2024

Software Engineering

Dr. Babasaheb Ambedkar Open University

Software Engineering

Course Writer:

Dr. Kamesh R. Raval
Assistant Professor,
Som-Lalit Institute of Computer Applications

Ms. Sejal Vaghela
Assistant Professor,
Lokmanya College of Computer Applications

Content Reviewer and Editor:

Prof. (Dr.) Nilesh K. Modi
Professor & Director School of Computer Science,
Dr. Babasaheb Ambedkar Open University

ISBN No:

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad While all efforts have been made by editors to check accuracy of the content, the representation of facts, principles, descriptions and methods are that of the respective module writers. Views expressed in the publication are that of the authors, and do not necessarily reflect the views of Dr. Babasaheb Ambedkar Open University. All products and services mentioned are owned by their respective copyrights holders, and mere presentation in the publication does not mean endorsement by Dr. Babasaheb Ambedkar Open University. Every effort has been made to acknowledge and attribute all sources of information used in preparation of this learning material. Readers are requested to kindly notify missing attribution, if any

Software Engineering

Contents

BLOCK 3: SYSTEM ANALYSIS AND DESIGN

UNIT 7 STRUCTURED ANALYSIS MODELING

Objectives, Structured analysis, Data Flow Diagram (DFD), Example of DFD, Entity Relationship Diagram, Types of relations, Example of ERD, Let Us Sum Up

UNIT 8 OBJECT-ORIENTED ANALYSIS AND DESIGN

Objectives, Basic terms of object-oriented analysis, UML diagrams, Use-case diagram, Class diagram, Sequence diagrams, Requirement management, Analysis modeling, Let Us Sum Up

UNIT 9 UML-DIAGRAM OF SYSTEM – A CASE STUDY

UNIT 10 SOFTWARE DESIGN

Objectives, Feature of good software design, Design concepts, Cohesion and Coupling, Design modeling, Pattern based software design, Let Us Sum Up

---

BLOCK 4: SOFTWARE TESTING

UNIT 11 SOFTWARE TESTING CONCEPTS

Objectives, Introduction, SDLC, SDLC models, Quality concepts, Verification and Validation, Goals of software testing, Static and Dynamic testing, Let Us Sum Up

UNIT 12 BLACK-BOX TESTING

Objectives, What is black-box?, Need for black-box testing, Advantages and disadvantages of black-box testing, Boundary value analysis, Equivalence class partitioning, Decision tables testing, Let Us Sum Up

UNIT 13 WHITE-BOX TESTING

Objectives, White-box testing, Need of white-box testing, Advantages and disadvantages of white-box testing, Black-box vs White-box testing, Logic coverage criteria, Basis path testing, Let Us Sum Up

UNIT 14 SOLVED PROGRAMS-III

Objectives, Unit testing, Integration testing, System testing, Performance testing, Acceptance testing, Let Us Sum Up

---

Dr. Babasaheb
Ambedkar
Open University

BCAR-402

Software Engineering

BLOCK1: SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

BLOCK 1: SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

Block Introduction

In this block-1 of the Software Engineering, we have tried to emphasis on: What is Software Engineering? And What is the importance of it? Basically, when we are working on the large software products, then entire development process is divided into number of phases which are called Software Development Life Cycle (SDLC) phase. First an engineer has to evaluate feasibility study where project is Economically, Operationally and Technically feasible or not is inspected. Then information related to project has to be gathered and analyze. As an output of this several documents are produced which will help a project manager to manage the entire project which larger and complex.

We have also described, that the phases of SDLC in implemented in different way depending on the type of the project, which called models of SDLC implementation. For which type of project which model of SDLC has to be used in served in the first block of Software Engineering.

Block Objective

The objective of the block is to explain what is an engineering approach to build a large and complex software system. Students will able to learn that the software development cannot be accomplished like small program development. Detail study of requirement analysis is required to build software.

By learning this block of software engineering student will learn about different phases of Software Development Life Cycle, and tasks which an engineer has to perform during each phase of SDLC. Reader of this block, will know software development is a complex process and cannot be accomplished in one go. Using work break down structure, the entire development process is divided into phase, phases are divided into several tasks, and further tasks into different activities.

Different software projects have different challenges, depending upon nature of the problem, SDLC phases has to be implemented in different ways which are called models. This block servers detail knowledge of different types of models and depending upon project type, which model has to be adopted or suitable.

We hope, this block will clear the idea of software engineering process, it's requirement so that student can become ready for industrial development.

Block Structure

BLOCK1: SOFTWARE DEVELOPMENT LIFE CYCLE AND MODELS

UNIT1 INTRODUCTION TO SOFTWARE ENGINEERING

Objectives, Software Engineering – Evolution and Impact, Software products vs Programs, Importance of software engineering, Emergence of software engineering, Let Us Sum Up

UNIT2 SOFTWARE DEVELOPMENT LIFE CYCLE

Objectives, Why to use life cycle models?, Entry and Exit phase criteria, Phases of SDLC, Feasibility study, Requirement gathering and analysis, Design phase, Coding, Testing, Maintenance, Let Us Sum Up

UNIT3 SOFTWARE DEVELOPMENT MODELS

Objectives, Introduction, Operators and Expressions, Special Operators, Arithmetic Expressions, Operator precedence and associativity, Mathematical functions, Let us Sum Up

Unit 1: Introduction to Software Engineering

1

Unit Structure

1.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about evolution and impact of software engineering
- Understand about software crisis
- Understand difference between software products and programs
- Learn Emergence of Software Engineering
- Understand the importance of Software Engineering

1.2 INTRODUCTION

Since last seventy years, computers have been used for commercial purposes. Earlier computers were too slow and not much sophisticated. Slowly and gradually speed and sophistication have increased and the price of computers have reduced dramatically. Improvement in the speed, reliability and sophistication and reduction in the cost, has given some technological breakthroughs on regular intervals.

Usually, faster computers can run more sophisticated programs. In every aspect of hardware improvement, more reliable and complex and cost-effective software programs are also expected. The responsibilities of providing large, complex software systems into cost-effective and efficient manner is given to the software engineers. Software engineers can provide cost-effective and efficient solution for large complex software system by learning from the mistakes made in the past projects. Which means software engineering is an engineering approach in the development of software products.

Now, what is this engineering approach? To understand this, consider if a person what to make a small house or needs some repairing work into the house, we are calling to contractor and he can do the work in efficient way. But think, is he able to make large buildings? Answer is no. He doesn't have sufficient knowledge of the portion of cement, concreate and send in the mixture (mortar), which will provide sufficient strength to the large building. Or we can say, don't have sufficient knowledge to manage entire work of construction of a large building. If the instead of contractor the same work is given to civil engineer, then that person can do proper planning, space utilization, having proper knowledge of raw materials to be used which provide

necessary strength to the building which will not be collapse for longer period of time. By this example, we mean to solve small sized problem any programmer can write the program, by in the production of large and complex software system an engineering approach is to be used, in which the following characteristics has to be there:

- More use of past experience. Past experience is systematically arranged and used theoretical basis in the existing projects.
- During designing, for each complex problem or conflicts situations, several alternative approaches may be proposed and optimal approach have been chosen to solve the problem.
- To reduce the overall cost of the software system (as economical concern), cost-effective alternative has to be chosen.

1.3 SOFTWARE ENGINERING – EVOLUTION AND IMPACT

In this section we will discuss how software industry has adopted the engineering approach with respect to the time and impact of it on a software project.

1.3.1 Evolution of software engineering

Number of researchers and software professionals have given their contribution in these evolutions, since last 70 years. The early programmers used an experimental programming style. In that every programmer evolves his own development style from guided by the organization, his past experience, impulses and desires. Usually, students are doing the same thing while writing the program. This experimental programming style is to be considered as art, which mostly guided by their organization. This art is transformed into craft, where programmers have started to learn from the mistakes they made in the past. Still, this point the past experience and mistakes are unorganized. In modern software industry, an engineering approach is used, where all past experiences and past mistakes are stored in the organized way and used as theoretical concepts in the existing system development process.

The following figure, shows how our software industry is transformed from art to craft and finally craft to engineering approach, which help to design more complex software products in efficient and cost-effective manner.

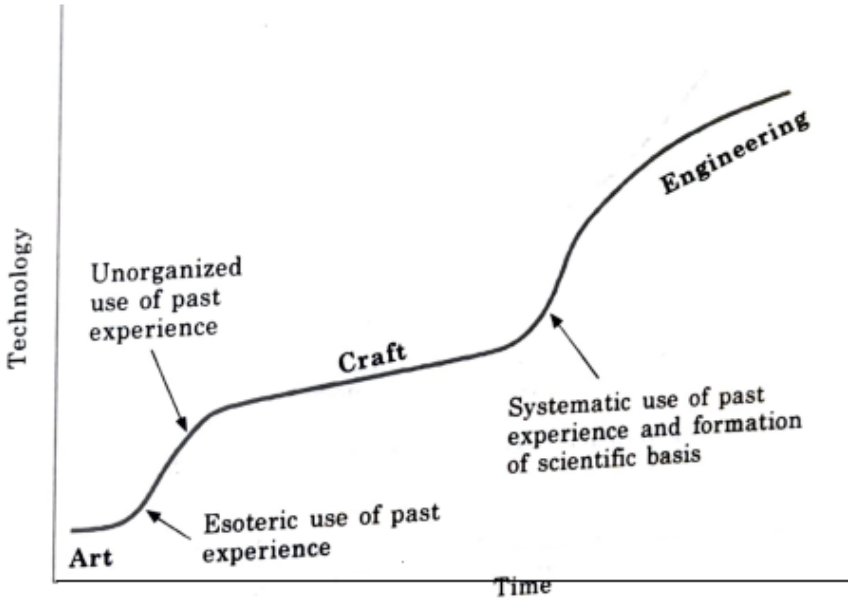


Fig: 1.1 Development of Technology with Time

1.3.2 What is Software Crisis?

Compare to hardware, software products face many challenges. Software products are difficult to alter (change), debug and enhance. They often fail to fulfill user's requirement and sometime uses resources in non-optimal way. When hardware industries produce more powerful, optimal and cost-effective products day to day. Because of the cost of hardware is decreasing day to day, software industry also has to reduce the cost of the software products.

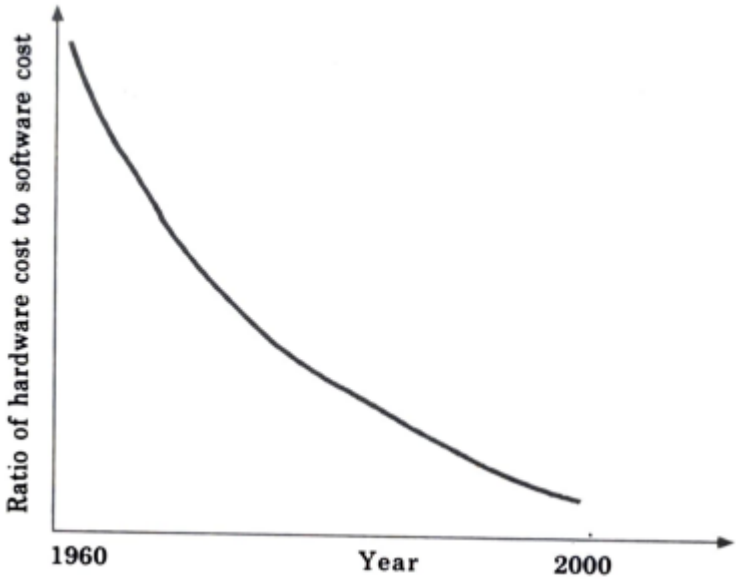


Fig: 1.2 Change in the relative cost of hardware and software over time

Today, when we brought any hardware product the essential software which runs on it comes for free. If this trend continues, we might soon have a rather amusing scenario. This effect where the prices of software products are being reduced day to day is called software crisis. But what factors have contributed to the making of the present software crisis? Unfortunately, there are many factors. Lack of necessary training in the software engineering, low productivity improvement, larger problem size increasing skill shortage are several reasons for this. But is the medicine to this problem? It is believed that only customer satisfaction can possibly solve the present software crisis problem. Software engineers need to use engineering approach with further advancements in the software industry.

1.4 SOFTWARE PRODUCTS vs PROGRAMS

Programs are usually developed by individual for their personal use. Therefore, programs are smaller in size and it should have some limited functionalities. Here, the author of the program himself maintain his program and usually a program should not have good user interface. Whereas, the ***software*** product has multiple users and therefore it has good user interface, proper user manual and good quality of documentation support is available. Because multiple users are using the software product, it is systematically designed, carefully implemented and thoroughly tested. A software product is usually having large size (more functionalities), it is developed by group of developers and engineers as a team. The person who writes the code for the program is called ***programmer*** and a person who develop a software product is called ***software engineer.*** Because of the large software developed my many software engineers as a team, they need to adopt systematic methodology. Otherwise, it will become difficult to understand the work carried out by another person from the group.

1.5 IMPORTANCE OF SOFTWARE ENGINEERING

Let us now discuss what skill you could be acquiring after learning this course of software engineering. The first and important skill you could be acquiring after learning this course is, how develop large software products. In this course we will learn, how the larger software product is decomposed into smaller and manageable parts. A major part of problem lies in the exponential growth in the perceived complexity and difficulty with program size if one attempts to write the program for a

problem without suitable decomposition the problem. This will increase the efforts of development, time to develop the software and finally cost of the software product.

After learning this course, you will understand the problem size in better way. You can breakdown the larger problem into small manageable modules and further you can estimate the size of the software, efforts to develop the software, time to develop the software and cost of that software product. Not only that, this knowledge of software engineering enables you to optimize effort, time and cost of the software. As a being a software engineer, this knowledge helps you to produce better and quality-oriented software products.

1.6 EMERGENCE OF SOFTWARE ENGINEERING

We have already discussed that software engineering techniques have progressed over many years as an outcome of number of innovations and buildup of experience of program writing. Let us discuss briefly, these innovations and experiences of programming which have given their contribution in the software engineering.

1.6.1 Early Computer Programming

Early commercial computers were too slow in speed and very basic compare to our today's computers. Even very simple processing activity took significant computation time on those computers. Because of this fact, programs during that time were very small in size and were not much sophisticated. Those programs were mostly written in assembly programming language. The length of the program was few hundreds of lines. Different programmers are writing their programs in their own way and their own style. All programmers were developed their own style of writing their programs. This type of ad hoc programing writing style is called exploratory programming style.

1.6.2 High-Level Programming languages

Computers became more faster with the invention and use of the semiconductors in the computers. When the more powerful computers are available, that makes possible to write more complex and large programs. During this time, high-level languages such as FORTRAN (FORmula TRANslation), COBOL (Common

Business Oriented Language) etc. are invented. This high-level programming languages reduce the effort of the software development, because they are easy to learn, instructions are mostly in the English language and they make possible where one line of high-level instruction is same as two or three lines of low-level language programming code.

1.6.3 Control or Flow-based programming

Exploratory programming style is not sufficient for the program having complex logic and larger size. Programmers found that it is difficult to find the errors, and difficulty in understanding the logic of the programs written by someone else. To deal with this problematic situation senior programmers advised to their juniors to pay more attention on the control flow of the program. A control flow structure of the program indicates the sequence in which the program statements are executed. Flow-chart is a technique developed during this time to represent the programming sequence in pictorial way, which provides ease explanation about control flow sequence of the program. Even today this technique is used to represent the algorithm logic in easy way of representation.

In the earlier programming language, flow control was managed by either ‘goto’ statement or ‘jump’ statement. It is to be observed that these programs are not readable as well as if we draw flow-chart for the same program then it will also become more complex. Newer programming languages has adopted structured programming syntax to manage flow-control of the program such as, if – then – else, do-while etc. The programs written in this type of newer languages are more readable, their flow-charts are easy to understand and the programs are more optimized in terms of processing and memory utilization (compare to using unconditional jumps). This type programming languages which uses structured way of programming flow-control are called ***structured programming languages.*** PASCAL, MODULA, C-Language are the examples of structured programming languages.

1.6.4 Data-Structure Oriented programing

When the ICs (Integrated Circuits) are invented and used in the computer system, they become more powerful to solve the complex problem. Software engineers where now expected to develop more complicated and large software products which requires to write several tens of thousands of programming lines.

Programming languages based on control-flow would not be satisfactorily used to handle these problems and more effective languages were needed.

Soon it was discovered that developing programs needs to pay much attention to the design of the data structures of the program than to the design of the control-flow of structure of programing. Design techniques based on this principle are called ***data-structure oriented*** design technique. For example, in the programming language like C, we can design our own data type like students, customer, employee etc. using structure.

