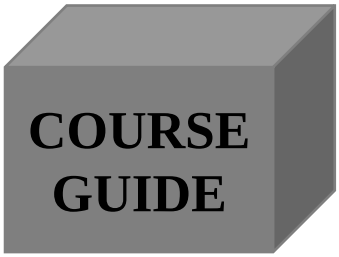
NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT308

COURSE TITLE: FORMAL METHODS AND SOFTWARE DEVELOPMENT



COURSE CODE: CIT308

COURSE TITLE: FORMAL METHODS AND SOFTWARE DEVELOPMENT

COURSE TEAM: Dr. Moses Ekpenyong and Dr. Francis B. Osang (Developers)

University of Uyo and National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria Headquarters
University Village, Plot 91 Cadastral Zone, Nnnamdi 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

PUBLISHED BY:
National Open University of Nigeria

First Printed 2022

ISBN: 978-058-851-5

Course Guide

Introduction

CIT 308: Formal Methods and Software Development is a 3-credit unit course for students studying towards acquiring a Bachelor of Science in Computer Science and other related disciplines.

The course is divided into 6 modules and 23 study units. It will first take a brief review of the concepts of Formal Methods and Software Development. This course will then go ahead to deal with the detailed discussion on formal methods as well as Software development. This course also introduces such other knowledge that will enable the reader have proper understanding of how Formal methods can be used in Software Development in order to produce high quality software product.

The course guide therefore gives you an overview of what the course; CIT 308 is all about, the textbooks and other materials to be referenced, what you expect to know in each unit, and how to work through the course material.

Course Competencies

The overall aim of this course, CIT 308 is to introduce you to basic concepts of Formal Methods and Software Development in order to enable you to understand the basic elements of Formal Methods as well as Software Development use in design of software especially, in security and life critical applications such as aviation etc. In this course of your studies, you will be put through the definitions of common terms in relation to Formal Methods and Software Development, stage in software development life cycle, requirement specification, development tools etc.

Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objectives of this course. On completing this course, you should be able to:

- State why software quality is important

- Outline some of the characteristics of a high-quality software
- Enumerate a typical software development phase
- Define formal methods
- Mention the of formal method used in Software Development
- Mention and the uses of formal methods in Software Development
- Explain and where to use Formal Methods
- Give a description on the need to used formal methods
- 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
- Define proposition
- Identify proposition operators
- Construct and interpret propositions
- Construct truth tables
- Give a background of formal methods
- Discuss formal proof
- Mention some terminologies used in mathematical proof
- Briefly explain the four proofing methods
- 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
- Discuss the various stages to apply formal methods
- Discuss what to do at various stages
- Discuss terminologies used in Z notation
- Outline the various functions in Z notation
- Relate software development with engineering process
- State some software evolution laws
- Discuss E-Type software evolution
- Discuss the need of Software Engineering
- Outline the characteristics of good software
- List the SDLC activities

- Explain the SDLC activities with aid of a diagram
- List and explain the Software Development Paradigm
- Identify the characteristics of a software project
- Describe a software project
- Justify the need for software project management
- Identify the job of a software project manager
- Explain the following: project planning, scope management and project estimation
- Mention at least 3 project management tools
- List and explain the four steps in requirement engineering process
- Depict the requirement elicitation process with a diagram
- Mention at least 6 requirement elicitation techniques
- List at least 10 software requirement characteristics
- Differentiate between functional and non-functional software requirements
- Mention at 10 user interface requirements
- Outline the responsibility of a system analyst
- Differentiate between software metric and software measures
- Software design yields three levels of results. Mention and briefly describe them
- Discuss modularization and state its advantages
- Differentiate between cohesion and coupling
- List and explain any 5 types of cohesion
- Mention and discuss difference types of software design
- List and explain the different concepts of object-oriented design
- Mention and discuss two generic approaches for software design
- Mention and discuss difference types of software design
- List and explain the different concepts of object-oriented design
- Mention and discuss two generic approaches for software design
- Discuss the 3 main concept used in structured programmig
- Discuss the concepts used in functional programming
- State any 5 coding guideline
- 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
- 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
- Briefly describe CASE tool
- Mention and discuss different types of CASE tools
- Outline the concern of configuration management

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.

This course material contains six modules and twenty-four study units as follows:

MODULE 1: AN OVERVIEW OF FORMAL METHODS AND SOFTWARE DEVELOPMENT

Unit 1: General Information
Unit 2: Approaches to formal methods and their use in software development

MODULE 2: FORMAL METHODS

Unit 1: Introduction to Formal Methods
Unit 2: Proposition
Unit 3: Predicates
Unit 4: Sets
Unit 5: Series or Sequence

MODULE 3: FORMAL METHODS CONTINUES

Unit 1: Mathematical Proof
Unit 2: Testing
Unit 3: Application to Formal Specification
Unit 4: Z Notation

MODULE 4: SOFTWARE DEVELOPMENT OVERVIEW

Unit 1: **S**oftware Development and **S**oftware Engineering
Unit 2: Software Development Life Cycle
Unit 3: Software Project Management
Unit 4: Software Requirements

MODULE 5: OVERVIEW OF SOFTWARE DESIGN, ANALYSIS AND DESIGN TOOLS, DESIGN STRATEGIES AND USER INTERFACE BASICS

Unit 1: Software Design Basics
Unit 2: Analysis and Design tools
Unit 3: Software Design Strategies
Unit 4: Software User Interface Design

MODULE 6: OVERVIEW OF DESIGN COMPLEXITY, **SOFTWARE IMPLEMENTATION, TESTING**, **MAINTENANCE AND CASE TOOLS**

Unit 1: Design Complexity
Unit 2: Software Implementation
Unit 3: Software Testing
Unit 4: Software Maintenance
Unit 5: Software Case Tools

References and Further Reading

Every study unit contain list of references and Further Reading. Do not hesitate to consult them if need be.

Presentation Schedule

The Presentation Schedule included in your course material gives you important dates for the completion of Tutor Marked Assignments and tutorial attendance. Remember, you are required to submit all your assignments by the due date. You should guard against falling behind in your work.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total

course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

How to get the Most from the Course

In distance learning, the study units replace the university lectures. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suits you best. Think of it as reading the lecture instead of listening to the lecturer. In the same way a lecturer might give you some reading to do, the study units tell you when to read, and which are your text materials or set books. You are provided exercises to do at appropriate points, just as a lecturer might give you an in-class exercise.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit, and how a particular unit is integrated with the other units and the course as a whole. Next to this is a set of learning objectives. These objectives let you know what you should be able to do by the time you have completed the unit. These learning objectives are meant to guide your study. The moment a unit is finished, you must go back and check whether you have achieved the objectives. If you make this a habit, then you will significantly improve your chances of passing the course. The main body of the unit guides you through the required reading from other sources. This will usually be either from your set books or from a reading section. The following is a practical strategy for working through this course. If you run into any trouble, telephone your tutor. Remember that your tutor‘s job is to help you. When you need assistance, do not hesitate to call and ask your tutor to provide it.

In addition, do the following:

1. Read this Course Guide thoroughly, it is your first assignment.
2. Organise a Study Schedule. Design a ―Course Overview‖ to guide you through the Course. Note the time you are expected to spend on each unit and how the assignments relate to the units. Important information, e.g. details of your tutorials, and the date of the first day of the semester is available from the study centre. You need to gather all the information into one place, such as your diary or a wall calendar. Decide on a method and write in your own dates and schedule of work for each unit.
3. Once you have created your own study schedule, do everything to stay faithful to it. The major reason students fail is that they get behind with their course work. If you get into difficulty with your schedule, please, let your tutor know before it is tool late for help.
4. Turn to Unit 1, and read the introduction and the objectives for the unit.
5. Assemble the study materials. You will need your set books and the unit you are studying at any point in time.

6. Work through the unit. As you work through it, you will know what sources to consult for further information.
7. Keep in touch with your study centre as up-to-date course information will be continuously available there.
8. Well before the relevant due dates (about 4 weeks before due dates), keep in mind that you will learn a lot by doing the assignments carefully. They have been designed to help you meet the objectives of the course and therefore will help you pass the examination. Submit all assignments not later than the due date.
9. Review the objectives for each study unit to confirm that you have achieved them. If you fee unsure about any of the objectives, review the study materials or consult your tutor.
10. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
11. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and on the ordinary assignments.
12. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in the Course Guide).
13. Finally, ensure that you practice on the personal computer as prescribed to gain the maximum proficiency required.

Facilitation

The dates, times and locations of these Tutorials will be made available to you, together with the name, telephone number and address of your Tutor. Each assignment will be marked by your tutor. Pay close attention to the comments your tutor might make on your assignments as these will help in your progress. Make sure that assignments reach your tutor on or before the due date.

Your tutorials are important; therefore, try not to skip any. It is an opportunity to meet your tutor and your fellow students. It is also an opportunity to get the help of your tutor and discuss any difficulties you might encounter when reading.

Course Information

Course Code: CIT 308

Course Title: FORMAL METHODS AND SOFTWARE DEVELOPMENT

Credit Unit:
Course Status:

Course Blurb: This course covers principal topics in Formal Methods and Software Development. It will first take a brief review of the concepts of Formal Methods and Software Development. This course will then go ahead to deal with the different stages involved developing good and functional Software. The course went further to deal with stages of SDLC. This course also introduces such other knowledge that will enable the reader have proper understanding of developing reliable, functional, maintainable software.

Semester:
Course Duration: Required
Hours for study

Course Team

Course Developer: xxxxxxx
Course Writer: xxxxxxxx
Department of Computer Science
University of xxxxxxx
xxxxxxx.
Content Editor: xxxxxxxx
Department of Computer Science
University of xxxxx
xxxxxx
Instructional Designer:
Learning Technologists:
Copy Editor
Xxxxxxxx

MODULE 1: AN OVERVIEW OF FORMAL METHODS AND SOFTWARE DEVELOPMENT

This module is divided into two (2) units

Unit 1: **General Information**

Unit 2: Approaches to formal methods and their use in software development

Unit 1: **General Information**

Contents

Contents

1.0 Introduction

Software is driving the society. Software is used in almost every area of endeavours, if not all. It therefore becomes imperative, for software developers to produce software that meets the user's need in terms of functionality, reliability, availability etc. To achieve this, the software developer makes use of several tools and techniques which includes formal methods among others. In this unit, we shall be discussing some shared characteristics of a high-quality software, the need for precision in the specification of software, typical software development phases and the role of formal methods.

