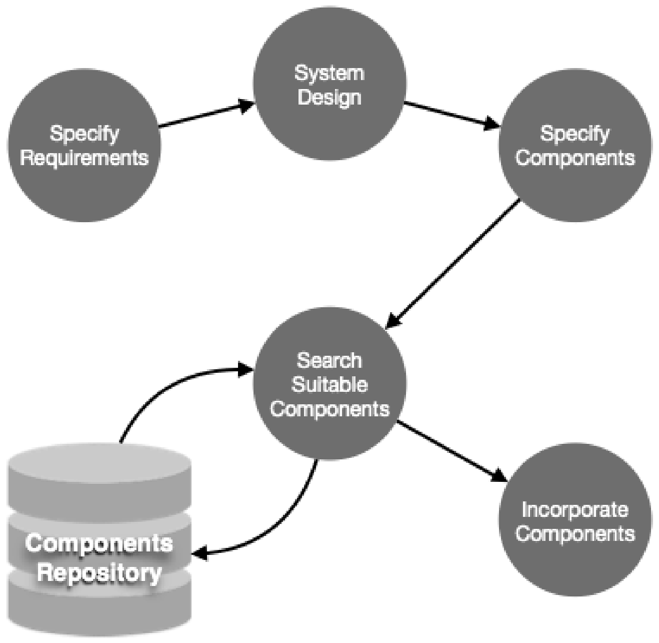
Re-use can be done at various levels

- **Application level -** Where an entire application is used as sub-system of new software.
- **Component level -** Where sub-system of an application is used.
- **Modules level -** Where functional modules are re-used.

Software components provide interfaces, which can be used to establish communication among different components.

3.11 Reuse Process

Two kinds of method can be adopted: either by keeping requirements same and adjusting components or by keeping components same and modifying requirements.



- **Requirement Specification -** The functional and non-functional requirements are specified, which a software product must comply to, with the help of existing system, user input or both.
- **Design -** This is also a standard SDLC process step, where requirements are defined in terms of software parlance. Basic architecture of system as a whole and its sub-systems are created.
- **Specify Components -** By studying the software design, the designers segregate the entire system into smaller components or sub-systems. One complete software design turns into a collection of a huge set of components working together.
- **Search Suitable Components -** The software component repository is referred by designers to search for the matching component, on the basis of functionality and intended software requirements.
- **Incorporate Components -** All matched components are packed together to shape them as complete software.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Mention and discuss different types of software maintenance
2. State the factors affecting software maintenance cost
3. Explain the software-end factors affecting maintenance cost
4. Explain software reuse at three levels of system development

5. Describe the following terms: Forward engineering, reverse engineering and program restructuring.

5.0 Conclusion

Software maintenance allows for continuous modification of deployed software in order to meet changing requirements. This elongates the life span of the software. There are basically four types of maintenance: Corrective Maintenance, Adaptive Maintenance, Perfective Maintenance and Preventive Maintenance

6.0 Summary

In this unit we discussed the following:

- *Types of maintenance*
- *Cost of Maintenance*
- Factors Influencing Software Maintenance Cost
- *Maintenance Activities*
- *Software Re-engineering*
- Reverse Engineering
- Program Restructuring
- Forward Engineering
- *Component reusability*
- Reuse Process

7.0 Further

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: "Software Engineering"- A Practitioner's Approach", McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

Unit 5: Software CASE Tools

1.0 Introduction

CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- Briefly describe CASE tool
- Mention and discuss different types of CASE tools
- Outline the concern of configuration management

3.0 Main Content

CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.

3.1 CASE Tools

CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.

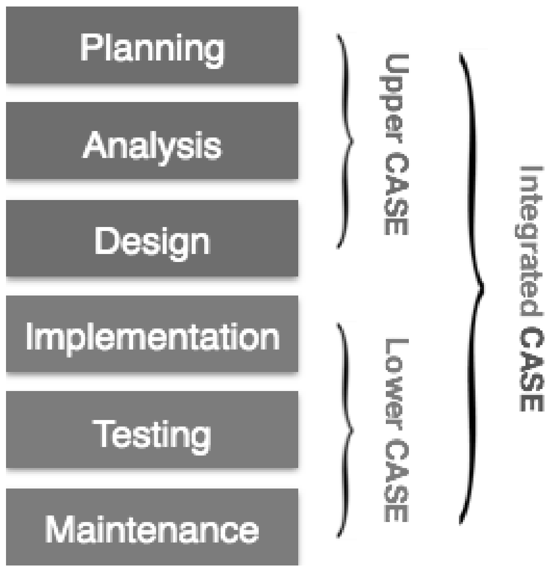
There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few.

Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development.

3.1.1 Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:

- **Central Repository** - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management is stored. Central repository also serves as data dictionary.