1.6.5 Data Flow-Oriented programing

When the VLSI (Very Large-Scale Integration Circuits) are implemented to the computer systems, they become more powerful and still faster than its previous generation. Some new architectural concepts, more complex and sophisticated software products were needed to solve further challenging problems. Here, software engineers were looking for more effective techniques for designing software products and very soon data flow-oriented design was proposed. This design insists that the major data items handled by a software product is identified first and then the processing required on such data items has to be identified. The processes of a software exchange data items, and its diagrammatic representation is called data flow diagram.

DFD was considered to be a generic technique, which can be used in the modeling of all different types of software. In the following example we have shown Data Flow modeling for car assembly plant.

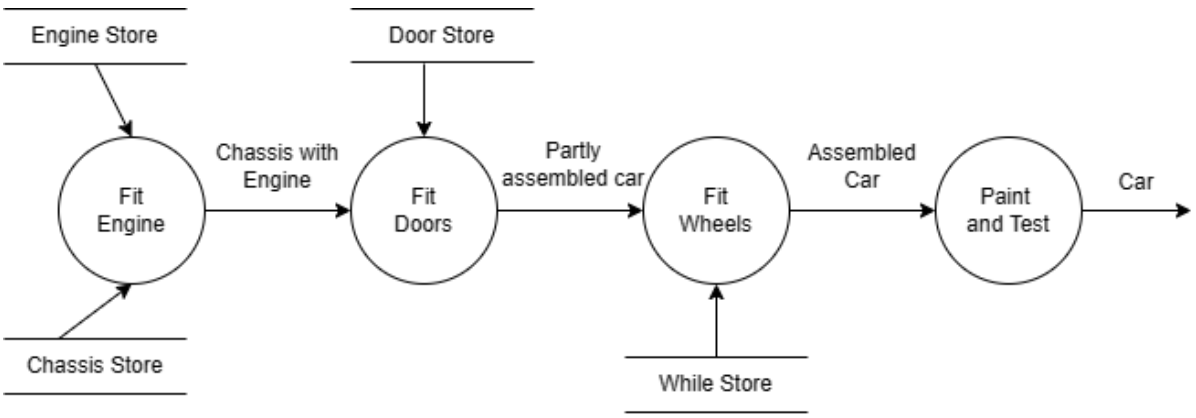


Fig: 1.3 Data flow model of a car assembly plant

In the above, diagram Engine Store, Chassis Store etc. are considered as a data store from where we take the data, whereas Fit Engine, Fit Doors etc. (which are denoted in the circle) are called processes.

1.6.6 Object-Oriented programing

In the next development, data flow design technique is evolved by Object-Oriented design. In the Object-Oriented programming design, any real word entity is represented in the form of objects, which can have some data associated with it is called properties and actions which an object can perform is represented as methods. The process of identifying properties and methods is called ***data abstraction***. Object-Oriented technique gain popularity because of its simplicity, reusability of design and code, lesser development time and ease of maintenance.

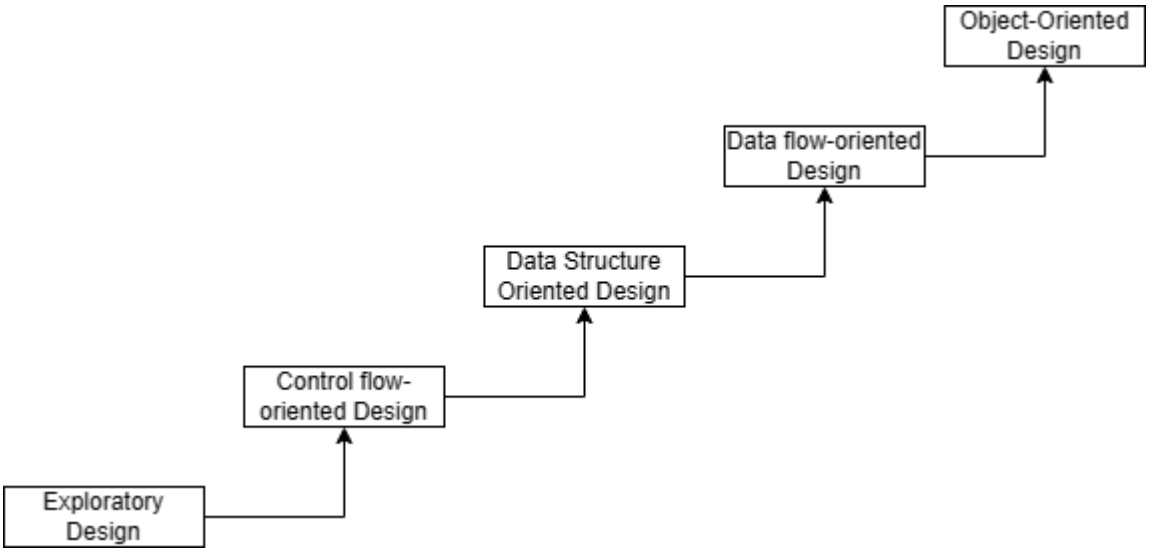


Fig: 1.4 Software design technique and its evolution

The figure given above, summarize the evolution in the software industry. Here, we have pictorially shown how the software design and programming technique methodologies have been improved from Exploratory Design to Object-Oriented design.

Check Your Progress:

1. In _____ software design technique, programmers where used their own specific style of writing the program.
2. In _____ software design technique, use of goto reduced by if-then-else and loop syntax.
3. In _____ software design technique, real word entities and their properties and methods are abstracted.

4. The problem, where the cost of the software products is decreased drastically is called _____.
5. In the data-flow diagram circles represents _____.

1.7 Let us sum up

In this chapter we have learnt how the software industry is evolved with time and new techniques of software development is adopted by the industry. In this chapter we have discussed that how we have migrated from Art to Craft and Craft to Software Engineering in the software development. We have also discussed the difference between Programs and Software products.

1.8 Check your progress: Possible Answers

Exercise: 1

1. Exploratory design
2. Control-flow oriented
3. Object-Oriented
4. Software crisis
5. Process

1.9 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Unit 2: Software Development Life Cycle 2

Unit Structure

2.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know various techniques to eliciting requirements
- Understand requirement analysis and negotiation
- Understand functional and non-functional requirements
- Learn how to document functional requirements
- Write good Software Requirement Specification (SRS)
- Represent complex logic using decision table and decision tree.

2.2 INTRODUCTION

In the previous chapter we have provided some preliminary information about the software engineering. We also discussed how the software industry has been evolved from exploratory programming style to software engineering approach. Software engineering approach emphasize to use of a life cycle model in the building of software product. In this chapter we will focus on life cycle models and the activities, which will be carried out by each phase of software development life cycle model.

A software life cycle is the series of recognizable stages that a software product passthrough during its lifetime. The first phase of the software life cycle model is a feasibility study. Similarly, the subsequent phases are: requirement analysis and specification, software design, coding, testing, system implementation and maintenance. Each of these phases is called a software development life cycle phase.

A life cycle of the software can be represented as a time taken from initiation of the software to it is being implemented. The entire development life span is divided into the variance stages, which are called phases of the SDLC (Software Development Life Cycle). In each phase of SDLC, some activities will be performed. The phase of the SDLC will be executed in the sequential manner, that means the output of one phase will be the input of the next phase.

Here, we need to distinguish between a process and a methodology. A process covers all the activities starting from initiation of the product to its delivery or retirement. Whereas, a methodology covers a single activity or at best a few individual steps in the development.

2.3 WHY TO USE LIFE CYCLE MODEL?

In a modern software development, all organizations are using life cycle model is universally accepted. But, why is it essential for the software developing organizations to follow life cycle model? The main advantage of adopting life cycle model is that it promotes, software development in a systematic manner. When a program is developed by one programmer, he has the freedom to decide the exact steps through which he will develop the program. But when the software product is developed by a team, it is essential to have precise understanding among the members of the team as to – when to do what. If the proper,

2.3.1 Importance of Documentation in the Life Cycle Model

Life cycle model provide common understanding for the project activities between the software engineer and it helps to develop the software in disciplined and systematic manner. Organizations which are developing the software products are normally prepare documentation of the life cycle model. Documented life cycle models provide misinterpretation that normally occurs in the different phases of life cycle model between different software engineers. IT helps in finding inconsistencies in the software project, data redundancies in the database design, and lapses in the development process. Proper documentation of each phase of work also provides better understanding about the project among the developers, and provide useful knowledge to the management of developing organization. It is true that, person cannot write if he is not clear about the particular idea. Proper documentation of the project helps to identify, what are actual requirements, and where exactly the necessary tailoring should be done. A properly documented life cycle model in the essential part of the modern software developing organizations. Therefore, it is important that not only development process follow well-defined process, but its documentation is also adequately important.

2.3.2 Entry and Exit Phase criteria

In the life cycle model software development process is divided into different phases. Before starting any development, the entire software project development life cycle is divided into different phase. These phases have to be identified, when a software development project is initiated.

Identification of the phases is not sufficient, but in a life cycle modes entry and exit criteria for every phase should also defined, when a project is initiated. A particular phase can begin only when the particular phase-entry criteria are fulfilled. Similarly, a particular phase is considered to be completed when the particular phase-exit condition. For example, for the phase 'System Requirement Specification' the exit criteria are, SRS (System Requirement Specification) documentation should be ready, it should be analyzed, review and approved by the customer. When the entry and exit criteria of each phase are clear, then it will become easy to observer and monitor the progress of the project.

If the entry and exit criteria of each phase are not clear or well define, then it will be difficult to know about the progress of the software project. These normally leads to the problem that is defined as 99% complete syndrome. This means every time when you observe the progress of the project, every time you will feel like project is 99% completed. Many different types of life cycle models have been developed. Before we start our discussing on different types of models, let us understand different phases of SDLC (Software Development Life Cycle).

2.4 Phases of SDLC

The software industry considers software development as a process. According to Rumbaugh and Booch, "A process defines who is doing what, when and how to reach a certain goal?" Software engineering is an arena which combines methods, process, and tools for the software development. In the software engineering approach, the concept of process is the main step. That means, a software process is nothing but a set of activities. Here all activities are performed in specific sequence in harmony with ordering constraints, the anticipated results are produced. Roughly, software development project needs two types of activities. Those are System development and project management activities. These activities together contain of a software process. As number of activities are being performed in the software development process, these activities are categorized into groups called phases. Each phase has its own well-defined activities.

The various phases which are accepted in the development of this process are together termed as Software Development Life Cycle or SDLC. The different phases of SDLC are discussed below. Normally, these phases are performed linearly or in a

circular fashion. It also can be changed according to project as well. Here, software is also considered as a product and its development of it, as a process. Thus, these phases can be termed as Software Process Technology. In general, different phases of SDLC are defined as following:

1. Feasibility Study phase
2. Requirement Gathering, Analysis and Specification phase
3. Design phase
4. Coding phase
5. Software Testing phase
6. Maintenance phase

Now, step by step we will discuss the actives has to perform in each phase of SDLC given above.

2.4.1 Feasibility Study phase

In this phase, it is to be determined that the proposed project is feasible to do or not. Obviously, we will start working on the project if it observed that the project is feasible. During the feasibility study we need to observe that the project is Economically (financially), Technically and Operationally feasible or not. We have to ensure that the cost incurred in the project should be lesser than the benefits served by the project, which means the project is ***Economically feasible***. We also have to ensure the all-technical resources needed to run the software project is available or not, which is called ***technical feasibility***. Similarly, we also have to ensure that the once the software project becomes ready and implemented, then sufficient computer literate employees are there or not, which called as ***operational feasibility***. Once the software project passed from all the feasibility tests then and then we need to start work on its development. If the software project found to be not feasible (either economical, technical or operational), that project should be abandoned in this phase.

2.4.2 Requirement Gathering, Analysis and Specification phase

The main aim of the requirement gathering activity is to collect all necessary and relevant information about the project from the customer, so that we can understand the project properly. To collect the information from customer different techniques are used like questionnaire, personal interview, onsite observation etc. If the customer is not clear about requirement, then the prototype (toy implementation of

the software which contains only design not code) is prepared and shown to the customer. This phase of the SDLC is very important because unless we cannot gather requirement properly, we should not come to know actually what to do in the project. And suppose if we are not clear with the functional requirements of the customer, we cannot make proper estimation about the time to develop the software and estimation about the cost of the software product under development.

Requirements describe the “What” of a system. The objectives which are to be attained in Software process development are the requirements. In the requirements analysis phase, the requirements should be properly defined and noted down. The outcome of this phase is SRS (Software Requirements Specification) document written in natural language. As per IEEE, requirements analysis may be defined as (1) the process of studying user’s needs to arrive at a definition of system hardware and software requirements (2) the process of studying and refining system hardware or software requirements.

The important component of the SRS document is functional, non-functional requirements and the objective of implementation. Functional requirements are those requirements which we need to serve into the software project. Usually, functional requirements have to be collected from the customer during requirement gathering techniques. Non-functions requirements are like, security, performance, load of the system etc. Generally, SRS document serves as a contact between the customer and development team.

2.4.3 Design phase

In this phase, a logical system is prepared which achieves the given requirements. Design phase of SDLC deals with transforming the customer’s requirements into a logically functional system. Normally, in the design phase following two steps has to be performed:

i) **Primary Design Phase:** In the Primary Design phase, the system is designed at block level. The blocks are created on the basis of analysis done on the functional requirement gathered from the customer. Different blocks are designed for different functions by more emphasis is put on minimizing the flow of information between blocks. Thus, all activities which needs more interaction are kept in one block.

ii) **Secondary Design Phase:** In the secondary design phase the detailed design of every block is done.

The input to the design phase is the SRS document and the output of the design phase is Software Design Document (SDD). The common tasks involved in the design phase are the following:

i) Design various blocks of the system for overall system processes.

ii) Design smaller, manageable, compact, and workable modules in each block.

iii) Design required database structures.

iv) Specify details of programs to achieve anticipated functionality.

v) Design the inputs and outputs forms of the system.

vi) Write documentation of the design.

vii) System reviews.

The Software design is the core of the software engineering process. It is also the first activity of three important technical activities like design, coding, and testing that are necessary to build software. The design should be done keeping the following points in mind.

i) It should correctly and completely describe the system.
ii) It should exactly describe the system. It should be comprehensible to the software developer.
iii) It should be done at the right level.
iv) It should be maintainable.

Make sure, while designing the system we have to take care of:

i) **Practicality** the system has to be stable and can be operated by a person of average intelligence.
ii) **Efficiency** which includes accuracy, timeliness and comprehensiveness should be there in the output of the system.
iii) **Flexibility** allows use the system should be modifiable depending upon changing needs of the customer. Such provisions should be possible with minimum changes.
iv) **Security** which is an important aspect of design and should cover areas of

hardware reliability, security of data and provision for fraud detection.

2.4.4 Coding phase

SDD document generated in the earlier (design) phase, will be considered to be input for the coding phase. In this phase, the design document is coded according to the functional requirement of the module specification. This phase converts the SDD document into a high-level language code. At present most software companies follow to some well specified and standard style of coding called coding standards.

Good coding standards improve the readability and understanding of code. Once a module is developed, a check is carried out to ensure that coding standards are followed. Coding standards normally give the guidelines about the following:

i) Name of the module

ii) Internal and External documentation of source code

iii) Modification history

iv) Uniform appearance of codes.

2.4.5 Testing phase

Testing is the process of checking the software with manually created inputs (test cases) with the intention to find errors in the software. In the process of testing, an effort is made to detect errors, to correct the errors in order to develop error free and quality-oriented software. The testing is performed by keeping the user's requirements in mind and before the software is actually run on a real system, and it is tested. Testing is the process of executing a program with the intention of finding bugs or error.

Normally, while developing the software code, the developer also performs some testing. This process is known as debugging. This extracts the defects that must be removed from the program. Testing and debugging are separate processes. Testing is intended for finding the existence of defects while debugging means locating the place of errors and correcting the errors during the process of testing. The following are some strategies for testing:

i) Test the modules carefully, cover all the possible paths of the program, and

generate enough data (test cases) to cover all the access paths arising from conditional statements.

ii) Test the modules by intentionally passing wrong data.

iii) Specifically create test cases for conditional statements. Enter data in test file whichwould satisfy the condition and again test the script.

iv) Test for locking by invoking multiple concurrent processes.

Different types of testing are there which are discussed in the Block-4 of this subject in the greater details.

2.4.6 Maintenance

Maintenance of the software products requires more effort than the effort required to develop the product. According to the studies carried out in the past, indicate that the effort required to develop the software and effort required to maintain the software is in the ration of roughly 40:60. Mainly maintenance requires to perform following three types of activity.

1. Correcting the uncovered errors, which are not detected during software development or software testing.
2. Improving the process of implementation of the system, and enhancing the functionalities according to user's requirements.
3. In the case of changing the environment, porting the software into the new environment.

Check Your Progress:

1. SDLC stands for _____.
2. In _____ phase of the SDLC, we need to gather details from the customer.
3. In _____ feasibility study, we determine that the software is feasible in the financial term.
4. Testing phase has to be performed after completion of _____ phase of SDLC.
5. SRS document is the output of _____ phase of SDLC.
6. _____phase of a SDLC required highest effort among all.
7. SDD is the output of _____ SDLC phase.