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- State why software quality is important
- Outline some of the characteristics of a high-quality software
- Enumerate a typical software development phase

3.0 Main Content

3.1 The importance of high-quality software

Software development is a vital activity in modern society, and is likely to have increasing significance in the future. Software manages our bank accounts, pays our salaries, controls the aircraft we fly in, regulates power generation and distribution, controls our communications, etc.

3.2 Characteristics of high-quality software

High quality software shares the following obvious attributes:

- high quality software is intuitive and easy to use -- the right things happen "automatically"
- it is efficient -- people use computers to get things done quickly
- above all, high quality software is correct -- it always produces the advertised results and does not crash!

3.3 The need for precision in the specification of software

The notion that software components can be reused is a principal motivation of object-oriented programming, and has virtually become a postulate of programming. To reuse a previously written software component (or create a new one), a software engineer must have a precise description of its behavior. This precision is essential as even a minor misconception of the function of a component that is unapparent at the outset may cause serious errors that are difficult and expensive to correct later in the process.

3.4 Typical software development phases

Software development models commonly subdivide the process into phases similar to the following :

1. requirements analysis: determine user needs
2. specification: describe precisely what the role of the software will be
3. design: determine how to realize the software, and devise overall organization
4. implementation: formulate the algorithms and program(s)
5. verification: certify that the program(s) meet the specification
6. maintenance: perform ongoing changes and corrections after the software is in use

3.5 The role of formal methods

Formal methods are intended to systematize and introduce rigor into all the phases of software development. This helps us to avoid overlooking critical issues, provides a standard means to record various assumptions and decisions, and forms a basis for consistency among many related activities. By providing precise and unambiguous description mechanisms, formal methods facilitate the understanding required to coalesce the various phases of software development into a successful endeavour.

The programming language used for software development furnishes precise syntax and semantics for the implementation phase, and this has been true since people began writing programs. But precision in all but this one phase of software development must derive from other sources. The term "formal methods" pertains to a broad collection of formalisms and abstractions intended to support a comparable level of precision for other phases of software development. While this includes issues currently under active development, several methodologies have reached a level of maturity that can be of benefit to practitioners.

There is a discernible tendency to merge discrete mathematics and formal methods for software engineering (e.g., see the books by Denvir, Ince, and Woodcock & Loomes). Many such topics do indeed support software engineering and it is neither possible nor desirable to avoid these topics when pursuing formal methods. But we will not take the approach that applying discrete mathematics to software engineering assures germane formal methods. The overriding concern of software engineering is the creation of high quality software systems. With "formal methods" we pursue melding those things that nurture rigor and precision into this endeavor. While our focus on the activities that precede the actual programming itself does lead to machine independent abstractions often associated with mathematics, much of the material has been developed (or tailored) to suit the context of software creation.

Some specific formalisms used in software development include:

- algebraic specification (including OBJ) -- used for specification and verification,
- predicate logic (including Z) -- used for specification and verification,
- statecharts -- used for specification of "reactive" systems
- UML -- used primarily for design, and also for requirements analysis.

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Why is software quality important in organization?
2. Outline some of the characteristics of a high-quality software
3. Enumerate the stages in software development
4. What is the role of formal methods in software development?
5. Outline some specific formalism used in software development

5.0 Conclusion

This unit provides a general information concerning the use of formal methods in software development.

6.0 Summary

Some of the points heighted include the importance of high-quality software, characteristics of high-quality software, the need for precision in the specification of software, typical software development phases and the role of 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) Formal Methods in Software Engineering

Unit 2: Approaches to formal methods and their use in software development

Contents

1.0 Introduction

Formal methods in software engineering or development are mathematical techniques that are used in the design, implementation and testing of computer systems. The application of mathematical methods in the development and verification of software is very labor intensive, and thus expensive. In this unit, we shall be looking at some of the semantics used in formal methods, application areas of FM etc

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- **Define formal methods**
- **Mention the of formal method used in** Software Development
- Mention the uses of formal methods in Software Development
- Explain and where to use Formal Methods
- Give a description on the need to used formal methods

3.0 Main Content

3.1 Definition

Formal methods in software engineering or development are mathematical techniques that are used in the design, implementation and testing of computer systems. The application of mathematical methods in the development and verification of software is very labor intensive, and thus expensive. Therefore, it is not feasible to check all the wanted properties of a complete computer program in detail.

3.2 Type formal methods in Software Development

Formal methods include

- Formal specification
- Specification analysis and proof
- Transformational development
- Program verification

3.3 Classification of Formal Methods Semantics

As with programming language semantics, styles of formal methods may be roughly classified as follows:

- Denotational semantics, in which the meaning of a system is expressed in the mathematical theory of domains. Proponents of such methods rely on the well-understood nature of domains to give meaning to the system; critics point out that not every system may be intuitively or naturally viewed as a function.
- Operational semantics, in which the meaning of a system is expressed as a sequence of actions of a simpler computational model. Proponents of such methods point to the simplicity of their models as a means to express clarity; critics counter that the problem of semantics has just been delayed.
- Axiomatic semantics, in which the meaning of the system is expressed in terms of preconditions and post conditions which are true before and after the system performs a task, respectively. Proponents note the connection to classical logic; critics note that such semantics never really describe what a system does (merely what is true before and afterwards).

3.4 Uses of Formal Methods

Formal methods can be applied at various points through the development process.

Specification

Formal methods may be used to give a description of the system to be developed, at whatever level(s) of detail desired. This formal description can be used to guide further development activities additionally, it can be used to verify that the requirements for the system being developed have been completely and accurately specified. The need for formal specification systems has been noted for years. In the ALGOL 58 report,John Backus presented a formal notation for describing programming language syntax, later named Backus normal form then renamed Backus–Naur form (BNF).

Development: Once a formal specification has been produced, the specification may be used as a guide while the concrete system is developed during the design process (i.e., realized typically in software, but also potentially in hardware).

For example:

•If the formal specification is in an operational semantics, the observed behaviour of the concrete system can be compared with the behaviour of the specification (which itself should be executable). Additionally, the operational commands of the specification may be amenable to direct translation into executable code.

•If the formal specification is in an axiomatic semantics, the preconditions and postconditions of the specification may become assertions in the executable code.

Verification

Once a formal specification has been developed, the specification may be used as the basis for proving properties of the specification (and hopefully by inference the developed system).

Human-directed proof

Sometimes, the motivation for proving the correctness of a system is not the obvious need for reassurance of the correctness of the system, but a desire to understand the system better. Consequently, some proofs of correctness are produced in the style of mathematical proof: handwritten (or typeset) using natural language, using a level of informality common to such proofs. A "good" proof is one which is readable and understandable by other human readers.

Critics of such approaches point out that the ambiguity inherent in natural language allows errors to be undetected in such proofs; often, subtle errors can be present in the low-level details typically overlooked by such proofs. Additionally, the work involved in producing such a good proof requires a high level of mathematical sophistication and expertise.

Automated proof

In contrast, there is increasing interest in producing proofs of correctness of such systems by automated means. Automated techniques fall into three general categories:

•Automated theorem proving, in which a system attempts to produce a formal proof from scratch, given a description of the system, a set of logical axioms, and a set of inference rules.

•Model checking, in which a system verifies certain properties by means of an exhaustive search of all possible states that a system could enter during its execution.

•Abstract interpretation, in which a system verifies an over-approximation of a behavioural property of the program, using a fixpoint computation over a (possibly complete) lattice representing it.

Some automated theorem proofers require guidance as to which properties are "interesting" enough to pursue, while others work without human intervention.

Model checkers can quickly get bogged down in checking millions of uninteresting states if not given a sufficiently abstract model.

Proponents of such systems argue that the results have greater mathematical certainty than human produced proofs, since all the tedious details have been algorithmically verified. The training required to use such systems is also less than that required to produce good mathematical proofs by hand, making the techniques accessible to a wider variety of practitioners.

Critics note that some of those systems are like oracles: they make a pronouncement of truth, yet give no explanation of that truth. There is also the problem of "verifying the verifier"; if the program which aids in the verification is itself unproven, there may be reason to doubt the soundness of the produced results. Some modern model checking tools produce a "proof log" detailing each step in their proof, making it possible to perform, given suitable tools, independent verification.

The main feature of the abstract interpretation approach is that it provides a sound analysis, i.e., no false negatives are returned. Moreover, it is efficiently scalable, by tuning the abstract domain representing the property to be analyzed, and by applying widening operators to get fast convergence.

3.5 Applications of formal methods

- Their principal benefits are in reducing the number of errors in systems so their main area of applicability is critical systems:
  - ✓ Air traffic control information systems,
  - ✓ Railway signalling systems
  - ✓ Spacecraft systems
  - ✓ Medical control systems
  - ✓ Formal methods are applied in different areas of hardware and software, including routers, Ethernet switches, routing protocols, and security applications.
- In this area, the use of formal methods is most likely to be cost-effective
- Formal methods have limited practical applicability

It is more cost effective to first determine what the crucial components of the software are. These parts can then be isolated and studied in detail by creating mathematical models and verifying them.

In this course we will focus on some formal methods techniques which includes model checking and theorem proving. The first method takes a finite transition system and systematically checks whether all the desired properties hold for every state of the system. Because the number of states increases exponentially with the size of the model, this method will often have to limit itself to small variants of the system that is under

investigation. An advantage of model checking is that, when it finds a problem, it can indicate how it got into an error state. This information can be used to improve the model.

The second method uses general mathematical techniques to reason about the models. This makes it possible to reason about systems of which the number states is unlimited. For this a price must be paid: the reasoning cannot occur fully automatically. By combining the strong points of model checking; automation and finding counter examples, with the more general mathematical power of theorem proving, it takes less effort to guarantee the reliability of the investigated systems. This is a methodology in which critical software components are investigated, improved and verified, using the earlier mentioned formal methods. Subsequently, the verified models can be used to derive (fragments of) computer programs, which satisfy high reliability demands. These fragments can then replace the original components.

3.6 Software development

In software development, formal methods are mathematical approaches to solving software (and hardware) problems at the requirements, specification, and design levels. Formal methods are most likely to be applied to safety-critical or security-critical software and systems, such as avionics software. Software safety assurance standards, such as DO-178B, DO-178C, and Common Criteria demand formal methods at the highest levels of categorization.