- **Upper Case Tools -** Upper CASE tools are used in planning, analysis and design stages of SDLC.
- **Lower Case Tools -** Lower CASE tools are used in implementation, testing and maintenance.
- **Integrated Case Tools -** Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.

3.2 Scope of Case Tools

The scope of CASE tools goes throughout the SDLC.

3.2.1 Case Tools Types

Now we briefly go through various CASE tools

Diagram tools: These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form. For example, Flow Chart Maker tool for creating state-of-the-art flowcharts.

Process Modeling Tools: Process modeling is method to create software process model, which is used to develop the software.

Process modeling tools help the managers to choose a process model or modify it as per the requirement of software product. For example, EPF Composer

Project Management Tools: These tools are used for project planning, cost and effort estimation, project scheduling and resource planning. Managers have to strictly comply project execution with every mentioned step in software project management. Project management tools help in storing and sharing project information in real-time throughout the organization. For example, Creative Pro Office, Trac Project, Basecamp.

Documentation Tools: Documentation in a software project starts prior to the software process, goes throughout all phases of SDLC and after the completion of the project.

Documentation tools generate documents for technical users and end users. Technical users are mostly in-house professionals of the development team who refer to system manual, reference manual, training manual, installation manuals etc. The end user documents describe the functioning and how-to of the system such as user manual. For example, Doxygen, DrExplain, Adobe RoboHelp for documentation.

Analysis Tools: These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa, CaseComplete for requirement analysis, Visible Analyst for total analysis.

Design Tools: These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

Configuration Management Tools: An instance of software is released under one version. Configuration Management tools deal with:

- Version and revision management
- Baseline configuration management
- Change control management

CASE tools help in this by automatic tracking, version management and release management. For example, Fossil, Git, Accu REV.

Change Control Tools: These tools are considered as a part of configuration management tools. They deal with changes made to the software after its baseline is fixed or when the software is first released. CASE tools automate change tracking, file management, code management and more. It also helps in enforcing change policy of the organization.

Programming Tools: These tools consist of programming environments like IDE (Integrated Development Environment), in-built modules library and simulation tools. These tools provide comprehensive aid in building software product and include features for simulation and testing. For example, Cscope to search code in C, Eclipse.

Prototyping Tools: Software prototype is simulated version of the intended software product. Prototype provides initial look and feel of the product and simulates few aspect of actual product.

Prototyping CASE tools essentially come with graphical libraries. They can create hardware independent user interfaces and design. These tools help us to build rapid prototypes based on existing information. In addition, they provide simulation of software prototype. For example, Serena prototype composer, Mockup Builder.

Web Development Tools: These tools assist in designing web pages with all allied elements like forms, text, script, graphic and so on. Web tools also provide live preview of what is being developed and how will it look after completion. For example, Fontello, Adobe Edge Inspect, Foundation 3, Brackets.

Quality Assurance Tools: Quality assurance in a software organization is monitoring the engineering process and methods adopted to develop the software product in order to ensure conformance of quality as per organization standards. QA tools consist of configuration and change control tools and software testing tools. For example, SoapTest, AppsWatch, JMeter.

Maintenance Tools: Software maintenance includes modifications in the software product after it is delivered. Automatic logging and error reporting techniques, automatic error ticket generation and root cause Analysis are few CASE tools, which help software organization in maintenance phase of SDLC. For example, Bugzilla for defect tracking, HP Quality Center.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. State the importance of CASE tool
2. Mention and discuss different types of CASE tools
3. Explain the concern of configuration management
4. Outline the task of configuration management tools
5. Provide examples of each of the following software development tools: (i) Maintenance (ii) Quality Assurance (iii) Prototyping (iv) Quality assurance

5.0 Conclusion

CASE is an acronym for **C**omputer **A**ided **S**oftware **E**ngineering. It the application of automated software tools in the crafting software product. They come in different flavour depending on the task at hand: Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools

6.0 Summary

In this unit we discussed the following:

- *CASE Tools*
- *Components of CASE Tools*
- *Scope of Case Tools*
- *Case Tools Types*

7.0 Further Reading

Barry Boehm (1996)., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM)

Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.

Charles S. Wasson (2006) System Analysis, Design, and Development Concepts, Principles, and Practices, Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Hans-Petter Halvorsen (2020) Software Development A Practical Approach! https://halvorsen.blog

Pressman Roger S: “Software Engineering”- A Practitioner’s Approach”, McGraw Hill, 5th edition. 2000.

Richard H. Thayer, Barry W. Boehm (1986). Tutorial: software engineering project management. Computer Society Press of the IEEE. p.130

Rushby John: Formal Methods and the Certification of Critical Systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

Woodcock Jim, Larsen Peter Gorm, Bicarregui Juan and Fitzgerald John: Formal Methods: Practice and Experience, ACM Computing Surveys (CSUR), Volume 41 Issue 4, 2009 Article No. 19.

CIT432 SOFTWARE ENGINEERING II