2.5 Let us sum up

In this chapter we have learnt about how the Software Development Life Cycle is divided into number of phases. Initially we need to determine that the software product is economically, technically and operationally feasible or not. Once we confirm the

product is feasible, then we try to gather functional requirements of the customer, we analyse those requirements and prepare SRS document. Once the software development organization and customer are agreed upon SRS document, Software will be design and then coding will be written. After coding, the software is tested carefully and then it is implemented at customer's end. After implementation we need to provide maintenance for the software.

2.6 Check your progress: Possible Answers

Exercise: 1

1. Software Development Life Cycle
2. Requirement Gathering, Analysis and Specification
3. Economical
4. Coding
5. Requirement Gathering, Analysis and Specification
6. Maintenance
7. Design

2.7 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Unit 3: Software Development Models

3

Unit Structure

3.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about How SDLC phases can be used in different models
- Understand Classical and Iterative models
- Understand Evolutionary and Prototyping models
- Learn Spiral model
- Understand which model can be used in which situation by comparing them

3.2 INTRODUCTION

In the Unit:2 we have learn about SDLC and how entire SDLC process is divided in the number of phases. We also have discussed about the different activities which has to be performed in each phase. In this Unit we will focus on how the SDLC can be applied on the Software Development Life Cycle.

Here you do not have to confuse yourself. See, each project is unique product and therefore each project should have different needs and requirements. For example, in some projects you will find that the customer is not clear about his functional requirement and you are facing problem in requirement gathering. In some project more activities are there in which some risk factors are involved. Some projects are time critical and you do not get sufficient development time to perform all SDLC phases for entire software in one go.

So, as per the different requirements, the way of implementation of SDLC phases gets change in each software project, and it is called software modeling. ***Software model*** is nothing but the way of implementing SDLC phases into software development project. In this unit we will discuss about different types of SDLC models.

3.3 BUILD AND FIX MODEL

This is very simple model performed in just two phases. In the first phase, the developer is developing the design and code and in the second phase if any error is there then that errors have to be fixed. Usually, this model is suitable for very small project which is developed by one developer or two developers. The following figure represents the Build-and-Fix model.

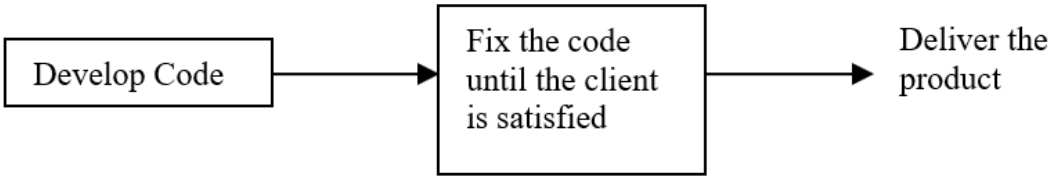


Fig: 3.1 Build and Fix model

3.4 CLASSICAL AND ITERATIVE WATEFALL MODELS

3.4.1 Classical Waterfall Model:

Classical waterfall model is the simplest, oldest and most broadly used process model. In this model, each phase of the SDLC is completed before the start of a new phase. It is actually the first engineering approach of software development. As discussed, in the classical waterfall model it is mandatory to complete the phase, before entering into the next phase of SDLC. Classical waterfall mode is depicted in the following figure:

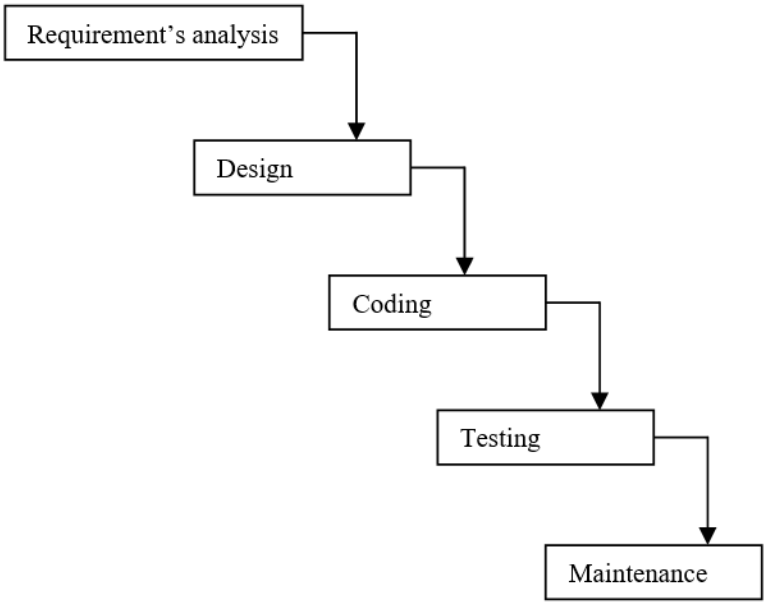


Fig: 3.2 Classical waterfall model

As depicted in the figure, once it is to be realized that the project is feasible after feasibility study, all other phases of the SDLC are performed in a sequential manner.

The waterfall model provides a systematic and sequential approach to the development of software and, it is also better than the build and fix model. But, the problem in this model is, we need complete requirements have to be available at the

time of beginning of the project, but in in reality, the requirements keep on instigating during different phases. The water fall model can includes the new requirements only in requirement gathering and analysis phase. Additionally, it does not include any kind of risk assessment. Because of in the classical waterfall model, there is not backward path is there (we cannot go towards the phases which we have completed), in reality is difficult to use practically in the development of the software. In this model, there is no methods to judge the problems of software between different phases.

3.4.2 Iterative Waterfall Model:

A slight change in the waterfall model is a model with feedback. Once software is developed, implemented and is operational, then the feedback to various phases may be given. The main difference between classical waterfall model and iterative waterfall model, is just of feedback path to the previously performed SDLC phases. Iterative model allows us to go in the backward direction for example, from coding phase to requirement gathering and analysis phase, or to design phase if it is to be observed that there is some requirement is missing or problem in the design during the coding phase. Iterative waterfall mode is mostly used in wide variety of software development project, because of it is simple in nature and more practical model compare to classical waterfall mode. Iterative waterfall mode is depicted in the following figure:

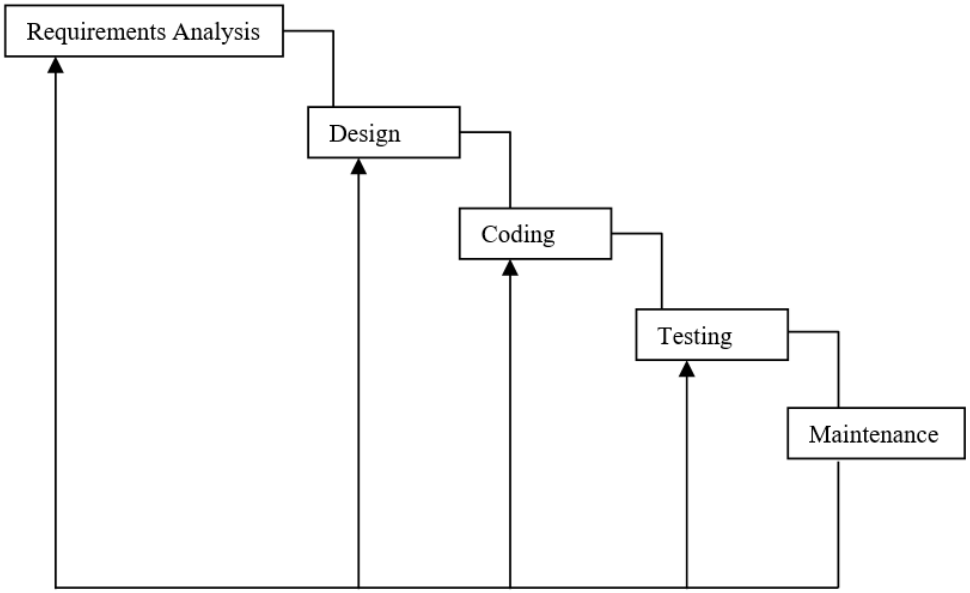


Fig: 3.3 Iterative waterfall model

3.5 EVOLUTIONARY MODEL

This model is also known as iterative enhancement model. Here, the entire software product is delivered in the different parts (deliverables). To deliver each deliverable SDLC will be followed. This model was developed to remove the limitations of waterfall model. In this model, the phases of SDLC remain the same, but the construction and delivery is done in the iterative mode. In the first iteration, a less capable product is developed and delivered for use. This product satisfies only few or limited requirements of the customer. In the next iteration, a product with incremental features is developed. Every iteration consists of all phases of the waterfall model. The complete product is divided into releases or versions and the developer delivers the product release by release.

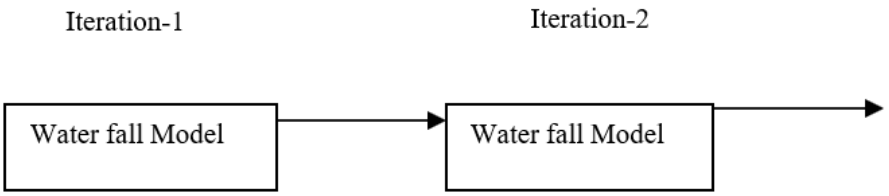


Fig: 3.4 Evolutionary model

As shown in the above figure, in each iteration one or more deliverables should be produces and implemented. In very succussing iteration product will becomes more capable and includes more user functional requirements. The following figure demonstrate how the product will be incrementally deliver to the customer.

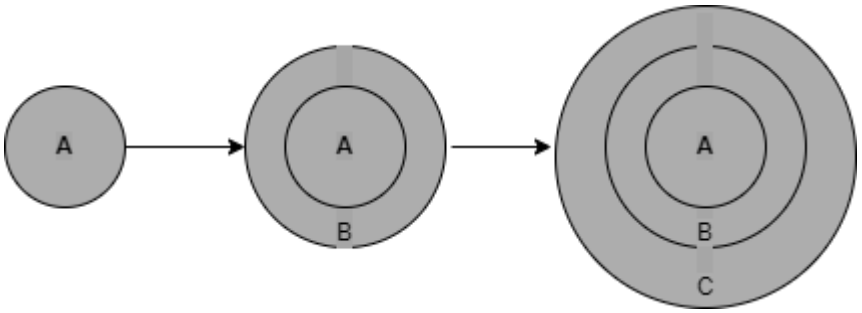


Fig: 3.5 Deliveries of functional requirements successive iteration in Evolutionary model

This model of the software development is also known as successive versions model or sometimes as the incremental model. This model is suitable when the customer needs software as early as possible and don't provide or wait till the time, in

which actual development process gets completed. In this situation, what we can do is the core functionalities (module) will be created in the first iteration and implemented at the customer site (so that the customer can start work with core functionalities). Slowly and gradually the other functionalities will be prepared in the successive iterations (versions) and it will be delivered.

3.6 PROTOTYPING MODEL

The prototyping model suggests that before starting actual development of the software, a working ***prototype*** of the system should be prepared. Prototype is nothing but a toy implementation of the system. Prototype system should have limited functional capabilities, less reliability and inefficient performance. A prototype is usually built if the customer is not clear with his functional requirements or not having sufficient knowledge about GUI (Graphical User Interface). The concepts of the prototyping are used to gather all functional requirements from the customer.

Developing a working prototype of the software in the first phase overcomes the disadvantage of the waterfall model where the reporting about serious errors is possible, only after completion of software development. The working prototype is given to the customer for operation. After using it, customer gives the feedback. Analyzing the feedback given by the customer, the software developer refines the requirements, adds the requirements and prepares the final SRS document. Once the prototype becomes finalized and operational, the actual product is developed using the normal waterfall model. The following represents the features prototyping model:

(i) It helps in understanding and determining user requirements more deeply.

(ii) At the time of actual product development, the customer feedback is available.

(iii) It also considers any types of risks in the initial level.

The figure given below describe the, working of protype model in more detail. First, we need to build the prototype of the software and we need to present it to the customer. We need to refine the customer's requirement and again need to work on the prototype. The process needs to be executed till the prototype is accepted by the customer and we understand all functional requirements deeply. Once the prototype is accepted by the customer then and then other SDLC phases will be performed.

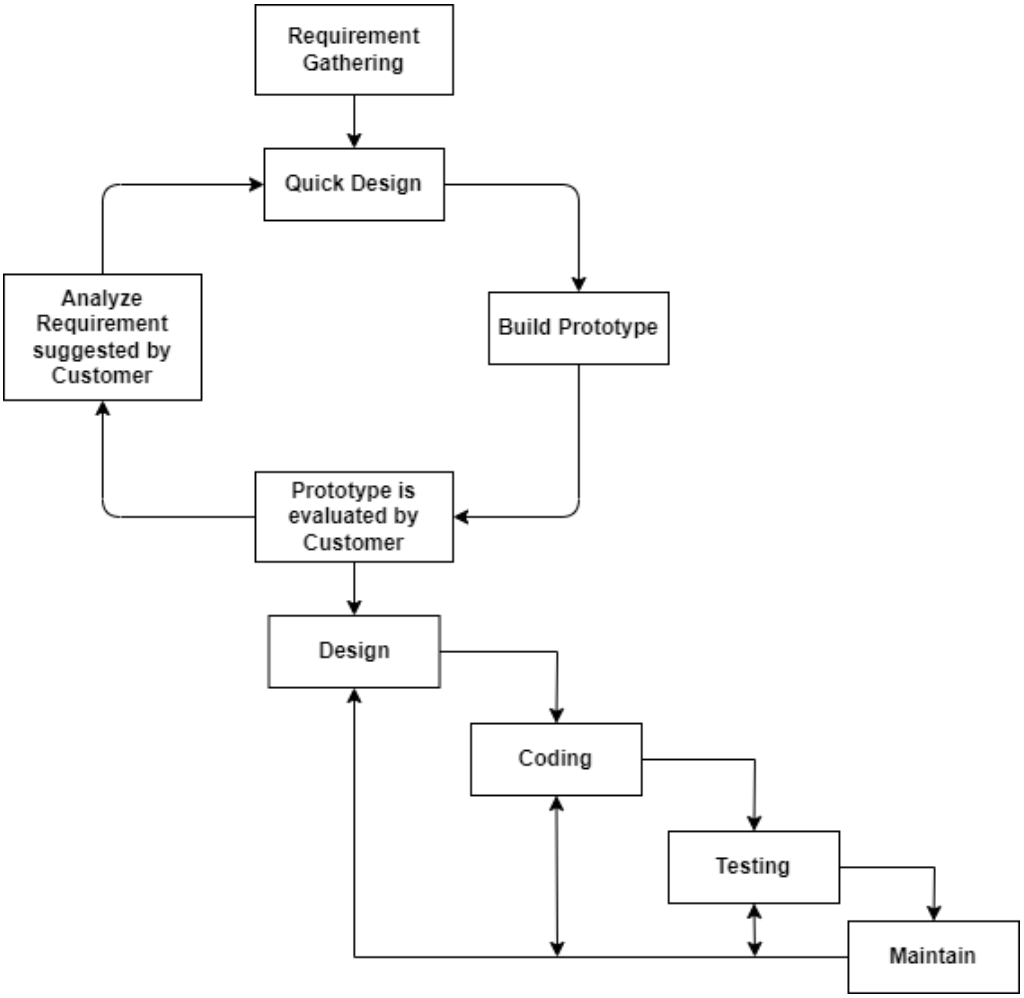


Fig: 3.6 Working of Prototype model

3.7 SPIRAL MODEL

This model can be considered as the model, which combines the strengths of various other models. Orthodox software development processes don't take risk or uncertainties into account. Many important software projects have failed because of unforeseen risks.

The other models we have seen so far, view the software process as a linear activity whereas spiral model considers it as a spiral process. This is made by representing the iterative development cycle as an expanding spiral.

The activities discussed below are to be considered as primary activities in the spiral model:

- **Finalizing Objective:** For each phase objectives are to be set or finalized.
- **Risk Analysis:** The risks are identified for each phase or activity to the extent possible.

Risks are identified, analyzed and necessary action is to be taken to avoid or resolve risk.

- **Development:** Based on the risks that are identified, proper SDLC model is chosen and is followed.
- **Planning:** During planning, the work done till this time is reviewed. Based on the review, a decision regarding whether to go through the loop of spiral again or notwill be decided. If there is need to go, then planning is done accordingly.

The phases discussed above are followed iteratively in spiral manner, in the spiral model. The following depicts the Boehm's Spiral Model (IEEE, 1988).