For sequential software, examples of formal methods include the B-Method, the specification languages used in automated theorem proving, RAISE, and the Z notation.

In functional programming, property-based testing has allowed the mathematical specification and testing (if not exhaustive testing) of the expected behaviour of individual functions.

The Object Constraint Language (and specializations such as Java Modeling Language) has allowed object-oriented systems to be formally specified, if not necessarily formally verified.

For concurrent software and systems, Petri nets, process algebra, and finite state machines (which are based on automata theory) allow executable software specification and can be used to build up and validate application behaviour.

3.7 Software verification

Important considerations when dealing with a formal system:

• Soundness/Correctness.

This property states that every property that can be obtained using the formal system/calculus is semantically true in some sense.

– Slogan: “What you can prove is also true.”

• Completeness.

This property is the opposite implication of correctness. It states that for every true sentence there is also a proof in the formal system/calculus.

– Slogan: “What is true can also be proven.” • Expressive power.

– Slogan: “Can I formulate all my properties in the language?”

• Decidability.

If a formal system is decidable, then all proofs can be found automatically by a program.

Slogan: “Can a computer do my work?”

3.8 Why use Formal Methods?

- Improve quality of software system
- Fitness for purpose
- Maintainability
- Ease of construction
- Higher confidence in software product
- Reveal ambiguity, incompleteness, and inconsistency in system
- Detect design flaws
- Determine correctness
- Incrementally grows in effective solution after each iteration.
- This model does not involve high complexity rate.
- Formal specification language semantics verify self-consistency.

3.9 Some Limitations to Formal methods

Formal methods have not become mainstream software development techniques as was once predicted.

- The scope of formal methods is limited. They are not well-suited to specifying and analysing user interfaces and user interaction
- Formal methods are hard to scale up to large systems
- Time consuming and expensive.
- Difficult to use this model as a communication mechanism for non-technical personnel.

- Extensive training is required since only few developers have the essential knowledge to implement this model.

3.9 When and where to use Formal Methods?

- Introduce FM into existing systems
  - ✓ Verify critical properties
  - ✓ Facilitate maintenance and reimplementation
- Introduce FM into new systems
- Capture requirements precisely
  - ✓ Reduce ambiguity
  - ✓ Guide software development process
  - ✓ Basis for testing o Formalize requirements analysis and design

3.10 Relevant arrears of research

- Programming environments
- Formal methods in software development
- Tools that support construction of formal specifications
- Design tools that will generate formal specifications
- Problem/specification decomposition
- Procedural and data abstraction
- Synthesis of efficient code
- "Smart" user interfaces (user-friendly ones!!)
- Methods for determining reuse (of design/specifications/code)

4.0 Self-Assessment Exercise(s)

Answer the following questions:

1. Define formal methods
2. Mention some of the formal methods used in software development
3. What are the uses of formal methods in software development?
4. Differentiate between human directed proof and automated proof
5. Formal methods can be applied in the production of critical systems: Mention any 5 of them
6. Write short notes on the following:
   a) Denotational semantics
   b) Operational semantics

c) Axiomatic semantics

5.0 Conclusion

The principal benefit of the application of formal methods in crafting software product is in the reduction of errors. It is therefore advisable to deploy them in software development, especially in life and security critical systems

6 Summary

Formal methods refer to mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially important in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process. Formal methods are particularly effective early in development at the requirements and specification levels, but can be used for a completely formal development of an implementation (e.g., a program). Formal methods are best described as the application of a fairly broad variety of theoretical computer science fundamentals, in particular logic calculi, formal languages, automata theory, and program semantics, but also type systems and algebraic data types to problems in software and hardware specification and verification. It is very beneficial, though it also has it demerits.

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) Formal Methods in Software Engineering

MODULE 2: 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

Contents

Contents

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) Formal Methods in Software Engineering

Unit 2: Proposition

Contents

Contents

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 truth table 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 pVq and will be true iff either p is true, or q is true, or both p and q are true.

The operation of V is summarised in the following truth table:

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, then the 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⇔q

and will be true iff either:
1. p and q are both true; or
2. p and q are both false.

The truth table 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

¬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∧q⇒r

p∧(q⇒r)

(p∧(q⇒r))∨s

((p∧(q⇒r))∨s)∧t

whereas none of the following is:

p∧

p∧q)

p¬

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∨q) ⇒r is evaluated by:
(v(p)∨v(q)) ⇒v(r) (1)
= (F∨T) ⇒F (2)
= T ⇒F (3)
= F (4)
Line (3) is justified since we know that F∨T = T.
Line (4) is justified since T ⇒F = F. If you can't see this, look at the truth tables for ∨and ⇒.

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

$(P \rightarrow Q) \vee (Q \rightarrow P)$

P	Q	$P \rightarrow Q$	$Q \rightarrow P$	**Result** $(P \rightarrow Q) \vee (Q \rightarrow P)$
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) Formal Methods in Software Engineering

Unit 3: Predicates

Contents

Contents

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 by the 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) Formal Methods in Software Engineering

Unit 4: Sets

Contents

Contents

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.

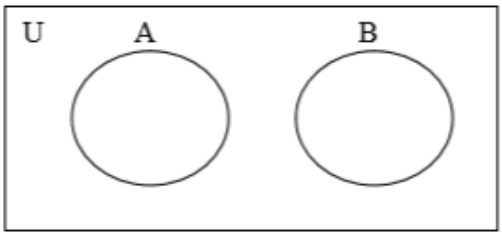
All elements in a set must be unique. In the Venn Diagrams the individual small letter element name are prefix with a dot. Capital Letter names is use to group many duplicates elements (sets) do not have a dot. The sequence or arrangement of the elements is not important.

There are two (2) ways to describe elements in a set.

1. Using a rule or semantic description:

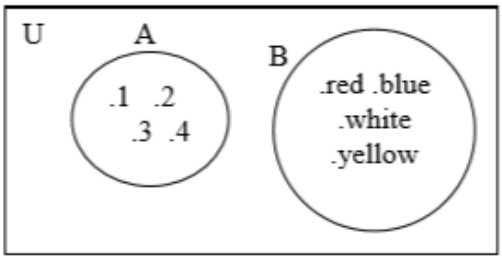
   A is the set whose members are the first four positive integers.

   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

$F = \{n2 - 4 \mid n \text{ is an integer; and } 0 \leq n \leq 19\}$

Or

$F = \{n2 - 4 : n \text{ is an integer; and } 0 \leq n \leq 19\}$

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

$| \{1, 3, 9, 15\} | = 4$

$| \{1, 2, 3, \ldots\} | = \infty$

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 ∈ 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}

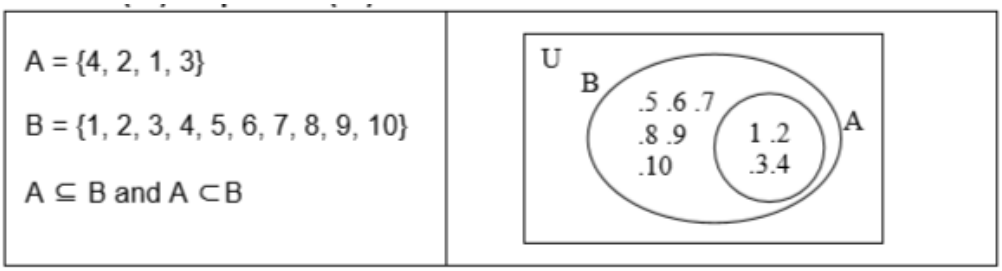
If A is a member of B, this is denoted A ∈ B.

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 (⊇)

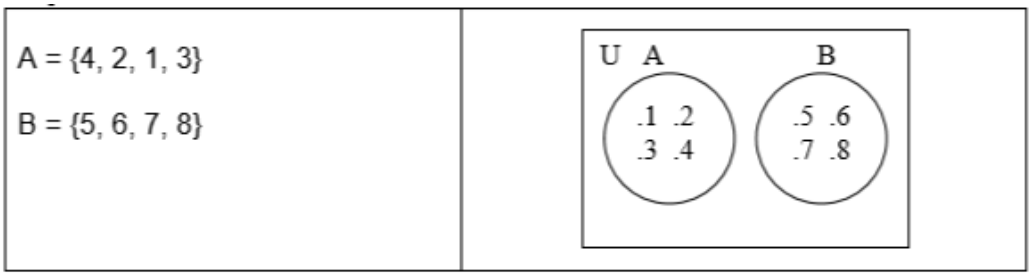


If every member of set A is found inside set B, then A is a subset of B (A ⊆ B). If set B has every member of A and more then B is a super set of A (B ⊇ A) This kind of relationship is also known as inclusion or containment.

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



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

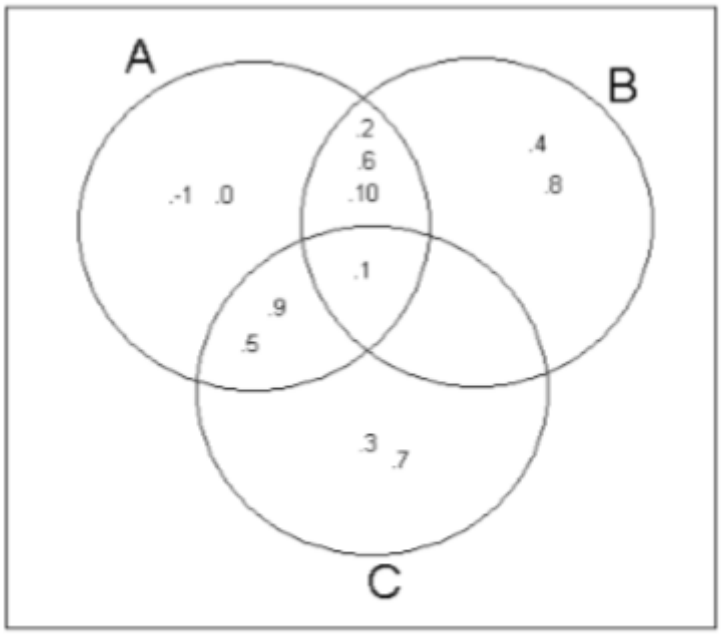
Given the following sets:

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) Formal Methods in Software Engineering

Unit 5: Series or Sequence

Contents

Contents

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 = 1398100

Given X = = 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576, Calculate the sum of X?

Sum X_{10} $= 4 (1 - 4^{10}) / 1 - 4$
$= 4 (1-1048576) / -3 = 4 (-1048575) / -3$
$= -4194300 / -3 = 1398100$

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 $X_{\infty} = 4 / 1 - 4 = 4 / -3 = -1.3333333$

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 sequence were 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) Formal Methods in Software Engineering

MODULE 3: FORMAL METHODS CONTINUES

This module is divided into four (4) units
Unit 1: Mathematical Proof
Unit 2: Testing
Unit 3: Application to Formal Specification
Unit 4: Z Notation

Unit 1: Mathematical Proof

Contents

Contents

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 to C 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) Formal Methods in Software Engineering

Unit 2: Testing

Contents

3.0 Main Content

Contents

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) Formal Methods in Software Engineering

Unit 3: Application to Formal Specification

Contents

Contents

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 assists in 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) Formal Methods in Software Engineering

MODULE 4: SOFTWARE DEVELOPMENT OVERVIEW

This module is divided into four (4) units

Unit 1: **S**oftware Development and **S**oftware Engineering

Unit 2: **Software Development Life Cycle**

Unit 3: **Software Project Management**

Unit 4: Software Requirements

Unit 1: Software Development and **S**oftware Engineering

Contents

Contents

2.0 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. **S**oftware Development process is the practice of organising the design and construction of software and its deployment in context. Note

that **S**oftware development and **S**oftware 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 S**oftware**

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 S**oftware 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.

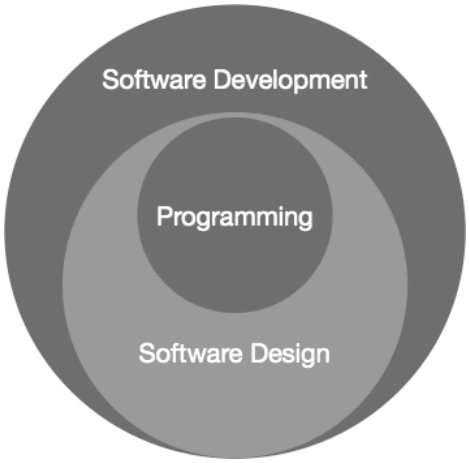
3.2.2 E-Type software evolution

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

Contents

Contents

3.0 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.

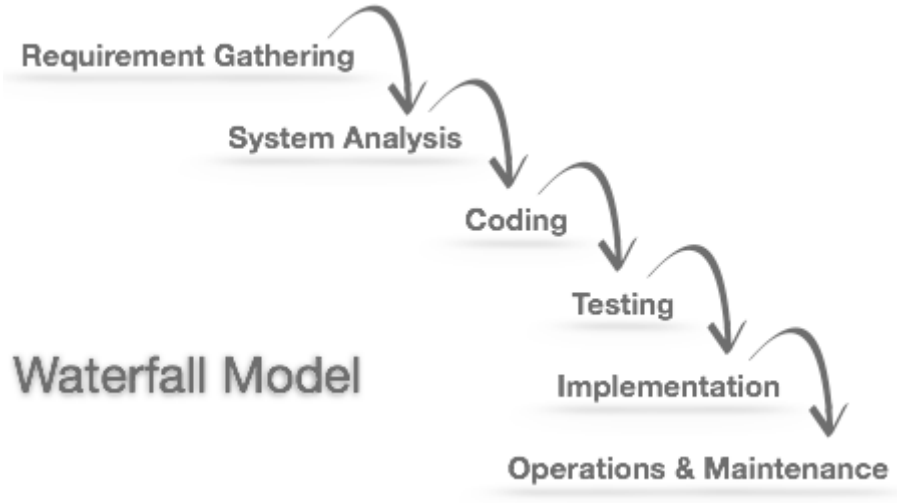
10 Disposition: As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense up gradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software components, closing down 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.

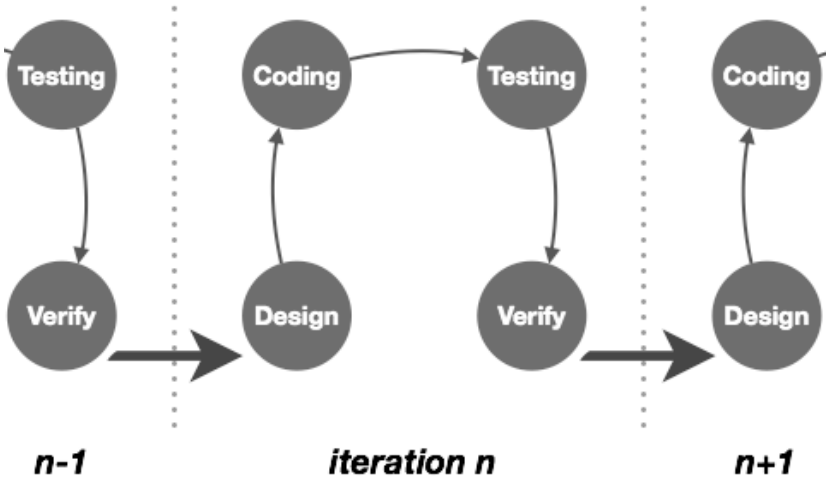


This model assumes that everything is carried out and taken place perfectly as planned in the previous stage and there is no need to think about the past issues that may arise in the next phase. This model does not work smoothly if there are some issues left at the previous step. The sequential nature 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.



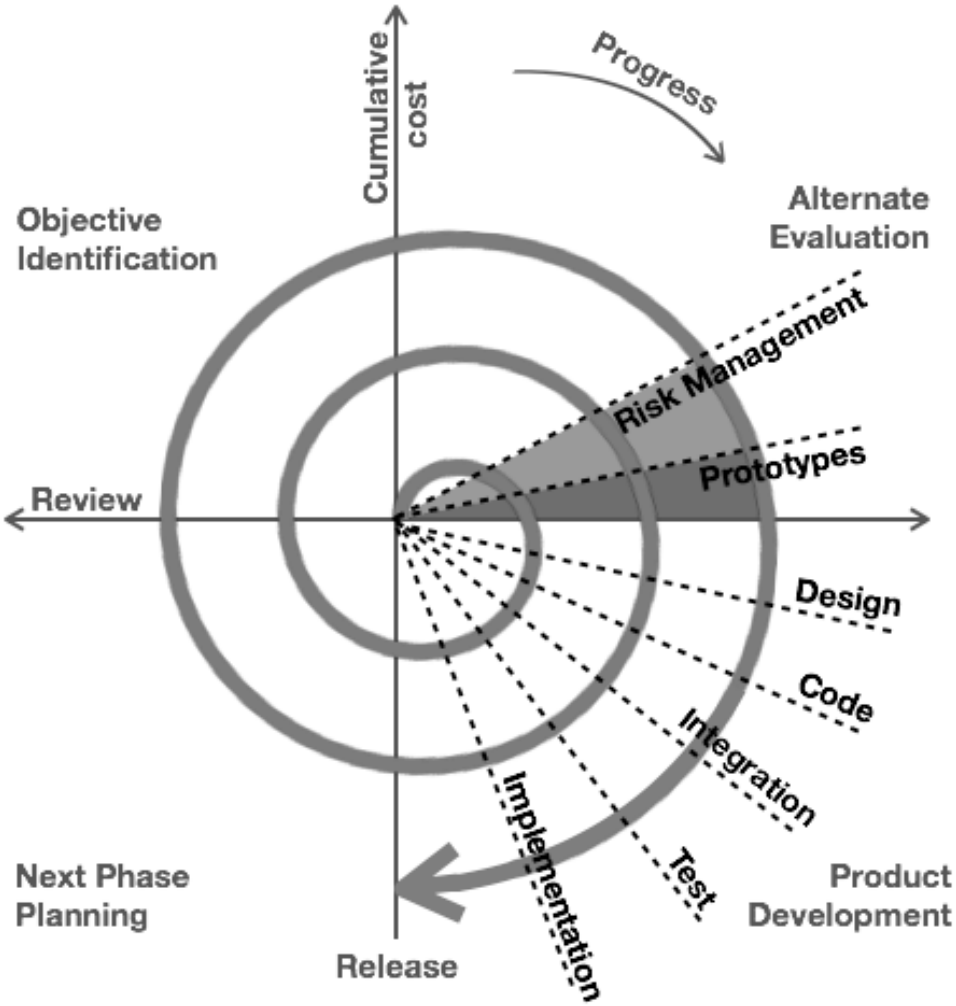
The software is first developed on very small scale and all the steps are followed which are taken into consideration. Then, on every next iteration, more features and modules are designed, coded, tested and added to the software. Every cycle produces a software,

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

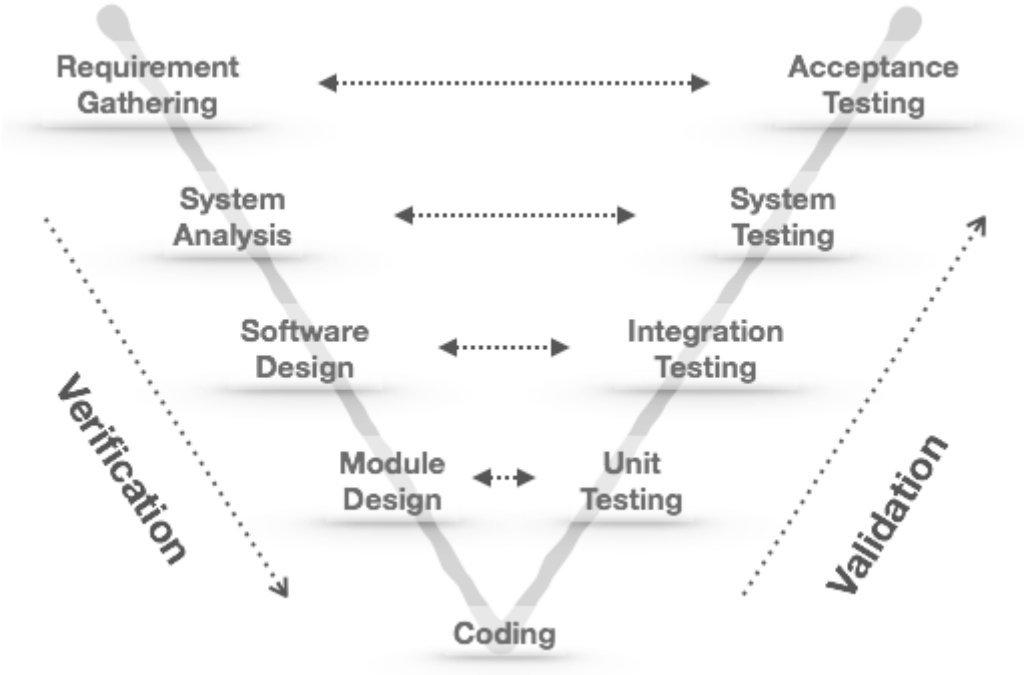
Spiral model is a combination of both, iterative model and one of the SDLC model. It can be seen as if you choose one SDLC model and combine it with cyclic process (iterative model).