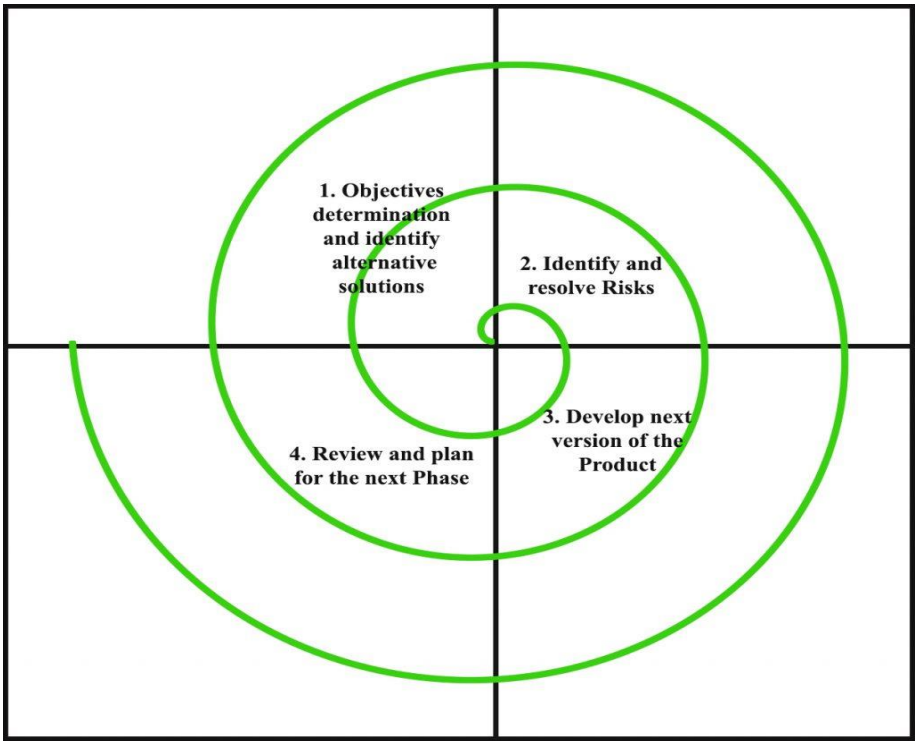


Fig: 3.7 Spiral model

3.8 COMPARISION OF DIFFERENT SDLC MODELS

The classical model which we have discussed first is to be considered as the basic model and all other SDLC models are its superfluities of this model. In fact, it is very difficult to use the classical waterfall model to use, as this model do not support mechanism to handle errors (there no feedback path towards previous phase). This problem is overcome by iterative waterfall model. The iterative waterfall model is probably most widely used SDLC model as it is simple to understand and easy to implement. Iterative model can be implemented on those software development

projects, where problem areas are clearly defined or well understood. Iterative model doesn't focus on risk assessment, is one of the down side of it.

The prototyping model is suitable to use in the area where the technical requirements are not clear or well understood. This model is very popular to resolve issues related to user interface.

For the large size of problem, usually evolutionary model is suggested, which actually decompose a large problem into small, manageable sub components. It actually reduces the complexity. This model is more preferable when the development is done using object-oriented approach.

Finally, we have discussed spiral model, which is considered as a meta model. It includes the features of all other SDLC models. The spiral model is good candidate for those projects in which certain amount of risk is involved. The spiral model is more suitable for development of technically challenging projects that are prone to certain kinds of risks.

Check Your Progress:

1. _____ model is the basic model for software development.
2. In _____ model the problem related to feedback path of classical waterfall model is resolved.
3. For large and complex software development project, _____ SDLC model is suitable.
4. _____ model of SDLC consider risk in account.
5. If the customer is not clear with functional requirements, then _____ SDLC model is advised.
6. _____ SDLC model is called meta model, includes features of all other models.
7. For the object-oriented product development, _____ model is suggested.
8. _____ SDLC model is ideal to understand requirements related to user interface.

3.9 Let us sum up

In this chapter we have learnt how the Software Life Cycle phase can be implemented in the form of software models, pertaining to the type of software project. We have discussed basic model that is classical waterfall model, Iterative waterfall model which has feedback problem of classical waterfall model has been resolved. We have also discussed how the evolutionary model helps to manage large and complex project, how interface related problem or suppose user is not clear with

technical functional requirement then prototype model helps. Finally, we have ended our discussion with spiral model which has cover the benefits of all other model (meta model), and how it helps in risk assessment and relevance.

3.10 Check your progress: Possible Answers

Exercise: 1

1. Classical waterfall
2. Iterative waterfall
3. Evolutionary model
4. Spiral
5. Prototyping model
6. Spiral
7. Evolutionary
8. Prototyping model

3.11 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Software Engineering

BLOCK 2: SOFTWARE PROJECT MANAGEMENT

BLOCK 2: SOFTWARE PROJECT MANAGEMENT

Block Introduction

In this block-2 of the Software Engineering, we have tried to emphasis on: How as a project manager a large size and complex project can be managed. Very important thing which a project manger has to do after Requirement gathering and analysis phase is Estimation. After analyzing all functional requirement from the customer, project manager need to estimate certain project parameter such as size of the project, Effort estimation, estimation about time duration required to complete the project and finally cost of the project. In this block we have discussed several formulations given by some researchers, to do estimation.

After estimation, the entire project has to broken down into number of task and number of tasks has to broken down into number of activities. In this block we have discussed about Work Breakdown Structure (WBS) and on the basis of it, how can we plan the project activity is discussed in this block in details.

Block Objective

Objective of this block is to explain how the large project is divided into number of tasks and further in the activities, how to estimate time duration and how to manage the project so that project can be completed in designated time duration and estimated cost.

After learning this block, reader of this block will be able to perform WBS for the project, Estimate various project parameters such as size, effort, cost, and time. Learner of this block will also learn organizational structure and staffing in the organization. In short, reader will be able to learn how software project is management by a developing organization as per IT industry standards.

Block Structure

BLOCK 2: SOFTWARE PROJECT MANAGEMENT

UNIT 4 SOFTWARE PROJECT MANAGEMENT – I

Objectives, Role of Software Manager, Planning of the project, Project size estimation metric, Software project size estimation techniques, Let Us Sum Up

UNIT 5 SOFTWARE PROJECT MANAGEMENT – II

Objectives, Estimation of staff, Scheduling, Structure of organization, staffing, Let Us Sum Up

UNIT 6 REQUIREMENT ENGINEERING PROCESS

Objectives, Requirement engineering process, Requirement elicitation, Requirement analysis and negotiation, Requirement specification, System modeling, Validation requirement, Requirements management, Let Us Sum Up

Unit 4: Software Project Management-1

4

Unit Structure

4.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about important parameters of the software project
- Understand about the estimation of project size
- Understand Empirical, Heuristic and Analytical estimation technique
- Learn Expert judgement and Delphi cost estimation technique
- Understand COCOMO model

4.2 INTRODUCTION

In the previous unit:3, we have learnt about different software engineering models in details. In this unit, we will focus on software project management. In the software project management, mainly we need to focus on two things:[1] Estimating various attributes like, duration to develop a particular software, Efforts required to develop the software and finally estimated cost of the software product and [2] Planning and Managing various activities such as making arrangement of staff, monitoring project activities etc.

Therefore, we have divided “Software Project Management” into two parts. In this unit we will focus only on estimating techniques and in the next unit we will focus on other software project management activities.

As being a software project manager, after gathering all functional requirements from the customer, you need analyze all functional requirements and you need to compute or estimate how much time is required to develop this software, you also have to estimate the efforts required to build this software. Once you have estimated the effort required to develop the software then you need to estimate cost which incurred to develop the software and staff required to build the project. After learning this unit, you will be able to learn several formulations, which will help you out to estimate these project attributes (duration, effort and cost).

4.3 ROLE OF SOFTWARE PROJECT MANAGER

Here we will discuss about the Job responsibilities of the project manager, and the essential skills which are required in the management of the project.

4.3.1 Skill requires in Software Project Management:

To manage the software project, a project manager should have enough theoretical knowledge of different types of project management techniques. A good project manager should have following skills:

- To manage the project decision making capabilities and good qualitative judgements are essential skills.
- A project manager should have good grasping of latest software project management like time duration estimation, effort and cost estimation etc.
- A project manage should also have good communication skill.
- A project manager should have enough controlling and leadership skills to handle team of employees.
- A project manager should also have skill to monitor or track the progress of the software project.
- A project manager should also have skills like, sound knowledge about the project; making, managing, controlling and leading the project team; interactions with customer; and resource utilization.

4.3.2 Job responsibilities of a Software Project Manager:

The main responsibility of a project manager is to lead a project successfully. This is surely a very hazy job description. A project manager has to gather all functional requirements from the customer and has to determine that the software project is worth in terms of economical, technical and operational feasible. A project manager needs to estimate size of the software, and from the size other attributes like effort, duration and cost of the software.

A project manager needs to divide the entire project development process into various phase and phase will be further dived into activities. A project manager has to schedule all project activities and also has to allot staff and required resources to each development activities. A project manager has to create team for design, coding, testing etc. and has to assign duties to these teams.

During the project development, a project manager has to monitor and control all development activities, so that each activity should perform in time as planned. During the development, a project manager has to produced different types of

documentations like SRS (Software Requirement Specifications), SPMP (Software Project Management Plan) and different types of test reports.

4.4 PLANNING OF THE PROJECT

After doing feasibility study, when the software project is to be found feasible, then project planning activity will be started and it has to completed before any development activity is started. Project planning consists of following activities:

1. Project manager has to estimate various attributes of the project like:
    - Size: What will be the size of the project in terms of number of lines of code or number of input and output functionalities?
    - Duration: What will be the duration within that the entire software will be developed?
    - Effort: How much effort is required in the development of the software?
    - Cost: How much cost will it take to develop the entire software?
2. Arrangement of human and other resources
3. Staffing and assigning responsibilities to them
4. Identification risks and making strategies to avoid it.
5. Making other plans like Testing of software, configuration management and Quality assurance plan etc.

In the following, figure it is demonstrated that project manager needs to do proper estimation about the size as the other attributes like duration and effort is based on size. If the project is large, obviously that takes more effort and more time duration to complete. Once the effort and duration are estimated then based on it, project manager can estimate staff required to develop the software.

If the project staffing is known then project manager can make a schedule for each activity and phase of the project. Similarly, if the effort is computed and known then, project manager can estimate about the development cost of the project.

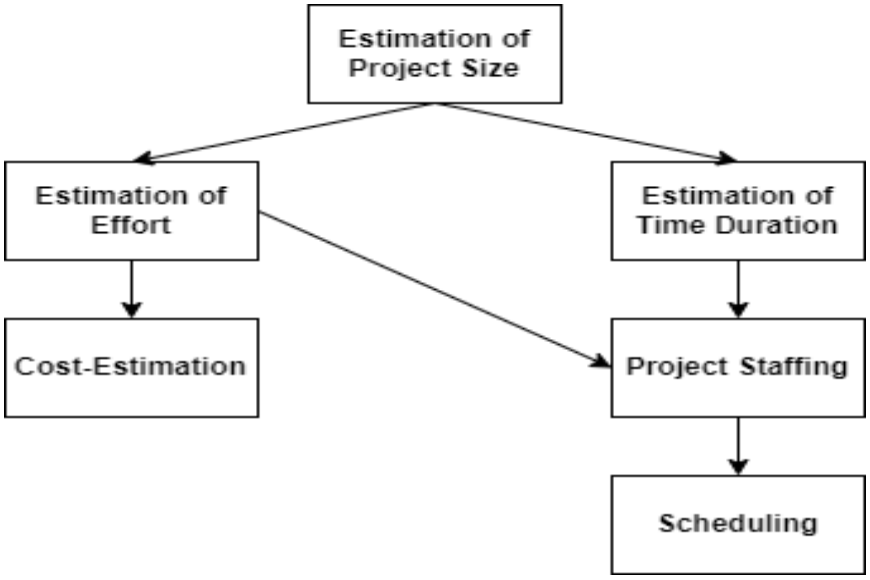


Fig: 4.1 Sequence of project estimating activities

4.5 PROJECT SIZE ESTIMATION METRICS

It important to estimate accurate problem size, if we want to estimate effort, time duration and cost of the software product. In order to estimate problem size, we need to define some metrics (units) so that we can express the problem size. But, before learning different metrics, we try understand that what is the term “problem size” means. The problem size does not mean number of bytes which is acquired by the source code of the software product. The problem size is a measure of the complexity of the program in terms of time and effort required to develop the software product.

Usually, there are two metrics are there to estimate the size of the software: [1] Line of Code (LOC) and [2] Functions Point (FP).

4.5.1 Line of Code (LOC):

Line of Code or LOC is the simplest metric among all available metrics to estimate project size. LOC is very popular and easy to use. In this metric, the size of the project is estimated by counting the number of lines of source code, which is written by the software developer. Obviously, in counting of LOC, comments written by the programmer as well as header files included is not counted (ignored).

When the development process is completed then determining line of code is very simple job, but estimating LOC in the beginning of the project is more difficult. To

estimate LOC in the beginning of the project, entire project is divided into various modules, modules are further divided into various sub-modules and sub-modules are in the classes or functions. Then for each low-level function, the number of lines is estimated and finally LOC for sub-module, module and finally for entire project is estimated. Even though, LOC metric is simple and easy to use, there are several limitations are there:

- LOC gives numeric value to the particular problem size, which can vary programmer to programmer because different programmer writes the source code in their own way. It is possible that one programmer writes several instructions where other programmer write a single instruction of the same task.
- LOC consider the effort of writing the code, but not the complexity. Obviously, writing complex program requires more effort than the simple program. LOC matric consider only number of lines but not give reward for writing complex program.
- LOC metric does not consider quality, efficiency and performance of the code into account. It is true that better quality and higher efficiency code should be rewarded but it doesn't happen in the case of LOC.
- LOC penalize the use of high-level programming language and reusability of code (as in both number of lines of code will reduce).
- Estimating accurate LOC is possible only after the software product is fully developed, therefore LOC metric has limited use for the manager of the project in project planning.

4.5.2 Function Point metric (FP):

Albrecht in 1983 has proposed Function Point metric. FP metric overcomes many shortcomings of the Line of Code metric, due to this reason FP metric has slowly gain popularity. The main advantage of using FP metric is, it is easy to estimate the size of the product directly from its specifications. The basic idea of the Function Point metric is the size of the software product is directly depending on the number of functions and features that it supports. If a software product has large number of supporting features, should have large size. Function Point metric considers number of input and output data values to the software as it gives some indications of the number of functions supported by the software. Along with the number of input and

output data values, the size of the software product is also depending on the number of files and number of interfaces.

The size of the product in Function Points (FPs) can be represented as the weighted sum of five properties of the problem. FP is computed in two steps. In the first step Unadjusted Function Point (UFP) is computed using the following formula:

UFP = (Number of External Inputs) *4 + (Number of External Outputs) *5 + (Number of External Inquiries) *4 + (Number of Internal logical Files) *10 + (Number of External Interfaces) *10

- **External inputs:** A process by which data crosses the borderline of the system to take input into the system. Data may be used to update one or more logical files. By mean of data here, we have to understand either business or control information.
- **External outputs:** A process by which data crosses the borderline of the system to give data to external file or device. It can be a user report or a system log report.
- **External user inquires:** A count of the process in which both input and output results in data retrieval from the system. These are basically system inquiry processes.
- **Internal logical files:** A group of logically related data files that resides entirely within the boundary of the application software and is maintained through external input as described above.
- **External interface files:** A group of logically related data files (it can be tables of the database system) that are used by the system for reference purposes only. These data files remain outside completely from the borderline and are maintained by external applications.

Once the unadjusted function point (UFP) is calculated, the next step is to calculate Technical Complexity Factor (TCF). The TCF can be calculated by considering 14 other factors like high transaction rates, response time requirements, throughput etc. Each of these factors are assigned a numerical value from 0 (no influence or not present) to 6 (highly influence). We have to sum of numerical values (from 0 to 6) assigned to these 14 factors, which known as degree of influence (DI). DI can be varied

from 0 to 70. Now, TCF can be (varied from 0.65 to 1.35) calculated using following formula:

$$TCF = 0.65 + (0.01 * DI)$$

Once the Technical Complexity Factor (TCF) and Unadjusted Function Point (UFP) is calculated then finally Function Points (FPs) can be calculated using following formula.

$$FP = UFP * TCF$$

Benefits of Using Function Points

- Function points can be used to estimate the size of a software application correctlyirrespective of technology, language and development methodology.
- User understands the basis on which the size of software is calculated as these arederived directly from user required functionalities.
- Function points can be used to track and monitor projects.
- Function points can be calculated at various stages of software developmentprocess and can be compared.

4.5.3 Feature Point metric:

The limitation of the function point (FP) metric is that it does not consider the complexity of an algorithm. That means that FP metric assumes that the effort required to develop two functionalities of the system is same. But in reality, we know that the in the development process efforts required to solve more complex function is higher than the effort required to develop a function which is either easy or less complex.

To solve this problem of Function Point metric, an extension of it, ***Feature Point metric*** is introduced. In Feature Point metric one more parameter is added, which nothing but the algorithm complexity. This parameter ensures that the function points of the more complex program will be higher than that of less complex program.

4.5.4 Other types metric:

Other types of metrics used for various purposes are quality metrics which include the following:

- **Defect metrics:** These metrics measure the number of defects in a software product.

This may include the number of changes required in the design, number of errors which are detected during testing, etc.

- **Reliability metrics:** These metrics measure mean time to failure of the software product. This can be done by gathering data over a period of time.

4.6 SOFTWARE PROJECT SIZE ESTIMATION TECHNIQUES

The estimation of various parameters of the project like size, effort, duration, cost etc. is a part of basic planning activity. These estimation helps to quote to the customer along with that it also helps to the manager to allocate resources and preparing project scheduling. Software project size estimation techniques are broadly classified into three categories:

[1] Empirical estimation

[2] Heuristic estimation

[3] Analytical estimation

4.6.1 Empirical estimation

In an empirical estimation technique, an educated guess is made about the important project parameters. In this technique, prior development experience of similar type of software is used to make estimation. Empirical estimation is done by experience and common sense, but it also has many different activities to be performed which are formalized over the years. These activities are:

(i) Expert Judgement Estimation Technique.

(ii) Delphi Cost Estimation Technique.

[1] Expert Judgement Estimation

Expert Judgement is one of the most broadly used technique for software project estimation. In this technique, an expert guess about the size of the problem based on knowledge and past experiences of working on similar type of project. In this technique, an expert divides the project into multiple components, then for each components size and cost estimations are made and finally the size and cost estimation are made for overall software product.

Expert judgement technique is subject to individual bias and human errors. Also, it is possible that the expert may overlook some of the factor of the software product unintentionally. It is also possible that the expert makes estimation of the software product, but expert may not have experience or knowledge about all features of the software project.

[2] Delphi Cost Estimation Technique:

In the expert judgement technique server problems need to face, which result in wrong estimation, because of only one person (expert) is making the estimation. This problem is rectified in the Delphi cost estimation, where the software estimation is done by group of experts and a coordinator.

The coordinator prepares and distribute the functional requirements of the project to various experts in the team. All expert gets some time duration, and based upon information supplied, each expert makes an estimate and then all the results are compared. If the estimates are reasonably close, they can be averaged and used as an estimate. Otherwise, the estimates are distributed back to the experts, who discuss the differences and then make another estimate.

The main advantage in Delphi cost estimation, group of experts are making their estimation which may be more reliable an accurate than a single expert's estimation.

4.6.2 Heuristic Estimation Techniques

Rather than doing guess work about the estimation as we have seen in the empirical estimation technique, a Heuristic technique relate the different project parameters using suitable mathematical expressions. Once the independent (basic) parameters are known, the dependent (other) parameters can be computed by using some mathematical expression. In short, a Heuristic Estimation Technique use mathematical modeling to compute other related parameters of the software project on the basis of known parameters.

[1] COCOMO – HEUTISTIC ESTIMATION TECHNIQUE

COCOMO which stand for Constructive Cost estimation Model was proposed by Boehm in 1981. As per Boehm, the process of the development can be categorized

as: Organic, Semidetached, and Embedded. Usually, data processing programs are considered as Application programs. Interpreters, Linkers and Compilers are considered as utility programs and Operating system like software are considered as system programs.

In 1975 Brooks has stated that the complexity of Application programs, utilities and system programs are 1:3:9. In 1981 Boehm has defined Organic, Semidetached and embedded systems are as follows:

- **Organic:** Small size project. A simple software project where the development team has good experience of the application
- **Semi-detached:** An intermediate size project and project is based on rigid and semi-rigid requirements.
- **Embedded:** The project developed under hardware, software and operational constraints. Examples are embedded software, flight control software.

As per Boehm, software cost estimation should be done in three stages. [A] Basic COCOMO [B] Intermediate COCOMO [C] Complete COCOMO

[A] *Basic COCOMO Model* :

Basic COCOMO model provides an approximate estimate of the project parameters. As per COCOMO model Effort and Time to develop software can be computed using following formula.

$$\text{Effort} = a_1 * (\text{KLOC})^{a2} \quad \text{PM}$$

$$\text{Tdev} = (\text{Effort})^{b2} \quad \text{Months}$$

Here, KLOC means Kilo Line of Code. 1000 lines of the source code will be considered as 1 KLOC, a1, a2, b1, b2 are constants and its values are depending upon the type of software that is Organic, Semidetached, or Embedded. The unit of measuring effort is PM (person month) and unit for measuring (Tdev) Time to develop software product is months.

As per COCOMO model Effort can be computed for three categories of software product are as given below:

Organic : $\text{Effort} = 2.4 * (\text{KLOC})^{1.05}$ PM

Semidetached : $\text{Effort} = 3.0 * (\text{KLOC})^{1.12}$ PM

Embedded : Effort = 3.6 * $(KLOC)^{1.20}$ PM

And Tdev (Time to develop software) can be computed using COCOMO model is:

Organic : Tdev = 2.5 * $(Effort)^{0.38}$ Months

Semidetached : Tdev = 2.5 * $(Effort)^{0.35}$ Months

Embedded : Tdev = 2.5 * $(Effort)^{0.32}$ Months

[B] *Intermediate COCOMO Model* :

This model computes development cost and effort as a function of program size (LOC) and a set of cost drivers.

[C] *Complete COCOMO Model* :

This model computes development effort and cost which incorporates all properties of intermediate level with assessment of cost implication on each step of development (analysis, design, testing etc.).

4.6.3 Analytical estimation

Analytical estimations are made on the basis of certain assumptions regarding the project. Compare to Empirical and Heuristic techniques Analytical estimation techniques use scientific basis. Halstead's Software Science metric is an analytical estimation method (which is outside of the scope of the book).

Check Your Progress:

1. LOC stands for _____.
2. _____ metric is enhanced version of the function point metric.
3. FP = _____ * TCF.
4. In _____ estimation technique, we use guess work about the parameters of the software.
5. COCOMO stands for _____.
6. In _____ estimation of Empirical estimation technique, estimation is done by group of experts about the important software parameters.
7. In COCOMO model, if the problem is well-understood and experience staff members then, it is considered as _____ type of project.
8. Unit of measure for Effort is _____.

4.8 Let us sum up

In this chapter we have learnt how the estimation about important project parameter can be made. We have discussed the size of the project can be measured by counting Line Of Code (LOC) or by Function Point (FP). Based on the project size we need to compute Effort and Time to develop a software. To compute these different approaches can be used like Empirical Technique, Heuristic or Analytical estimation technique. In Empirical approach Effort and Time to develop software is guessed by an expert (Expert Judgement Technique) or Group of experts (Delphi Cost estimation Technique). In Heuristic estimation technique we use COCOMO model to estimate parameter by using some formula.

4.9 Check your progress: Possible Answers

Exercise: 1

1. Line of Code
2. Feature Point
3. UFP
4. Empirical
5. COnstructive COst MOdel
6. Delphi cost
7. Organic
8. PM (Person Month)

4.10 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Unit 5: Software Project Management-2

5

Unit Structure

5.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about how to estimate staff for the software development project
- Understand preparing schedule for project
- Understand CPM, Gantt and PERT charting techniques
- Learn Organization and Team structures
- Understand staffing and choosing good software engineer in the team

5.2 INTRODUCTION

In the previous unit:4, we have learnt about how can we estimate different important project parameters like project size, efforts required to develop the project, time required to develop the project and so on. In this chapter we will learn about, what the project manager should do after estimating all these parameters.

Obviously, once the size, effort and time to develop is estimated then a project manager has to estimate staffing level. Initially we will start our discussion with how a project manager can estimate the human resources required for the project and then we will discuss how to prepare project schedule. During the discussion, we will learn about Work Break down Structure (WBS), we will also learn how can we make different activity diagrams like Critical Path Method (CPM), PERT and Gantt charts.

5.3 ESTIMATION OF STAFF

After determining effort requires to develop the software, it is important to estimate the staffing requirement for the project. Putnam has studied this problem first and what should be pattern for the staffing of the project is formulated. He has extended the work done by Norden who has formulated staffing pattern for his Research and Development (R&D) department. Let us first understand the work done by Norden to solve staffing requirement estimation problem.

5.3.1 Norden's estimation for staffing:

Norden has studied patterns of staff for several R&D projects. He found that the pattern of the staff can be approximated by Rayleigh distribution curve. Norden has represented the formula for Rayleigh curve is as follows:

$$E = \frac{K}{{t_d}^2} \times t \times e^{\frac{-t^2}{2{t_d}^2}}$$

In the above equation, E is the effort required at time t. E represent the number of engineers or staff required at any particular time of the project. K is the area covered by the curve, and t_d is the time at which the curve reaches its maximum value.

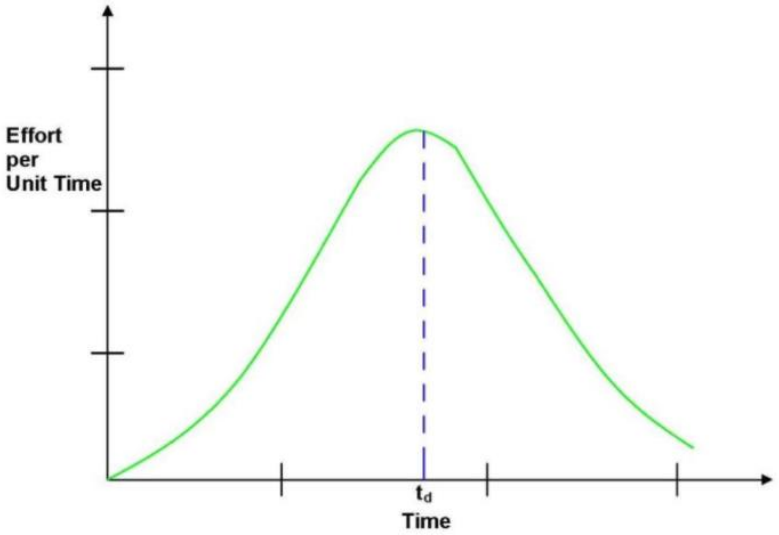


Fig: 5.1 Rayleigh curve

5.3.2 Putnam's Work:

Putnam considered the staffing of the software projects and found that staffing problem for the software development projects has similar types of characteristics, similar to the Norden's R&D staffing work. Putnam also has used Norden-Rayleigh to relate Line of Code required to develop a software with effort. Putnam has derived the following expression:

$$L = C_k \times K^{\frac{1}{3}} \times {t_d}^{\frac{4}{3}}$$

In the above equation, K is the total effort (in PM) expanded and L is the software product size in KLOC, t_d represents time of system in integration and testing, C_k is the

constant represent the development environment. If the value of C_k=2 then poor development environment and C_k=11 then it represents excellent development environment.

5.4 SCHEDULING

Scheduling is the main project tasks and very important in project planning activity. It contains deciding which tasks would be performed when. To prepare the schedule for the software project, a manager needs to perform the following activities:

1. Recognize all the tasks needed to complete the project.
2. All tasks of the software project need to dived into number of small activities.
3. Identify the dependency among different activities.
4. Identification risks and making strategies to avoid it.
5. Establish the most likely estimates for the duration of time required to complete each activity.
6. Assign required resources to all activities.
7. Decide starting and completion dates for all activities.
8. Determine the critical path. A critical path is the sequence of activities that determine the duration of the project.

In the first step of the scheduling a project is divided into number of tasks which are essential to complete the entire project. A deep knowledge of the particulars of the software project and development process, helps manager to recognize important tasks of the project. In the next step project manager has to take each task of the project one by one and has to split into number of activities, which are essential to complete that particular task.

Dividing entire project into different task and each task into number of activities is called Work Break down Structure (WBS). Once each task of the project is divided into number of activities, then a project manager has to identify which activities are dependent (dependent activities has to be performed sequentially that means after completion of first activity, another activity can be started) or independent activities (activities which can be performed parallelly).

Project manager now has to determine the time duration to complete each activity and based on that project manager has to choose the start date and end date for each activity. The start date and end date should be assigned based on the activities are dependent or independent. Project manager has to allocate resources that can be human resource, hardware resource or any other required resource to each activity.

Finally, project manager prepared Critical Path Method (CPM) or Project Evaluation and Review Technic (PERT) charts and find critical path which gives estimation of the time duration in which project development will be completed. Project manager will also prepare a Gantt chart, which represent schedule (from which date to which date particular activity has to perform).

This entire process is called project scheduling. Once the project is scheduled, all PERT, CPM and Gantt charts has been prepared then project manager has to manage and monitor the project activity as per the schedule is prepared.

5.4.1 Work Breakdown Structure (WBS):

Work Breakdown Structure is used to decompose the entire project into number of tasks and then each task into smaller and manageable activities. This actually makes a hierarchical (tree like) structure. The root node of this tree, is labeled as the name of the project, all the child nodes are the different tasks which are essential to perform to complete the project successfully. All these nodes should be labelled with the name of the task like Requirement Specification, Design, Code, Test etc. If required then each task should also be broken down into number of activities which will becomes child nodes under that task for example design task should further decomposed into activities like database design and Graphical User Interface (GUI) design. The Work Breakdown structure of a Library System is given in the figure 5.2 below.

Work breakdown allows manager to divide complex and larger tasks into smaller subtasks so that manager can take some hard decision. Once the tasks are broken into number of smaller activities then project manager can distribute these activities to a large number of engineers.

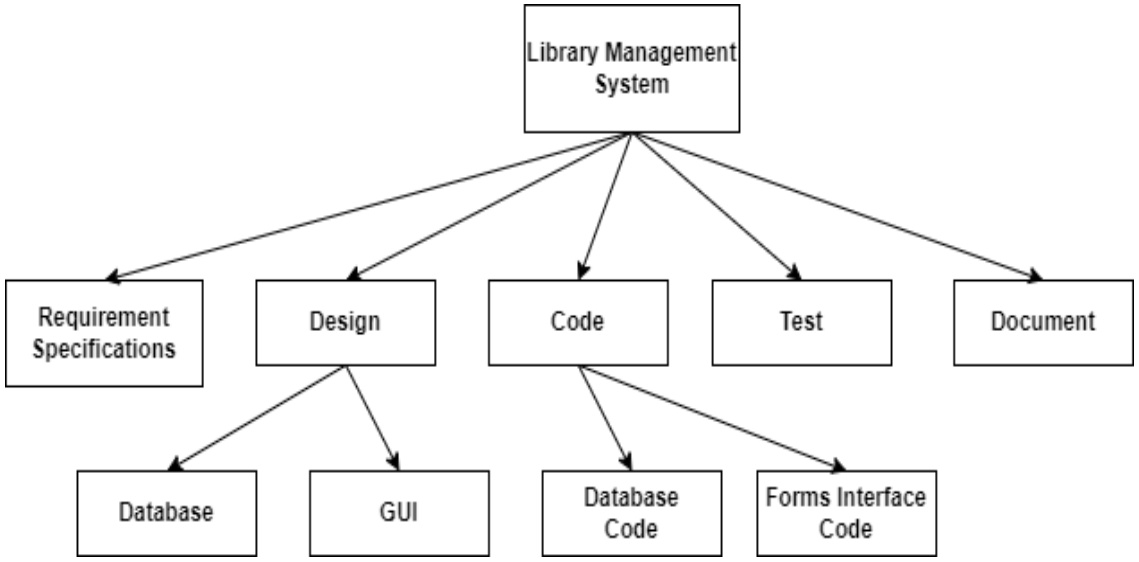


Fig: 5.2 Work Breakdown Structure for Library Management System

5.4.2 Critical Path Method (CPM):

WBS converts the entire project into activities, which can be represented in the activity network, depending upon on which time, which activity is scheduled by manager. Managers can easily estimate the time duration for different activities and tasks different ways as discussed in the previous unit. To represent activity network diagram a method called Critical Path Method (CPM) is used. Consider the following example in which, we have described to activity network diagram for a Library Management System.