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.

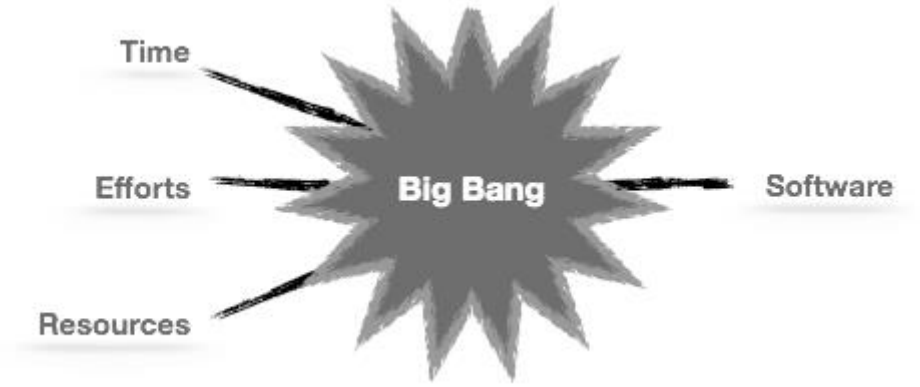


At every stage, test plans and test cases are created to verify and validate the product according to the requirement of that stage. For example, in requirement gathering stage the test team prepares all the test cases in correspondence to the requirements. Later, when the product is developed and is 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 of software 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

Contents

Contents

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 base lined, i.e. baseline is a measurement that defines completeness of a phase. A phase is base lined 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 base lined. 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.

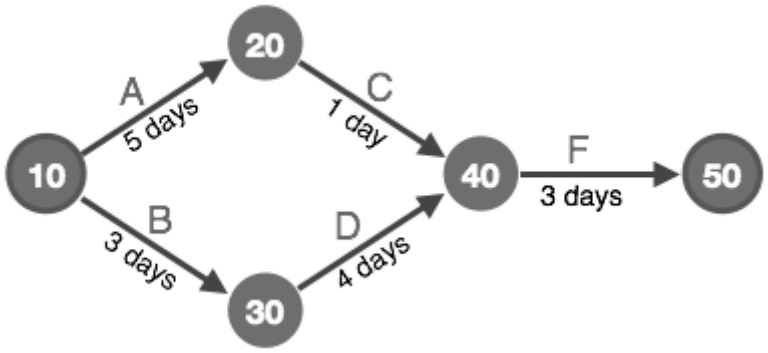
There are tools available, which aid for effective project management. A few are described:

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

Contents

Contents

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

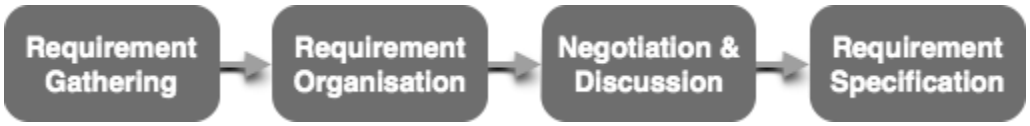
After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked 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 5: OVERVIEW OF SOFTWARE DESIGN, ANALYSIS AND DESIGN TOOLS, DESIGN STRATEGIES AND USER INTERFACE BASICS

This module is divided into four (4) units

Unit 1: Software Design Basics

Unit 2: Analysis and Design tools

Unit 3: Software Design Strategies

Unit 4: Software User Interface Design

Unit 1: Software Design Basics

Contents

Contents

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

Contents

Contents

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

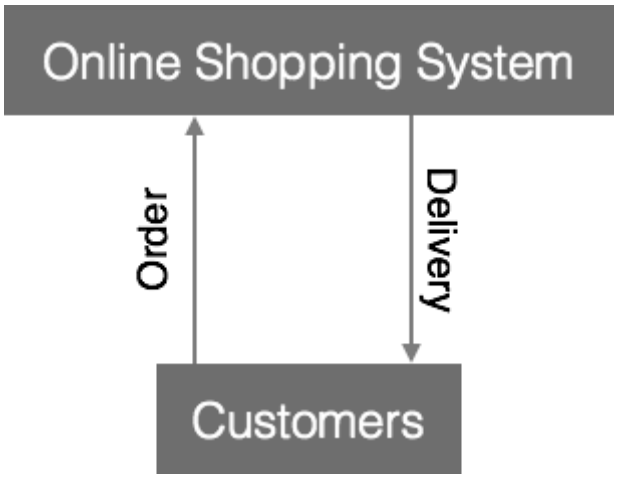
DFD can represent Source, destination, storage and flow of data using the following set of components -

- **Entities** - Entities are source and destination of information data. Entities are represented by rectangles with their respective names.

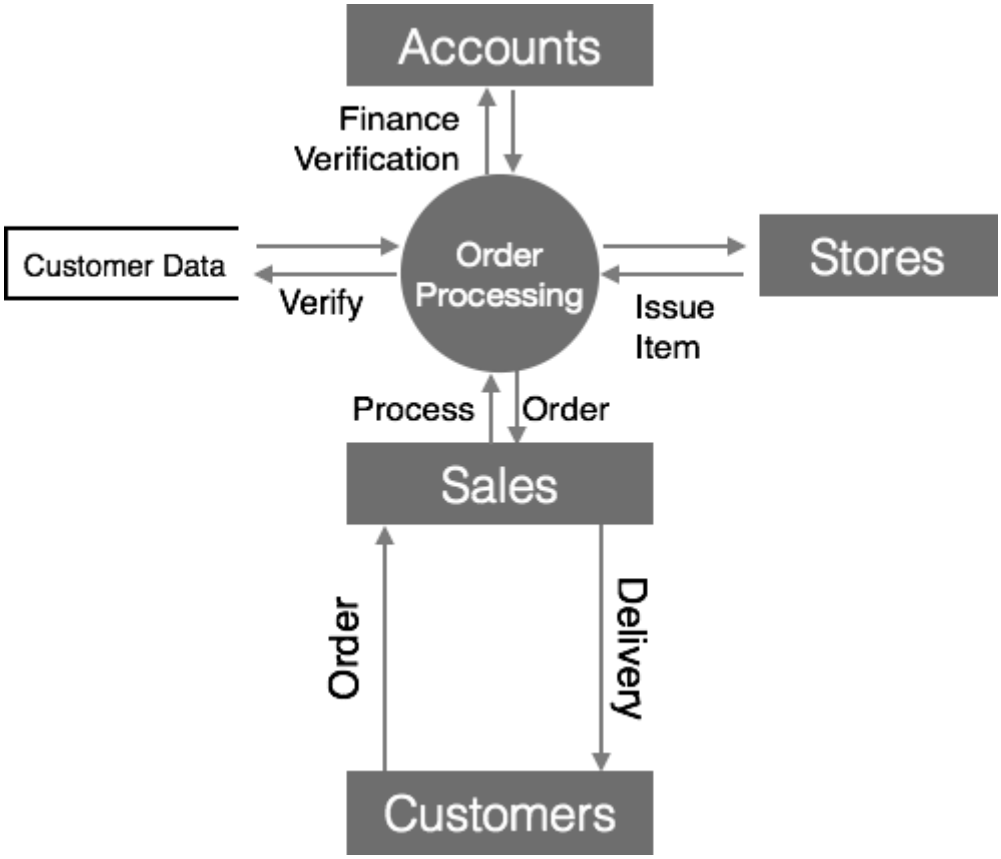
- **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
- **Data Storage** - There are two variants of data storage - it can either be represented as a rectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
- **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.

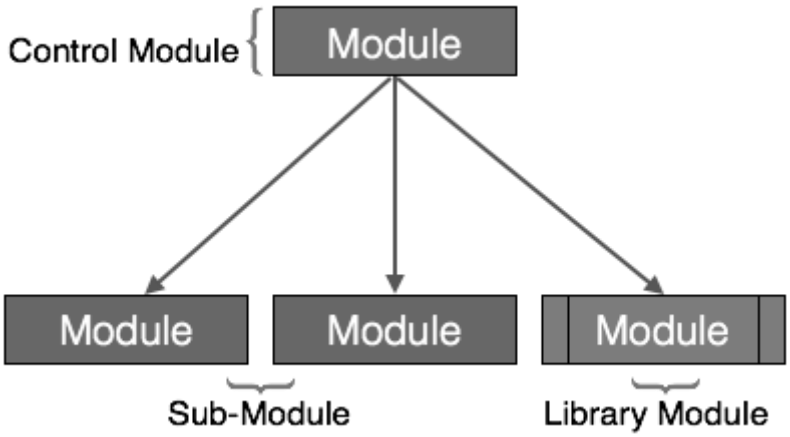
3.2 Structure Charts

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.

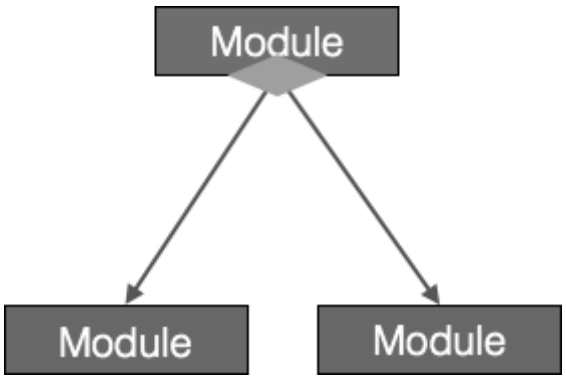
Structure chart represents hierarchical structure of modules. At each layer a specific task is performed.

Here are the symbols used in construction of structure charts -

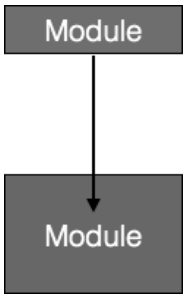
- **Module** - It represents process or subroutine or task. A control module branches to more than one sub-module. Library Modules are re-usable and invokable from any module.