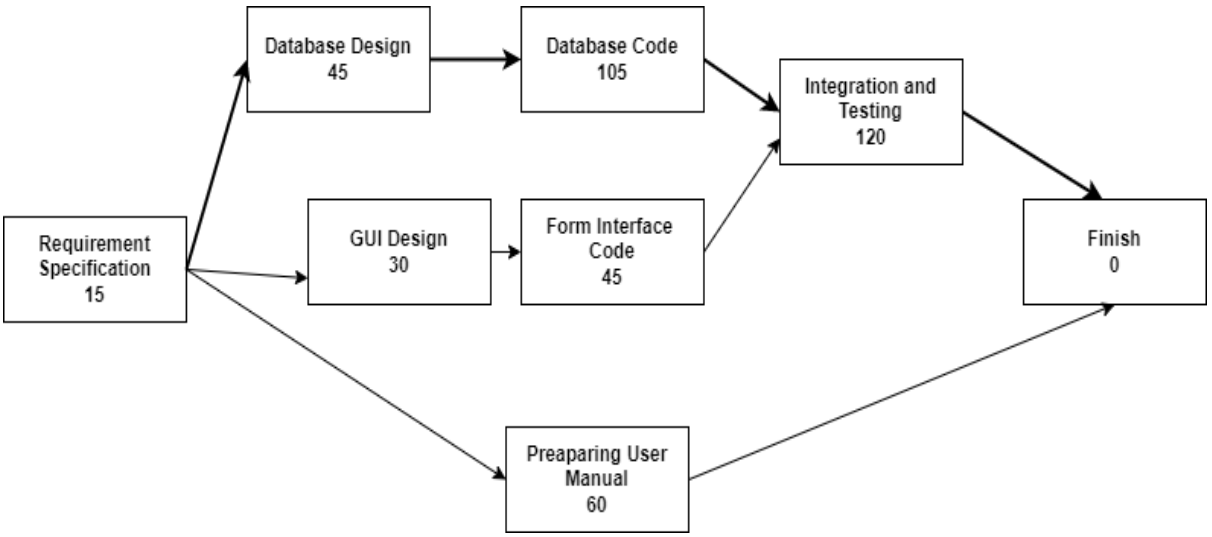


Fig: 5.3 Activity network diagram for Library Management System

After preparing activity diagram and estimating time duration for each activity, a project manager needs to compute critical path. Critical path is the longest path of activity diagram, and that represent the time duration to complete the entire project.

Following analysis can be made from the activity network diagram:

1. Minimum Time (MT): It is a time to complete the project, which the maximum of all paths (critical path) from start to finish.
2. Earliest Start time (ES): It is a time of a task is the maximum of all paths from the start to this task.
3. Latest Start time (LS): It is a difference between MT and the maximum of all paths from this task to finish.
4. Earliest Finish time (EF): It is time when the particular activity will finish. It is sum of ES and time duration of that particular activity.
5. Latest Finish time (LF): It a time of a task which is obtained by subtracting maximum of all paths from this task to finish from MT.
6. Slack Time: It is a time duration which represent a delay. It can be simply computed by LS – EF.
7. Critical tasks: A task having slack time 0 is called critical tasks.

The computation of above parameters from the activity network diagram is shown in the following table:

Task	**ES**	**EF**	**LS**	**LF**	**ST**
Requirement Specification	0	15	0	15	0
Database Design	15	60	15	60	0
GUI Design	15	45	90	120	75
Database Code	60	165	60	165	0
Form Interface Code	45	90	120	165	75
Integration and Testing	165	285	165	285	0
Preparing use manual	15	75	225	285	210

The critical paths are those paths whose duration is equal to MT. The critical path in denoted with darken arrow lines in the Figure 5.3.

5.4.3 Gantt Chart:

The main purpose of Gantt chart is plan resource allocation to the different activities. The resources which need to assigned to different activities by the project manager can be human resource, hardware or software. Gantt chart is developed by Henry Gantt. In the Gantt chart activities are represented as bars with respect to time duration. Each bar has two parts, a shaded part and white part. A shaded part length of the bar represents estimated time duration of that activity, whereas the length of white part or a bar represents slack time, which the latest time by which a task must be finished. For, our library management System, Gantt chart is represented as show below:

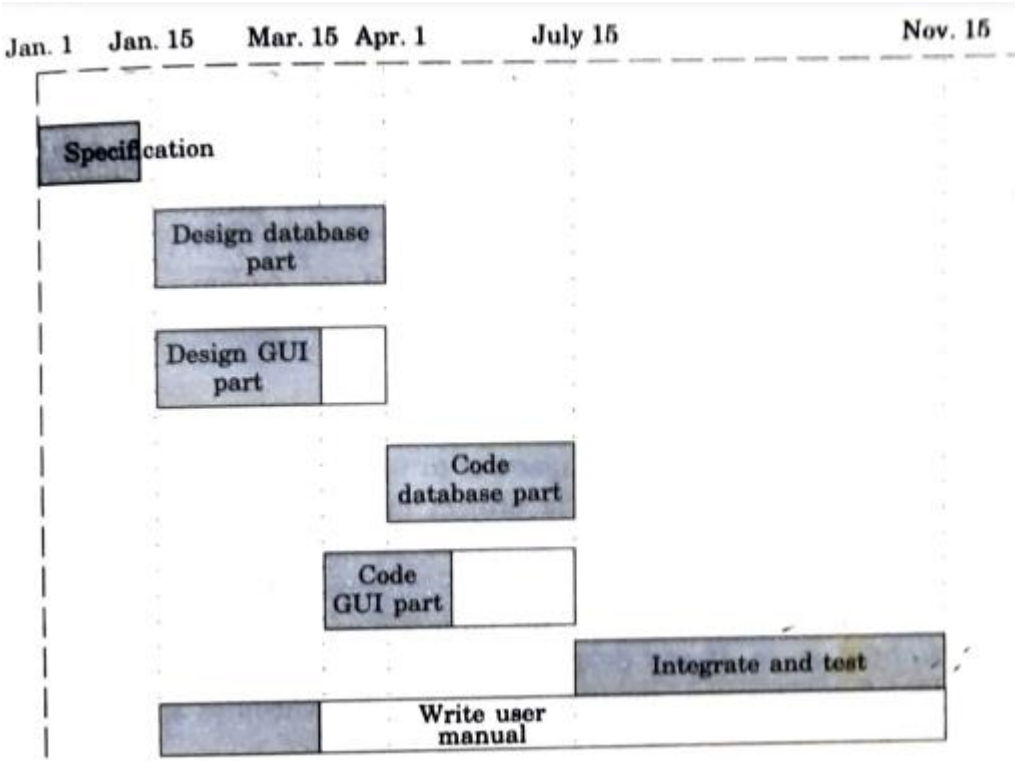


Fig: 5.4 Gantt chart for Library Management System

5.4.4 PERT Chart:

Project Evaluation and Review Technique (PERT) chart, consists of a network of boxes and arrows. The box in the PERT chart represents activity and arrow denotes dependency. In the CPM manager represent single estimate for number of days to complete the particular activity, whereas in PERT chart, manager put three estimates of time duration to complete each activity. These three different estimates for each activity are: optimistic estimate, most likely estimate and pessimistic estimate. The PERT chart for the library management system is shown in the following figure:

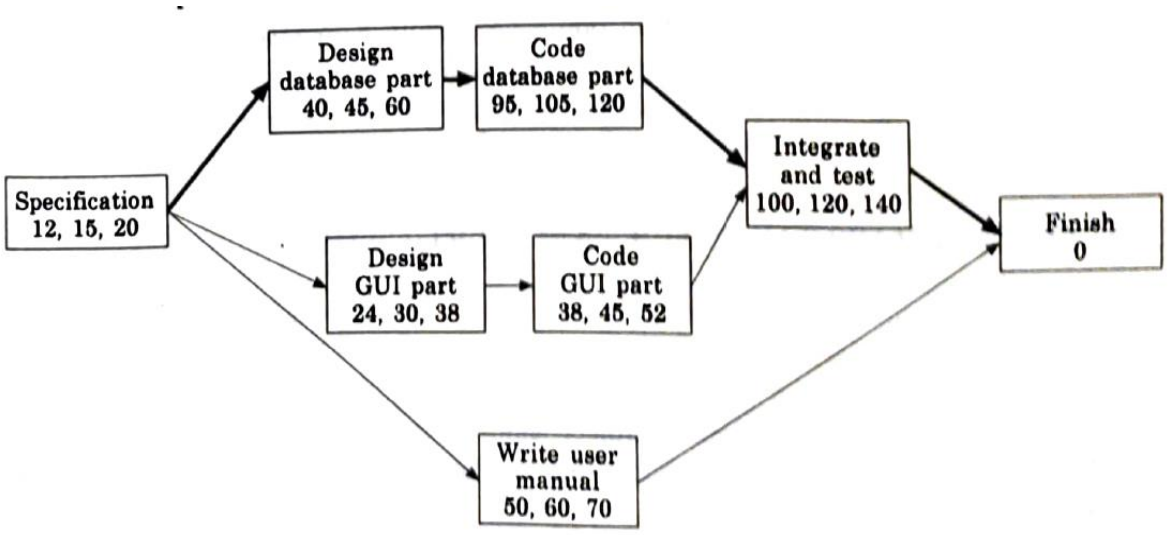


Fig: 5.5 PERT activity network diagram for Library Management System

The values 12, 15 and 20 in the activity 'Specification' represent, if everything done well and work done smoothly then this activity will be completed in it optimistic time which 12days. If during this activity, if challenges are faced, the work becomes difficult then it will take long (pessimistic) time to complete that is 20 days, otherwise the activity will be completed within it most likely (usual) time duration that is 15 days. So, here for each activity project manager takes three estimates about the time duration to complete particular activity.

5.5 TEAM STRUCTURES IN THE ORGANIZATION

Generally, most software development organization handles multiple projects at the same time. Software development organizations assign different teams of the engineers to handle different projects. So, one question can be arrived in the mind that how the whole software development organization is structured? Or how are individuate teams of software engineers structured? Let us discuss:

5.5.1 Structure of Software Development Organization:

There are mainly two ways in which software organization is structured. [1] Project format and [2] Functional Format

In the ***project format***, a several engineers are assigned to the particular project when the project starts and they have to continue their work till their project is not completed. Thus, the same team carries out all the activities of the life cycle. After competition of the project, the team is dissolved and engineers are placed in the different teams, where they assigned some other project.

In the ***functional format***, different tasks of a project will be performed by different teams of the software engineers, and also one team is also working on multiple projects. In this format, the staff is divided into functional groups based on their ability, specialization and interest.

5.5.2 Structure of team:

Structure of team addresses the issue of the development organization of the individual project teams. Here, we are discussing how an individual software engineer perform in the team. Usually, it is seen that the team structure can be divided into three categories. [1] Democratic team structure [2] Chief programmer team structure or [3] Mixed team structure.

[1] Democratic Team Structure:

In the democratic team structure, as the name suggests, team member follows democratic structure. That means they does not follow any formal hierarchy of team. The democratic organization leads to higher job satisfaction and morale. Accordingly, democratic structure suffers from less workforce turnover. This type of democratic team structure is usually suitable for less understood problems, where a group of engineers can find better solutions together rather than single individual that is group leader. See the following figure, which represent democratic team structure, here team members are coordinating with each other and there no central authority is there in the team.

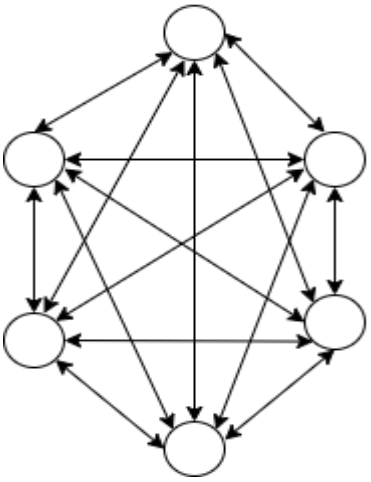


Fig: 5.6 Project coordination in Democratic Team Structure

[2] Chief programmer Team Structure:

In this type of team structure, senior engineer provides a role as technical leader for their team and is elected as chief programmer. The chief programmer decomposes the task given to his team into number of small activities and assign these activities to other team members. Other team members need to report periodically to their chief programmer about their progress in work. Chief programmer team structure is shown in the following figure. Darken circle is represent chief programmer, whereas white circles represent other team members.

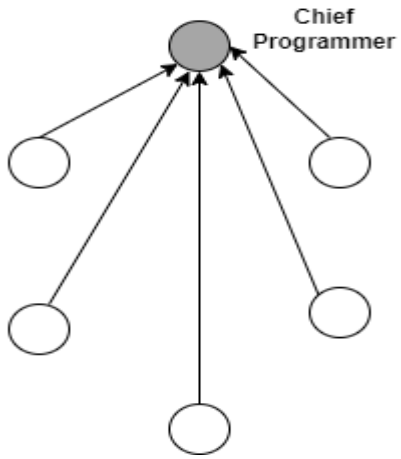


Fig: 5.7 Project coordination in Chief Programmer Team Structure

[3] Mixed Team Structure:

As the name implies, in the mixed team organization, team structure is formed by taking the idea from democratic team structure as well as chief programmer team structure. In the following figure, we have shown the mixed team structure. You can see in the figure, that mixed team structure forms a hierarchical structure. Here, different development teams need to report their senior engineer and similarly senior engineers needs to report to their project manager, which is shown as a solid line in the figure (as we have seen in the Chief programmer structure).

Not only that, withing the team junior programmers are also allowed to coordinate with each other, and in the same way senior engineers are also permitted to coordinate the project activities with each other, which is shown as dotted line in the figure (as we have seen in the Democratic team structure).

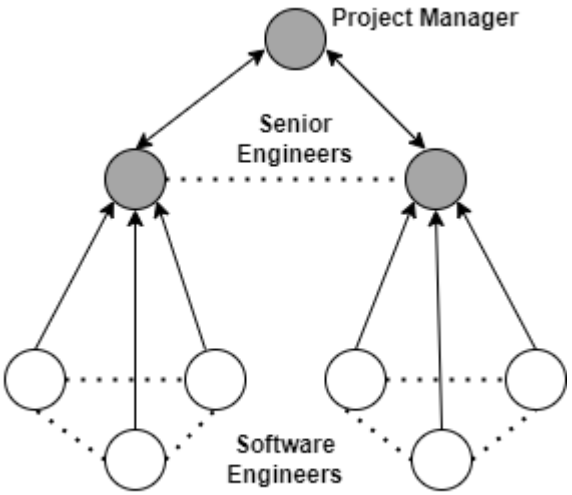


Fig: 5.8 Mixed Team Structure

5.6 STAFFING

Generally, project manager is responsible to choose development team. Therefore, they need to choose good developers and engineers for project success. It is observed that, in practical a worst engineer may decrease the overall productivity of the project. Therefore, it is essential to choose good engineers into the development team. But as a project manager, how can we identify good engineers? What are the qualities are there to becomes good software engineer? Well, we have mentioned some attributes of good software engineer below:

5.6.1 Skill required to be a good Software Engineer.

Several studies on the past software development projects, have be taken into the consideration to determine the skills in good Software Engineer. As per the study, a good Software Engineer should have following skills:

(i) Software engineer should be familiar with software engineering principles and systematic techniques

(ii) Software engineer should have good technical knowledge as well as knowledge of the project domain

(iii) Engineer should have good programming abilities

(iv) Software engineer should have good oral, interpersonal and writing skills

(v) Engineers should have high motivation factor

(vi) Software Engineer has to be intelligent

(vii) A good software engineer must have ability to work in a team

(viii) Discipline is obviously one attribute of good software engineer.

Check Your Progress:

1. Norden's and Putnam's work is related to _____.
2. WBS stand for _____.
3. CPM stands to _____.
4. In CPM, _____ is the longest path, represent estimate time duration to complete the project.
5. In _____ project manager annotated three optimistic, most likely and pessimistic estimates of time duration to each task of the project.
6. PERT stand for _____.
7. In _____ team structure all developers and engineers need to report to the senior project manager.
8. _____ choose the team members for software development in his/her team.

5.7 Let us sum up

In this chapter we have learnt how the estimation about staff can be made. Also, in this unit we have focus on how Work Breakdown Structure (WBS) helps in dividing project into number of tasks and project task into number of small and manageable activities. WBS also helps the manager to do planning and resource allocation for the software development project. We have also learnt how CPM chart, Gantt chart and PERT charts are useful in the scheduling of each activity, task and for entire project and also how can we prepare it. Finally in the unit we have discussed about how organization and team of the staff is organized within the organization. Finally, we have end up our discussion with what qualities should be there in a good software engineer, so that as manager you can select proper person (software engineer) into your team.

5.8 Check your progress: Possible Answers

Exercise: 1

1. Staff estimation
2. Work Breakdown Structure
3. Critical Path Method
4. Critical Path
5. PERT
6. Project Evaluation and Review Techniques
7. Chief Programmer
8. Project manager

5.9 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Unit 6: Requirement Engineering Process

6

Unit Structure

6.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know various techniques to eliciting requirements
- Understand requirement analysis and negotiation
- Understand non-functional and functional requirements
- Learn documentation of functional requirements
- Understand how to write good Software Requirement Specification (SRS)
- Learn different methods of representing complex logic

6.2 INTRODUCTION

The main objective of the Software Engineering is to develop, methodology or procedure for developing a software product for complex and large systems with superior quality, in minimum cost or designated time period. As we know Software engineering is a sequence of tasks, which focuses on different elements like analyzing, designing, implementing and organizing those elements into the form of system. It can be a product, technology or services. ***Software Engineering is a methodology which ensures engineer to build, produce and deliver right things, in right way and in right time***. As we know the entire process has to be done in various phases of SDLC, which starts from the feasibility study of the project, software requirement gathering and analysis, designing, coding, testing, implementing and providing support or maintenance. In this unit we will focus on software requirement analysis part. When the requirement for any software project is initiated, software engineer has to do detail feasibility study i.e. 'Whether the project is economical feasible? Is it technically feasible? And is it operationally feasible?', if answer is yes then software engineer has to gather required information (functional requirement of the project) and after analyzing all these functional requirements software engineers has to document these requirements into some proper and designated format. ***All other SDLC phases like design, coding, testing is strictly following this documentation, it called project management.*** So, if any mistake is made by the software engineer during requirement analysis, or if anything missed out by the software engineer during this phase then all other phases like designing, coding and testing will be affected.

Requirement gathering is also challenging task for the software engineers, especially when the customer is not aware with the computer software system. Usually, customer cannot give detail description of the various functional requirements of the project, that is just because of, lack of the technical knowledge or maturity level or do not having comprehensive knowledge. ***Systematic approach is needed by the software engineer to decrease the complexity in the area of requirement analysis, it is called Requirement Engineering.***

6.3 REQUIREMENT ENGINEERING PROCESS

Requirements Engineering is the systematic use of verified principles, techniques and tools which delivers cost effective analysis, documentation, on-going assessments of customer's need and the specification of external behavior of the system to satisfy necessity of the customer. Requirement Engineering Process can be defined as a discipline, which addresses requirements of different objects of system development process.

Computer-based specification or a software product, which is described at various level is the output of the system requirement specification. The requirement engineering process can be performed with the following steps:

- Requirement elicitation
- Requirement analysis and negotiation
- Software Requirement Specification (SRS)
- System modelling
- Requirement validation
- Requirement management

6.4 REQUIREMENT ELICITATION

Requirement elicitation or gathering is an art. It seems to be easy process of asking the user, stakeholders or customers of the system about the various functional requirements of the system. But it is not that much easy enough and number of problems involved in this process. Due to this problem requirement gathering is becomes complex and need more attention of the engineer. The problems might face by a software engineer are discussed below:

- ❖ Project Scope: The boundary limits of the project is not clear and sometime it may happen that customer specify those details which is not there in the project scope.
- ❖ Useless technical details: Instead of providing clear and precise information, customer provides too much technical details which may create misperception and surge complexity.
- ❖ Problematic understanding: If the customer is not clear with particular task, or having weak or incomplete details, then it will mislead to software engineer.
- ❖ Requirement volatility: Problem of volatility (frequent changes) in requirement occurs when the project requirements change over the time.

Software engineer who gathers information from the customer should have knowledge of what, when and how to gather functional requirements and by using which resources. The information is collected for the organization which comprises its objectives, policies, organizational structure, all stakeholders and staff of the organization.

The following tools are helpful to the software engineer in gathering of all functional requirements about the project.

A. Review of the Records: Software engineer need to review the various recorded documents of the organization. Different transaction books in which transactions are recorded, several procedures, forms etc. are studied to gain knowledge of different formats and functions of existing system. This is time consuming method.
B. On site observation: Here the software engineer needs to visit the actual site to get closely observe and understand the system correctly.
C. Questionnaire: This method provides effective way to gather the information with fewer effort, so that an engineer can produce written SRS document about requirements. This method examines large number of respondents parallelly and get their customized responses. It gives adequate time to the respondents to select the proper answer of the questions.
D. Interview: In this method Software Engineer takes a personal interview with each stakeholder of the project and identify their needs (requirements). It requires experience of placing the interview, setting the stage, avoiding arguments and assessing the outcome.

The outcome produces by the requirement elicitation process can differ depending on the scope of the system or type of the product to be produced. In most of the case the output document covers following points.

- Statement for the system requirement and its feasibility study.
- Scope or boundary of the system or product. (Functions which are included and functions which are not covered in the system).
- List of stakeholders participated in the requirement elicitation process.
- Details of the technical environment.
- List of the functional requirements.
- In some cases, prototype is built to perform requirement elicitation in a better way.

6.5 REQUIREMENT ANALYSIS AND NEGOTIATION

The outcome produced of the requirement gathering done earlier will be the input of the requirement analysis. In the requirement analysis each requirement which is recorded earlier will be observed sensibly and categories them into related sets. To classify requirements into different sets the relation between the requirements is studied. At this stage each requirement is observed whether requirement is correct, or has some mistake and it is ambiguous.

Requirements can be classified in to three types depends on their priority.

1. Requirements which are absolutely met
2. Requirements which are highly needed but not essential
3. Requirements which are possible to implement but could be abolished

Explicit & Implicit Requirements:

Explicit Requirements are those, which are stated by the customer. Those requirements which customer can easily stipulate and able to give broad description are called explicit requirements. For example, in the development of online banking application ‘deposit’ or ‘withdraw’ requirements will be explicit and an engineer gets complete particulars that is input, process and output about the requirements from the customer. Whereas, some requirements will not be mentioned or explained by the customer, but they should be specified in the requirement specification documentation by software engineer by their ability or skill, it is called implicit requirements. In the

Online banking application, it is a responsibility of the software engineer to validate the input data and give correct and suitable validation messages, while customer of that bank is filling online form. Software engineer can specify that 'Account Number' should be generated automatically by the system when new account is created, Email address and phone number must be validated by OTP methods and so on. Such requirements may not be denoted by the customer. These requirements are written by the software engineer to behave the system properly.

Source of Information

Source of information plays vital role in the requirement specification. Software engineer has to visit various stakeholders of the system to acquire requirements. Different stakeholders of the system will describe different requirements and also sometime, the same requirement in a different way. While writing the requirement specifications software engineer has to reference the source of the requirement. So, in future, in any phase, of the life cycle any confusion is occurs, then we can easy resolve that confusion, if we know the source. In such case, we can visit to that particular source (stakeholder of the system), we can get more detailed clarification about the requirements and so it's implementation in the system.

Types of Requirements:

On the basis of functionality, requirements can be classified into following two types:

1) **Functional Requirements:** In the functional requirements various aspects like input-output formats of the system, their structures of data storage, computational abilities, timing of the task completion and synchronization are measured. Functional requirements cover set of transaction in series which can easily expressed in the term of function. For example, finding of a book in the Library Management System, or cash withdrawal from the ATM system can be measured in this category.
2) **Non-functional requirements:** In the non-functional requirements various issues like security, performance, quality, efficiency, usability, reliability and probability is considered. Because of the non-functional requirements deals with the characteristics of the system, they cannot be expressed in the form of functions. Non-functional requirements focus on security issues, reliability issues, accuracy of the result and interfacing between computer and human.

How to write Functional Requirements?

To prepare documentation of the functional requirement into the SRS, software engineer has to postulate set of functionalities supported by the software system. For each task or function what data is crucial to input, what information is generated as an output and description about that process i.e., how an input data is processed to generate output. Some examples are given below to clarify how functional requirements can be documented in the SRS.

Example:1 Online Sales

Requirement: 1 Sales

Description: The sales function first shows the numerous categories of the product. When user selects particular category then all the products come under that category should be displayed. When user selects any particular product then detailed description about that product such as product features, price, rating and reviews of that particular product is shown. When customer selects option 'place order', then stock of the product is verified. If product is not available in the stock, then give proper message otherwise show payment options to the user. After payment is made generate and produce invoice to the customer.

Requirement 1.1 Select category

Input: Category name

Output: All products belong to selected category is shown

Requirement 1.2 Select product

Input: Product name/code

Output: Details description of product like features, price, rating, feedback and buy option

Requirement 1.3 Select Buy option

Input: Product name/code

Output: Check stock availability of the product. Give suitable message if product is not in the stock. Provide payment options if product is available.

Requirement 1.4 Select Payment option

Input: Payment details (Types of Payment, Card details, OTP, etc.)

Output: If payment is made successfully, show invoice details else display proper error message.

Example:2 Find Availability of Library book

Requirement: 2 find books

Description: When the user selects the option 'find book', user would be prompted to enter keywords. When user click on find button after entering keywords, system would search for the book in the library database and all the books whose title or author name is matched with the keyword given by the user, details are displayed to the user. The book details include title of the book, name of the author(s), it's ISBN number, catalog number and position (place) in the library.

Requirement 2.1 Select option find

Input: 'find' option

Output: user is instructed to input keywords

Requirement 2.2 Find

Input: Keywords

Output: Title of the book, Author's name(s), it's ISBN number, Catalog number, position of the book in the library

Negotiation:

It is also possible that different stakeholders of the system, suggest different requirements because they always work with limited business resources and it is also not possible for an engineer to fulfil all the requirements. The system engineer must resolve such types of conflicts by a process call negotiation. Customer or the stakeholder of the system are asked to rank their requirements on the basis of priority and on the basis of it conflicts in the requirements can be discussed or negotiated.

6.6 REQUIREMENT SPECIFICATIONS

In the software system, specification means different things for different people. A System Requirement Specification (SRS) can be a written document, diagrams, a mathematical model, prototype or combination of any of these.

Some software engineers believe that the requirement specification has to be written using the specific predefined format and several standard templets (predetermined formats) has to be there, so that SRS will become precise, clear, easy to understand and consistent. But sometime it is also necessary to be flexible in it. The system requirement specification (SRS document) is the final outcome produce by software engineer after doing requirement gathering and analysis. System designer will use this documentation to design the system. Not only in the system design but the SRS document will also helpful in the coding of the system. In fact, this document is also important in system implementation and testing and maintaining the system. If any disagreement occurs in the future related to the accomplishment of the requirement, then it can be resolved using this document.

As this is very important document, used in all future SDLC phases of the system and covers all the functional and non-functional requirements of the customer, it should be concise, correct, consistent, clear, unambiguous and complete document.

The format or template of the SRS document is given below:

1. Introduction
    1.1. Purpose
    1.2. Scope
    1.3. Definition, acronyms and abbreviations
    1.4. References
    1.5. Overview
2. Overall description of the Product
    2.1. Description of product
    2.2. Environmental characteristics
        2.2.1. Hardware
        2.2.2. Software
        2.2.3. People

2.3. Functions of the product
2.4. Characteristics of the user
2.5. Constraints if any
2.6. Dependencies and Assumptions made any
2.7. Goal of implementation

3. Requirement Specification
   3.1. Input output interfaces
   3.2. Functional and non-functional requirements
   3.3. Performance requirements
   3.4. Logical database requirements
   3.5. Design constraints
   3.6. Software system attributes
   3.7. Behavioral Description
      3.7.1. System states
      3.7.2. Events and actions
   3.8. Organizing the specific requirements
   3.9. Additional comments
4. Supporting information
   4.1. Index or Table of contents
   4.2. Appendixes

Characteristics of Writing Good SRS:

In fact, writing good SRS document is talent and can be achieved by experience. If the analyst keeps the following characteristics of SRS in mind, then decent quality of SRS can be written.

1) Conciseness: SRS document has to be complete, unambiguous, and concise. Conflicting and irrelevant information should be removed so that the document should be readable.
2) Well-structuredness: SRS document has to be well structured and it has to be in specific format as we have deliberated above. Structured SRS is always easy to understand.
3) Black-box viewing: Analyst has to write only black-box view of the events, which only highlight on what system has to do and not how system will do. Only the input data available and output information to be produced is deliberated.

This property asserts analyst has to write only external view of function and not coding.

4) Verifiable: The analyst has to verify; implementation of each requirement is whether it is feasible or not. Those requirements which are impossible to implement are listed distinctly in the goal of implement section of the SRS document.

Problems in writing good SRS

There are several problems, from which SRS document may suffer, which are conversed below:

1. Over-specification: If too much specification details are given to specific functionality, which makes document more lengthy, complex and ambiguous. This problem will occur when analyst also includes 'how to' aspects of the problem.
2. Forward referencing: While writing the SRS document, an engineer should avoid too much forward referencing (Reference to the points are discussed much later). This makes document boring and decreases readability of the document.

6.7 SYSTEM MODELING

Assume for a while, if all the essential components, parts and instruments are given to an automobile engineer, in this case an engineer will simply arrange the related components and parts together and prepare model of a vehicle. It is nothing but a blueprint or 3D modeling which describes position of each instrumental part or component of a vehicle. System modelling is similar to this. Once requirement gathering and analyzing is performed then, a system model is prepared which shows how the requirements will fit into the system. Here the relations between different requirements are absorbed and model of the system will be prepared.

6.8 VALIDATING REQUIREMENTS

In the validation of requirements, those requirements which are gathered and analyzed are evaluated for quality. In this process each requirement is inspected the

specification to confirm that all system requirements have been specified in the SRS are valid or not. By mean of requirement validation, any unpredictable, unclear, unrealistic or unfeasible requirements filtered out and resolved. As outcome of requirement validation, we have only clear-cut requirements in the SRS document. To validate requirements following questions has to be asked:

- Are requirements described clearly? Can they be misinterpreted?
- Is requirement source identifiable? Outcome has been examined against the source?
- Can we bound the requirements to quantitative terms?
- Is any requirement violating any domain constraints?
- Can we test the requirement?
- Is the requirement traceable to any system model or objective?
- Is the requirement do affect the system performance?
- Is the requirement described in the proper format?

Checklist of the above given questions for each requirement mentioned in the SRS document will help software engineer to validate all requirements.

6.9 REQUIREMENT CHANGE MANAGEMENT

During the SDLC of the computer-based software system, it is possible that requirements may change frequently and that also desire to change in the SRS document. ***Requirement management is a set of activities that help the software engineer to identify, control and track requirements and changes to requirements at any time as the project proceeds.***

6.10 HOW TO REPRESENT COMPLEX LOGIC

If the SRS is sensibly designed, then all the conditions will be properly considered in it. However, some conditions have complex logic, and more interactions and processing sequences are required. It is difficult to describe complex conditions into the textual data format. It is also not possible to check large number of alternatives if the complex condition is written in plain text data format. In such cases, decision

tables or decision trees can be used. Decision table and Decision tree is supportive to describe such complex conditions having large number of alternatives.

Decision Tree:

Decision tree is a pictorial representation of the complex conditions with all its alternatives and action taken corresponding to each alternative in hierarchical (tree shaped) manner. Decision tree show the logic structure in a horizontal form that look like a tree with the root at the left and branches to the right. Similar to flowchart, decision tree is also useful way to represent complex functions of the system. Decision table and decision tree represent the similar thing. Just decision tree represents all alternatives in hierarchical way, which becomes easier for the programmer to understand.

Decision Table

Decision table represents a complex logical structure, with all possible circumstances, and resulting action in the tabular form. It represents the policymaking logic and its corresponding action taken in the matrix form. The upper row of the decision table shows conditions (condition stub) to be evaluated and the lower rows represent actions (action stub) to be taken. Column of the table is known as rule. If the condition is TRUE then rule will be enforced and if it is FALSE then rule will not be enforced.

Consider the following example given below. Based on the situation described in it, we draw the decision tree and make decision table.

Example: 1

Consider SALES POLICY of the ABC company. Company offers 2% discount to its non-prime customers. If the customer is of prime category, then 5% discount will be given. If any prime customer gives order of more than Rs. 5000 then 10% discount is given. If any prime customer places the order of more than Rs 5000 and making the payment using credit card then addition 2% discount (12% discount) will be provided.

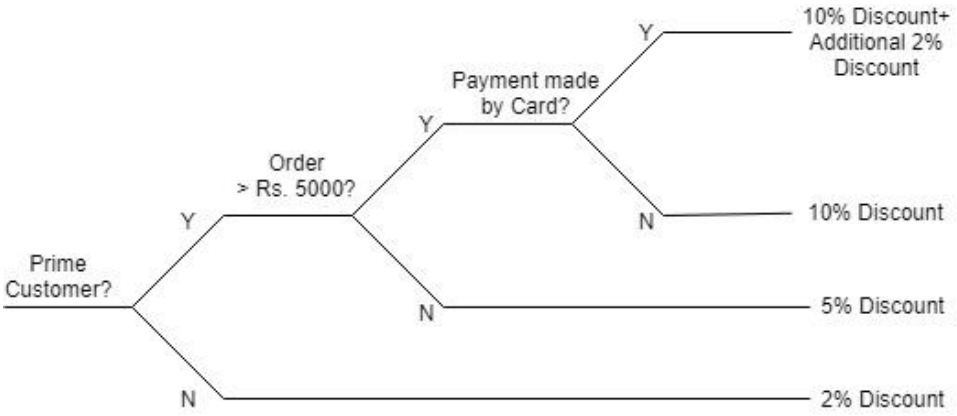


Figure: 6.1 Decision Tree for the sales policy of ABC company

	1	2	3	4	5	6	7	8
Prime Customer	Y	Y	Y	Y	N	N	N	N
Order more than Rs. 5000	Y	Y	N	N	Y	Y	N	N
Payment made by card?	Y	N	Y	N	Y	N	Y	N
2% Discount					X	X	X	X
5% Discount			X	X				
10%Discount	X	X						
Additional 2% Discount	X							

Table:6.1 Decision Table for the sales policy of ABC company

Exercise:1 Fill in the blanks

1. _____ represents graphical representation of the complex logic structure.
2. In the decision table upper rows represent _____ and lower rows represents _____.
3. After requirement gathering and analyzing, _____ work flow is used to prepare a blueprint or 3D rendering of the system.
4. _____ is used to resolve conflicting requirements.
5. _____ work flow is a process, in which analyst will check each requirement whether its implementation is possible or not.