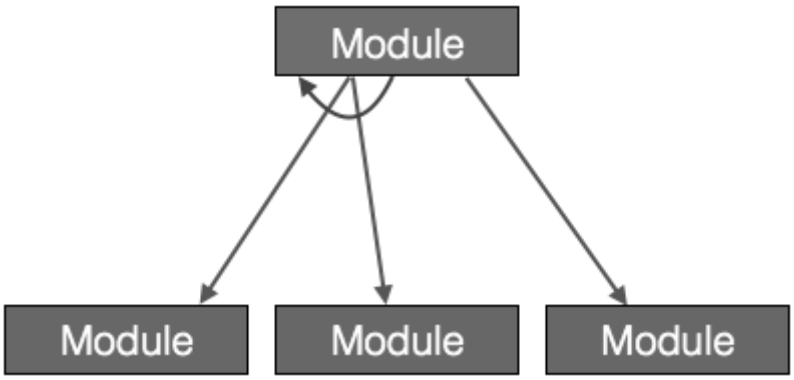
- **Condition** - It is represented by small diamond at the base of module. It depicts that control module can select any of sub-routine based on some condition.



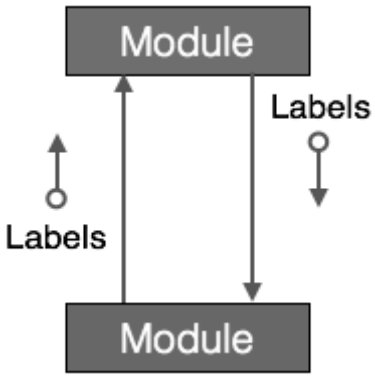
- **Jump** - An arrow is shown pointing inside the module to depict that the control will jump in the middle of the sub-module.



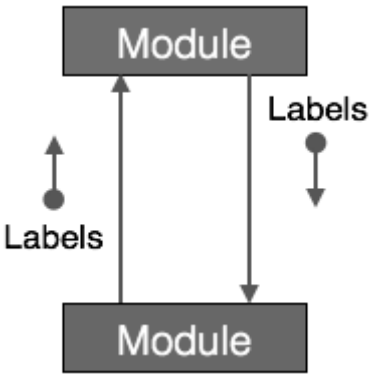
- **Loop** - A curved arrow represents loop in the module. All sub-modules covered by loop repeat execution of module.



- **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.

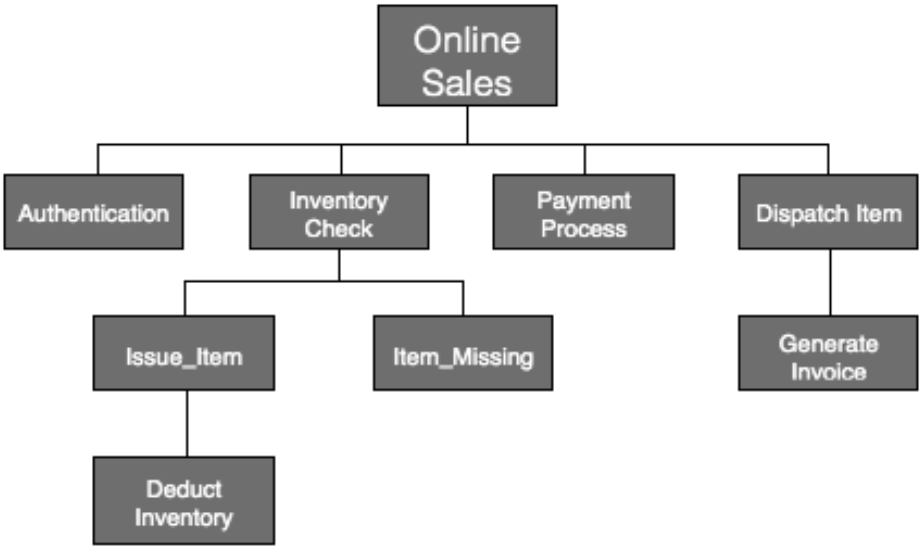


3.3 HIPO Diagram

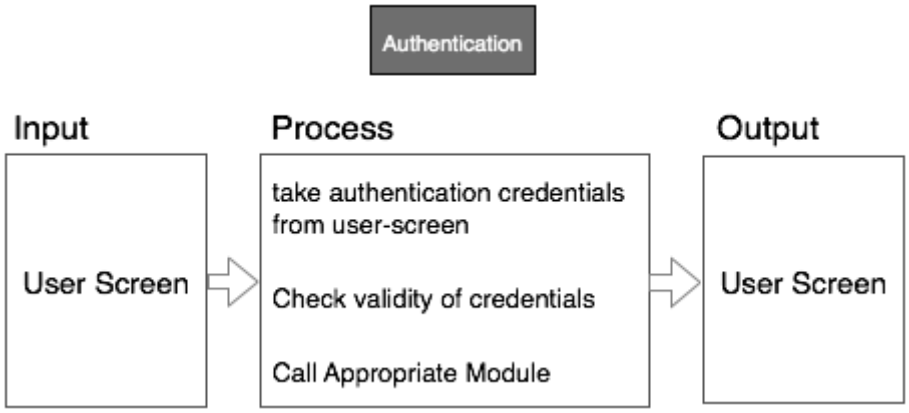
HIPO (Hierarchical Input Process Output) diagram is a combination of two organized method to analyze the system and provide the means of documentation. HIPO model was developed by IBM in year 1970.

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.



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.

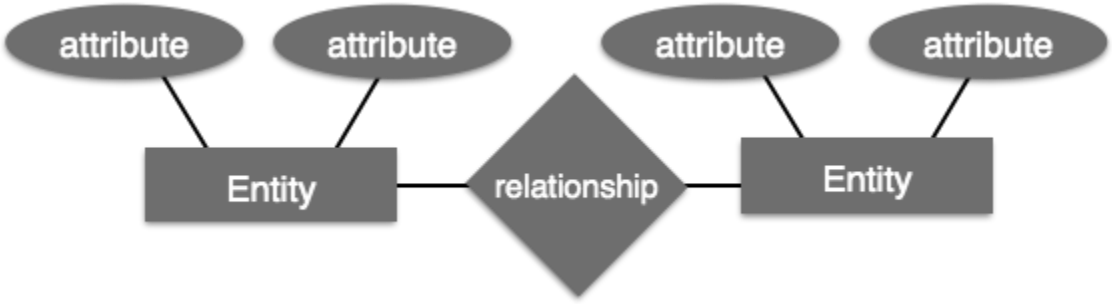
We list all possible problems under column conditions and the prospective actions under column Actions.

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

Contents

Contents

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

Contents

Contents

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)

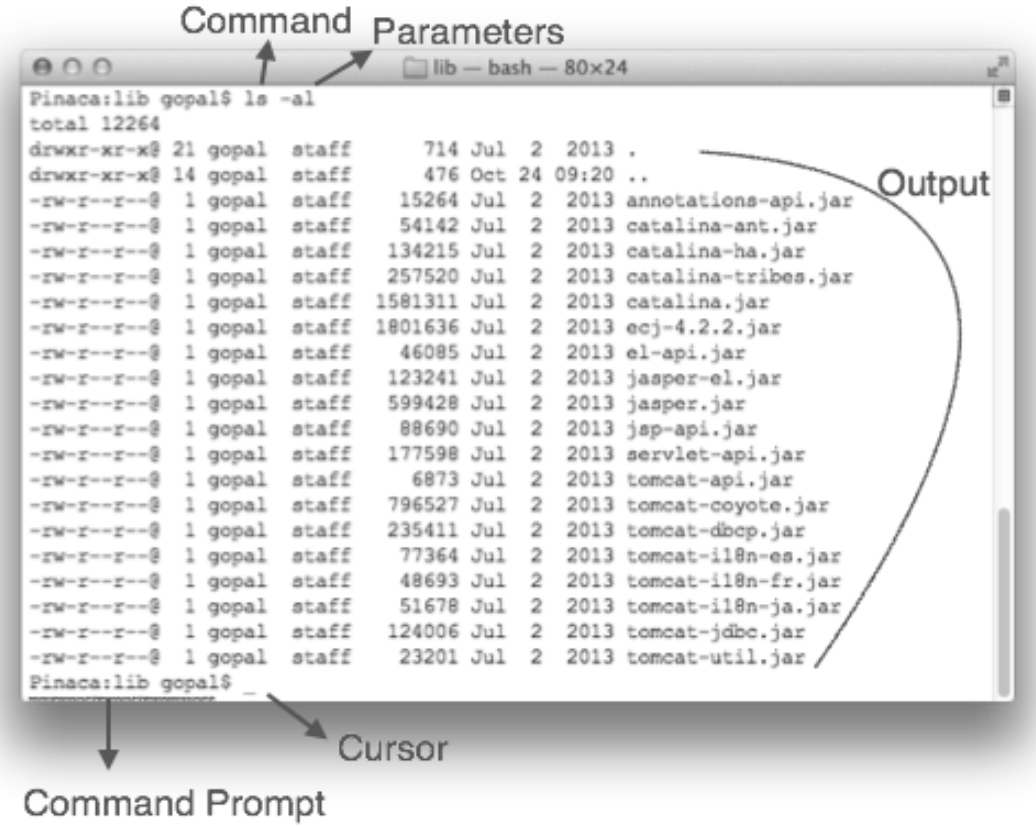
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

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



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

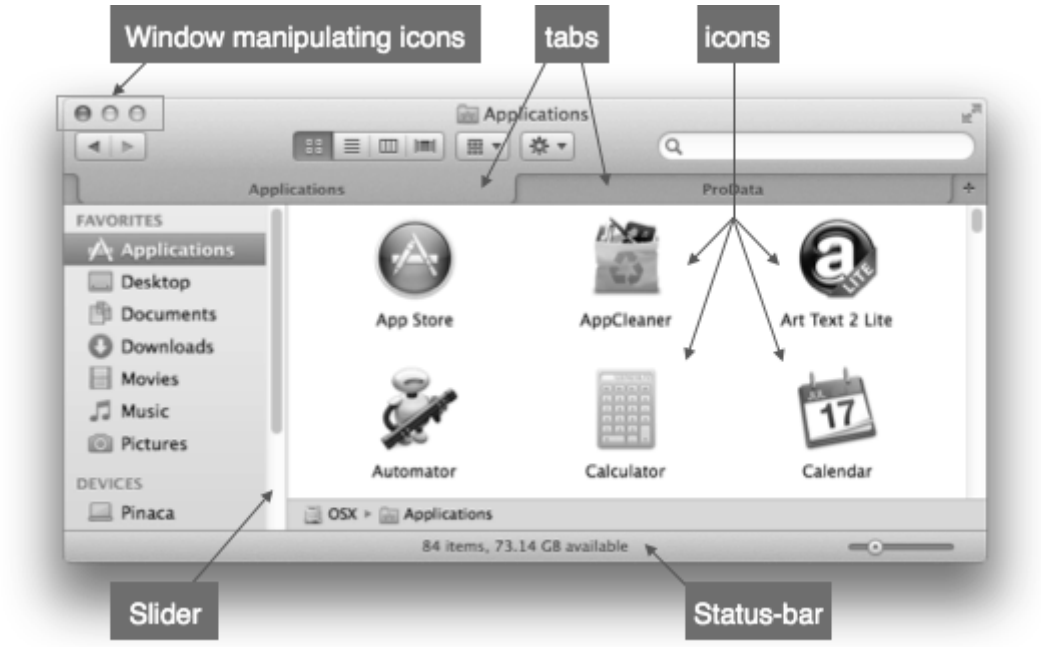
Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

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.

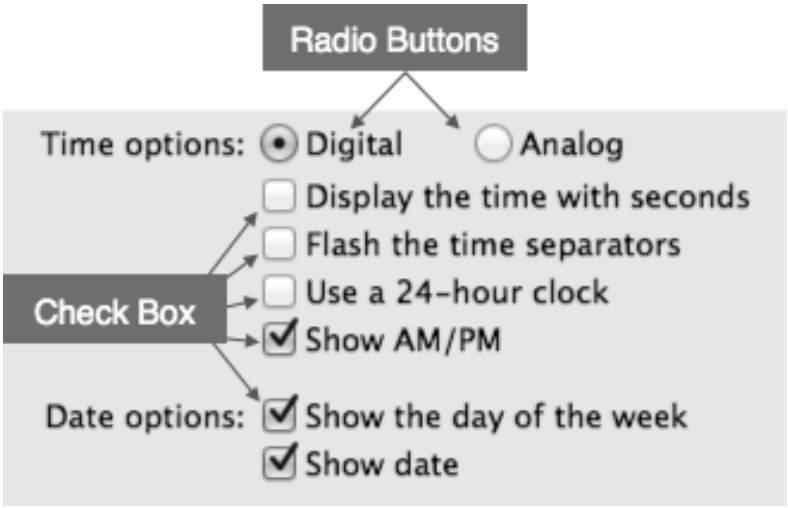
- **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers 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

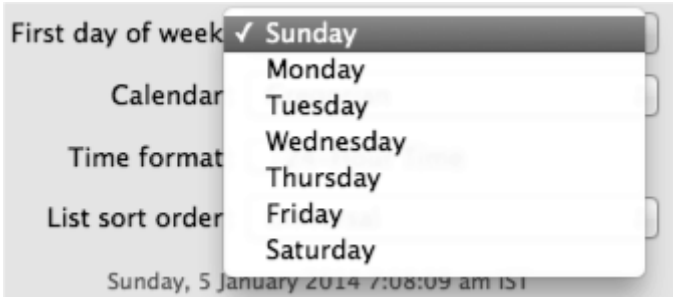
A GUI of an application contains one or more of the listed GUI elements:

- **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.

- **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.



- **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.

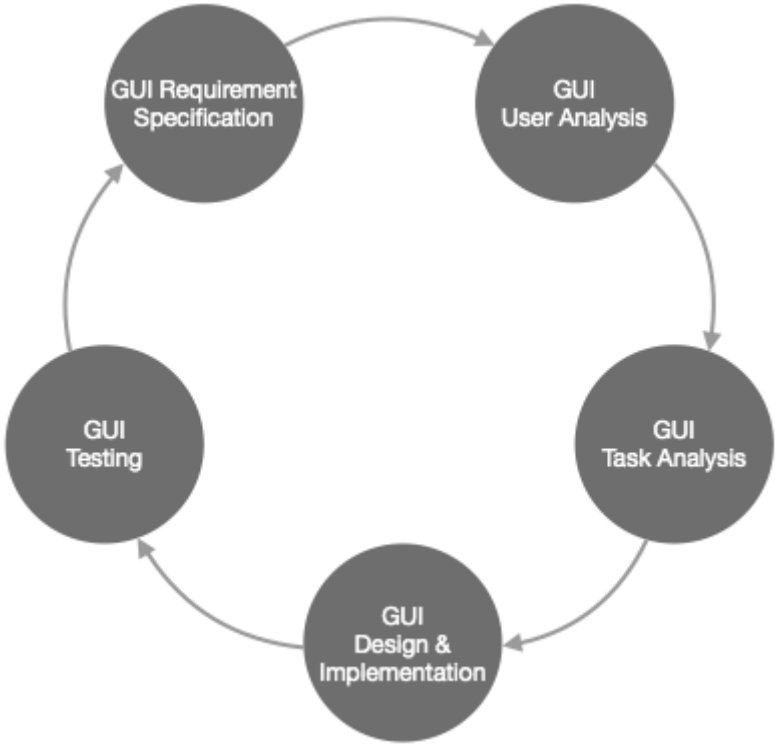


3.4 Other impressive GUI components are:

- 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.

MODULE 6: OVERVIEW OF DESIGN COMPLEXITY, SOFTWARE IMPLEMENTATION, TESTING, MAINTENANCE AND CASE TOOLS

This module is divided into five (5) units

Unit 1: Design Complexity

Unit 2: Software Implementation

Unit 3: Software Testing

Unit 4: Software Maintenance

Unit 5: Software Case Tools

Unit 1: Design Complexity

Contents

Contents

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.

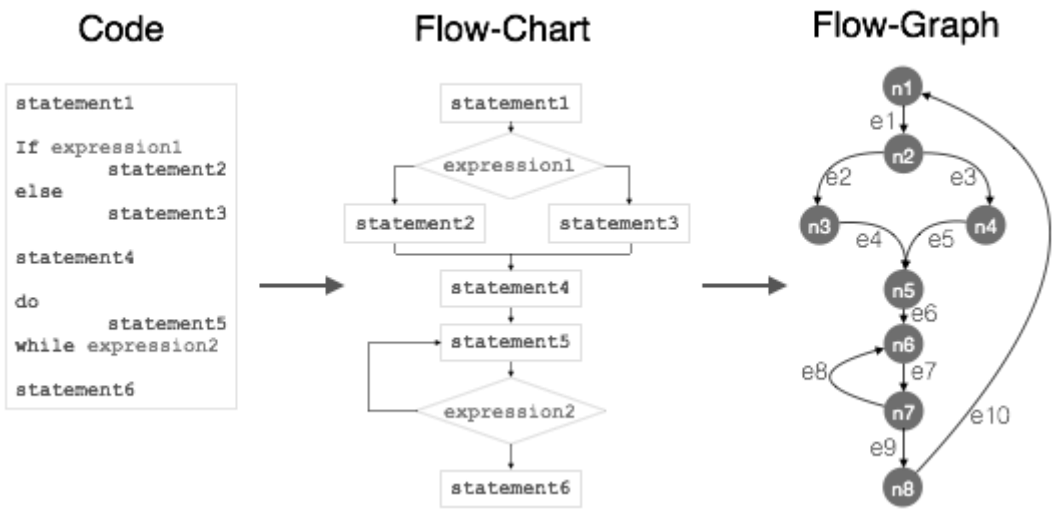
Process to make flow control graph:

- 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

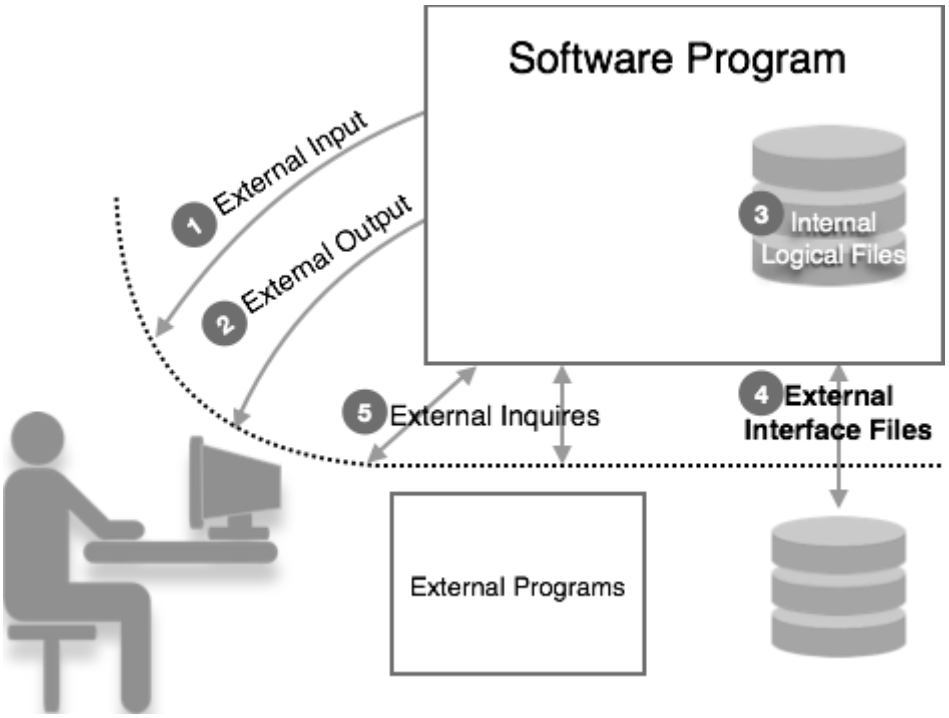
e = 10 n = 8 Cyclomatic Complexity = 10 - 8 + 2 = 4

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,

Delivered Function Points (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

Contents

Contents

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

Contents

Contents

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.

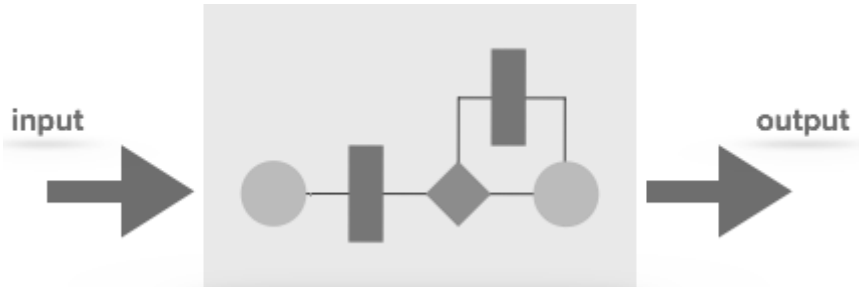
In this testing method, the design and structure of the code are not known to the tester, and testing engineers and end users conduct this test on the software.

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 and White-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

Contents

Contents

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.

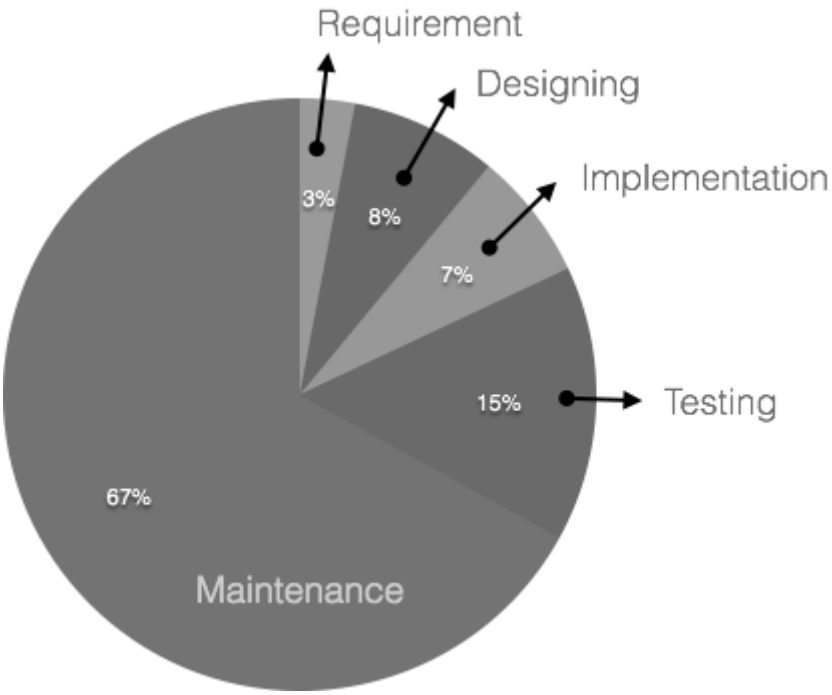
3.1 Types of maintenance

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

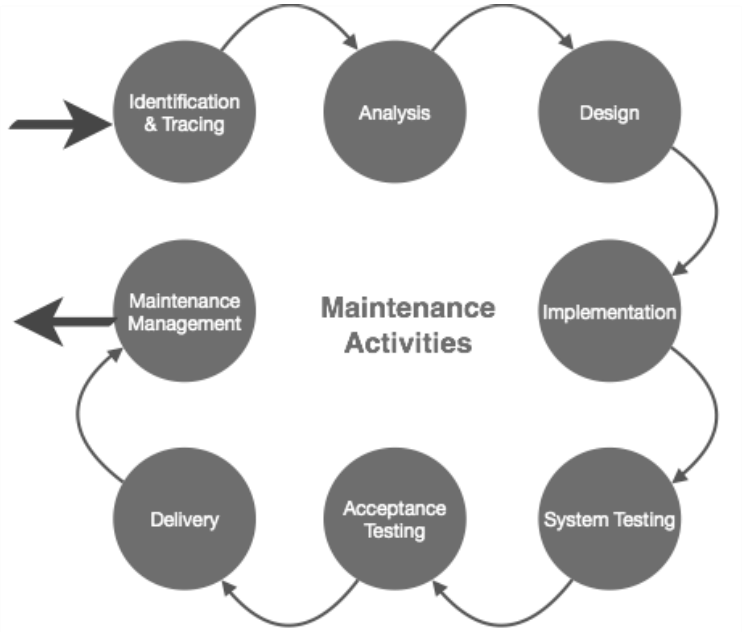
- The standard age of any software is considered up to 10 to 15 years.
- Older software, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced software on modern hardware.
- As technology advances, it becomes costly 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.

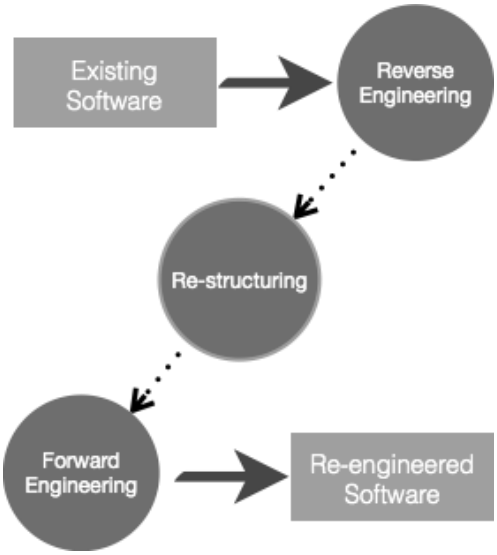
3.6 Software Re-engineering

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.

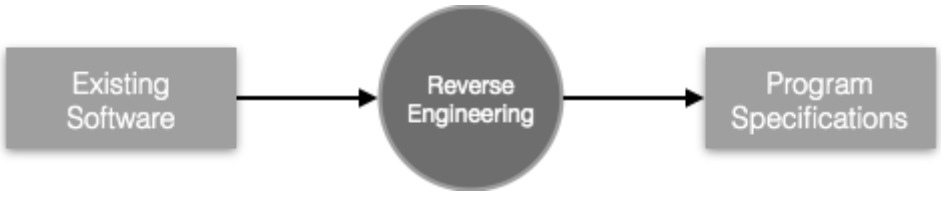
3.6.2 Terminologies in Software Re-Engineering

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.

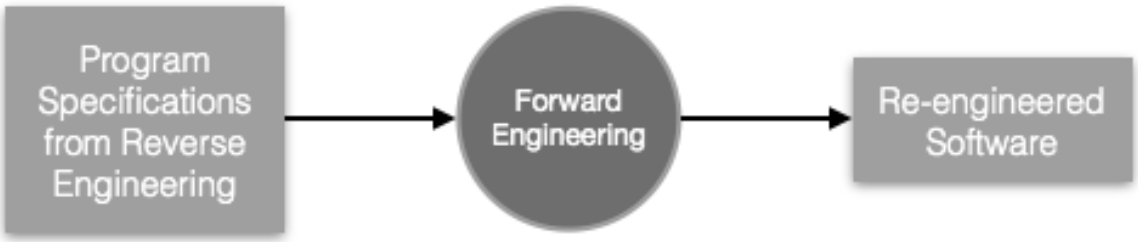
Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

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

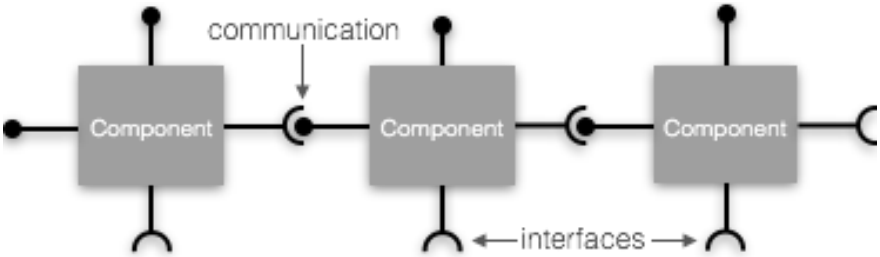
The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.

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).



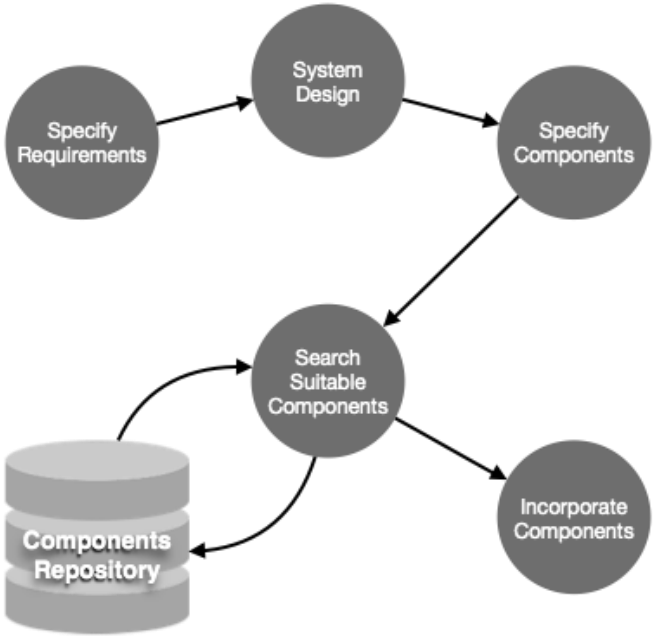
Re-use can be done at various levels

- **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

Contents

Contents

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.

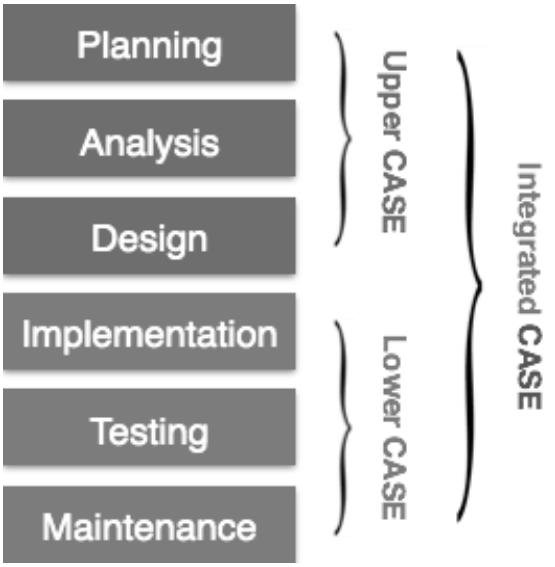
There are number of CASE tools available to simplify various stages of Software Development Life Cycle such as Analysis tools, Design tools, Project management tools, Database Management tools, Documentation tools are to name a few.

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.