6.11 Let us sum up

In this chapter we have learnt how requirement can be gather from the customer, how can we analyse and validate them. We have also discussed that what are functional and non-functional requirement and how can we write functional requirements. We have seen what is SRS document? What is the user of SRS documents and attributes of good SRS document? Finally, we have learnt how to represent complex logic using decision tree and decision table. We hope, now student will have sufficient idea of how to gather, analyse, document requirement in proper format.

6.12 Check your progress: Possible Answers

Exercise: 1

1. Decision Tree
2. Conditions, Actions
3. System Modeling
4. Negotiation
5. Requirement Validation

6.13 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

6.14 Activities

1. Consider a library automation system, where number of books are there in the library and various library members can borrow books (not more than 3 books at a time). Members can keep the book with them for maximum 3 weeks, and they have to return the book(s) within this period. If any member fails to return the book(s) within designated time then Rs.1/- per day fine has to pay as penalty by the library member. Fine amount should not be more than 5000/-. Every member needs to renew the membership with Rs. 1500/- per year basis. Initial membership and Registration charge will be 2500/-. Write functional requirements for the case discussed above. Make suitable assumption when needed.
2. prepare SRS document for any online shopping website system. Make suitable assumptions for the various functional requirements of the system.

Software Engineering

BLOCK 3: SYSTEM ANALYSIS AND DESIGN

BLOCK 3: SYSTEM ANALYSIS AND DESIGN

Block Introduction

In this block-3 of the Software Engineering, we are discussing how the requirement analysis can be represented in the various standard forms. There are two approaches are there to represent the system analysis: (1) Structural Analysis and (2) Object-Oriented analysis.

In this block we have discussed how structural analysis can be done by preparing Data Flow Diagram (DFD) and Entity Relationship Diagram (ERD). Similarly, when Object-Oriented methodology is used in the requirement analysis then several diagrams like use-case diagram, sequence diagram, collaboration diagram, activity diagram, state chart diagrams are drawn to represent different activities and to represent interactions of different user with the system. To represent back-end in the object-oriented analysis class diagram or object diagram has to be draw. How, an engineer can draw all these diagrams? What are the rules for drawing of all these diagrams? – is discussed briefly in this block.

Block Objective

The objective of the block is to explain doing structured and object-oriented analysis for the system. The main aim to explain how an engineer or project manager can depict his/her analysis by making of DFD and ERD in the case of structured analysis and use-case diagram, sequence diagram, class diagram, object diagram, state chart diagram and activity diagram kind of UML diagram in the system requirement specification documentation.

This block, has details discussion which clear reader's concept on structured and object-oriented analysis. Reader of this block, will learn rules for preparing different diagrams, and basic terms related to analysis. In this block, we have also mention case studies so that reader can understand how to apply the rules discussed for each diagram discussed can be applied on any real or hypothetical system.

Block Structure

BLOCK 3: SYSTEM ANALYSIS AND DESIGN

UNIT 7 STRUCTURED ANALYSIS MODELING

Objectives, Structured analysis, Data Flow Diagram (DFD), Example of DFD, Entity Relationship Diagram, Types of relations, Example of ERD, Let Us Sum Up

UNIT 8 OBJECT-ORIENTED ANALYSIS AND DESIGN

Objectives, Basic terms of object-oriented analysis, UML diagrams, Use-case diagram, Class diagram, Sequence diagrams, Requirement management, Analysis modeling, Let Us Sum Up

UNIT 9 UML-DIAGRAM OF SYSTEM – A CASE STUDY

UNIT 10 SOFTWARE DESIGN

Objectives, Feature of good software design, Design concepts, Cohesion and Coupling, Design modeling, Pattern based software design, Let Us Sum Up

Unit 7: Structured Analysis Modeling

7

Unit Structure

7.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know different methods of analysis
- Understand details of Structured analysis
- Draw system Data Flow Diagram (DFD)
- Learn important terms of database management system
- Understand different types of cardinalities between entity sets
- Draw Entity Relationship Diagram (ERD) of a system

7.2 INTRODUCTION

As we have seen in the previous unit that modeling is a method of transforming textual description about the problem into the graphical representation. After the requirement elicitation, analysis and validation, an engineer have all clear, concise, non-conflicting and implementable requirements in the textual format. In the analysis model those requirements are transform or represented into the graphical model in the form of different diagrams.

Analysis modeling can be performed by two methods:

1. Structured Analysis
2. Object Oriented Analysis

7.3 STRUCTURED ANALYSIS

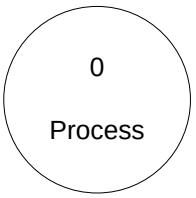
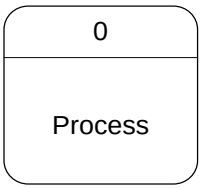
Structed Analysis is a classical method of analysis modeling. Structured analysis follows function-oriented methodology to carry out top-down approach to decompose the set of high-level function characterized in the problem definition and represents them graphically. In this method, system is divided into various functions. That is, each function which system performs is analyzed and decomposed into more comprehensive functions. For example, function ‘Sales management’ can be divided into ‘Order, ‘Sales’, ‘Payment’ and ‘Product return’. In the implementation of Structured Analysis following principles are essential.

- Structed Analysis follows Top-down decomposition approach of the system.

- It uses a principle of divide and conquer, as each function is independently decomposed.
- Outcome of the structured analysis is Data Flow Diagram (DFD).

7.4 DATA FLOW DIAGRAM (DFD)

A Data Flow Diagram (DFD) is as hierarchical graphical model, which represents the whole system, and its decomposition into number of functions. In the DFD each function is a process, and the flow of data between entity and process, as well as flow of data between process and data stored are characterized in the graphical form. DFD is also known and bubble chart. DFD uses few symbols which are denoted in the following table.

Sr. No.	Symbol (De Marco & Yourdon Notation)	Symbol (Gane & Sarson Notation)	Explanation
1			Process symbol is used to receive input data, process it and generate output. Output may be processed data (information) in the desire format or content. Process may be simple or complex (can be decomposed further). Process contains business idea or business logic. Process name appears in the circle in Yourdon notation or Rounded rectangle in Gane & Sarson notation. Process denotes functionality or action so usually its name should be verb. For example, manage purchase, return book, deposit money etc.
2			The rectangle int Yourdan notation and shaded rectangle in Gane & Sarson notation is used to represent entity. Entity name should be represented inside the rectangle/shaded rectangle. DFD shows

	External Entity	External Entity	only external entities which directly interacting with system and provides input data to the system or receives information from the system. For example, Member, Supplier, Customer, Teacher, Student etc. are the examples of the entities. Entity is also called terminator because it is an original source of the data or it will be the final destination which receives processed data (information).
3	Data Store	D1 Data Store	DFD also uses data stores to represent the database table in the system for future use of data. If a process stores the data in data store, then that can be used by the same or another process(es). Data stores represent a kind of data tables of the database. For example, in the online examination system, response of each student is recorded in the user_response data table, which will be retrieved and matched with correct options in result generation process.
4			Data flow indicates path of the data to move from one end of the system to another end. Data flows are indicated in the DFD by arrows, and which data is flowing by the arrow is denoted by a label on it. Usually, data are noun, and therefore the caption or label on the arrow has to be noun. For example, student details, customer details, product details or book details etc.

In the DFD, for each process, which data takes by a process as input and what is generated by that particular process as an output is described. But how the data is being processed or logic of the process is not illustrated, so DFD represents a black-box view of the system.

How to draw DFD:

DFD model represent flow of the data graphically and in hierarchical manner of the levels. DFD always starts with most abstracted view (low level) of the system and then at each higher level more details are introduced successively. When most abstracted view of the system is presented in the DFD, it is called context level DFD diagram.

1. **Context DFD Diagram:** The context level DFD represent most abstracted view of the whole system, which has only one process that represents entire system. All the entities which interact or communicate with the system are shown in the DFD and their communications with the system are shown by data flow arrows. The context diagram is also called as 0 level diagram. Because of context level DFD is at most abstracted level and represents whole system, all entities and their interactions, it becomes more complex. So, in the context level of DFD, no data store is present.
2. **1st Level DFD:** To develop the 1st level DFD, we need to examine the high-level requirement functions. If there are 3 to 7 high-level functional requirements are there, then that can be denoted in the 1st level DFD. If more than 7 high-level functional requirements are there, then similar types of functional requirements are merged into one function and that can be decomposed in the 2nd level DFD.
3. **2nd Level DFD:** Those functional requirements which can be separated in the 1st level but due to the constraint that maximum 3 to 7 processes can be shown in the 1st level DFD are separated or decomposed in the 2nd level.

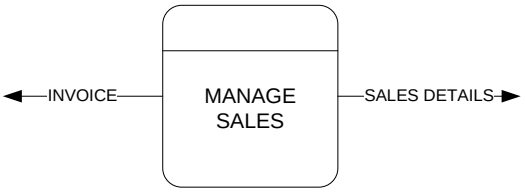
So, we can assume that context level DFD is nothing but a black-box view of whole system. In the 1st level DFD we divide the entire system into 3 to 7 different modules, and each module represented in the 1st level can be again decomposed into number of forms and reports in the 2nd level.

Process Numbering:

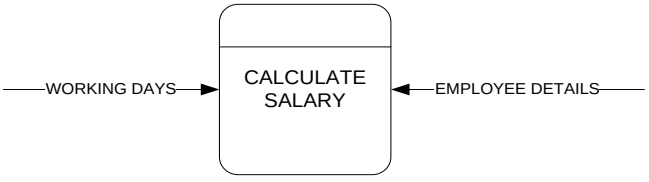
It is necessary to number the different processes of DFD to consistently to identify all processes uniquely. The process at context level is generally assigned the number 0, to denote, it is 0-Level DFD. Processes at 1st level DFD are numbered as 0.1, 0.2, 0.3 and so on. Suppose process 0.2 of 1st level DFD is further divided in to 3 more processes at 2nd level DFD then, it should be numbered as 0.2.1, 0.2.2 and 0.2.2 and so on.

Rules of drawing DFD:

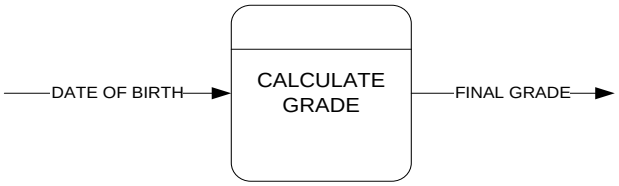
1. Context level DFD must have only one process that represents whole system. All entities must be present in the context level DFD, and context level DFD must not have any data store.
2. Name of the processes are mostly verb as processes are indicating functionality (which perform some action) and name of the data flow (arrow) must be noun as it is indicating what type of data is flowing.
3. There is no data flow has to there from one entity to another entity directly. Those transaction which are performed between entity and system is not involved in it should not be presented in the DFD.
4. DFD also should not have data flow from entity to data store. Generally, entity give the data to the process and process will store that data in the data tables.
5. No process should be there in the DFD, which has only outgoing edges. If the process has only outgoing edges and no incoming edge then it is called a problem Spontaneous generation.



6. No process should be there in the DFD, which has only incoming edges. If the process has only incoming edges and no outgoing edge then it is called a problem Black hole.



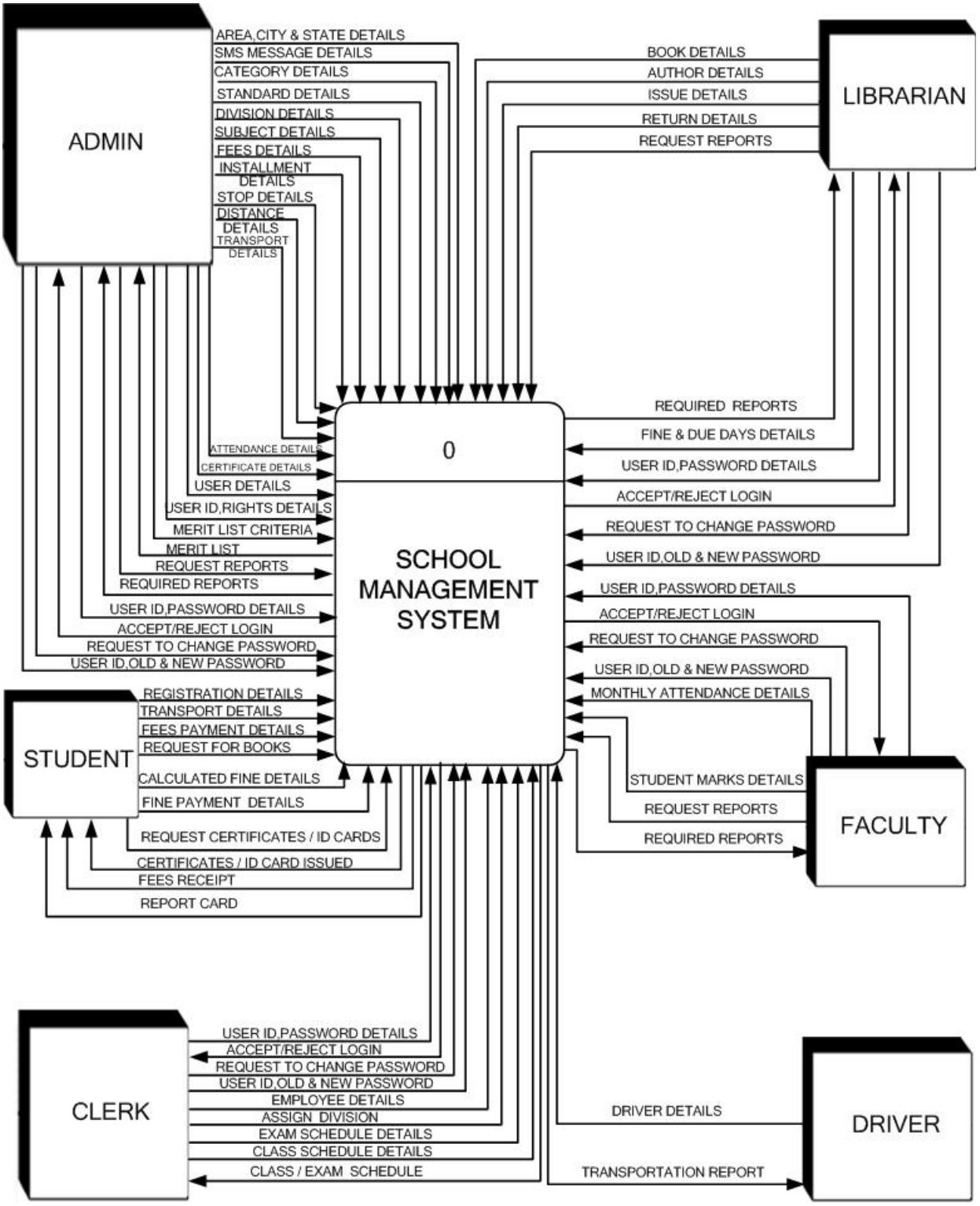
7. If the process of the DFD has one incoming and one outgoing edge, but the input data to process is not sufficient to produce output then that problem is called Gray hole problem. Such type of process should not be there in the DFD.



8. When we change the level of the DFD then data flow in to the particular process and data flow out from the particular process should be same. This means when we draw the 2nd (next) level DFD then number of data items in and number of data items out should match with the 1st (parent) process. Number of in and out data elements should not be increased or decreased while changing the level. It is also called balancing of DFD.
9. DFD has to be concise and clear. Arrows (data flow) should not cross each other. Otherwise, it makes DFD more confusing.
10. We can repeat the Entity or Data table to avoid crossing of data flows (arrows), but these repeated objects have highlighting cross line on the top-left side.

7.5 DFD OF A SCHOOL MANAGEMENT SYSTEM

In this section we have denoted a Data-Flow Diagram (DFD) of the school management system. Students are getting enrolment in the school by filling registration form. Student needs to pay the fees on regular interval (semester basis). Fees details are managed by the clerk in the school office. Student gives examination and which will be evaluated and result will be prepared by the teachers. School also has small library in the campus, from which students can issue the books. Library system handles by the school librarian. School also provides Transportation facility. Entire system managed by Administrator, and administrator have full access of this system.

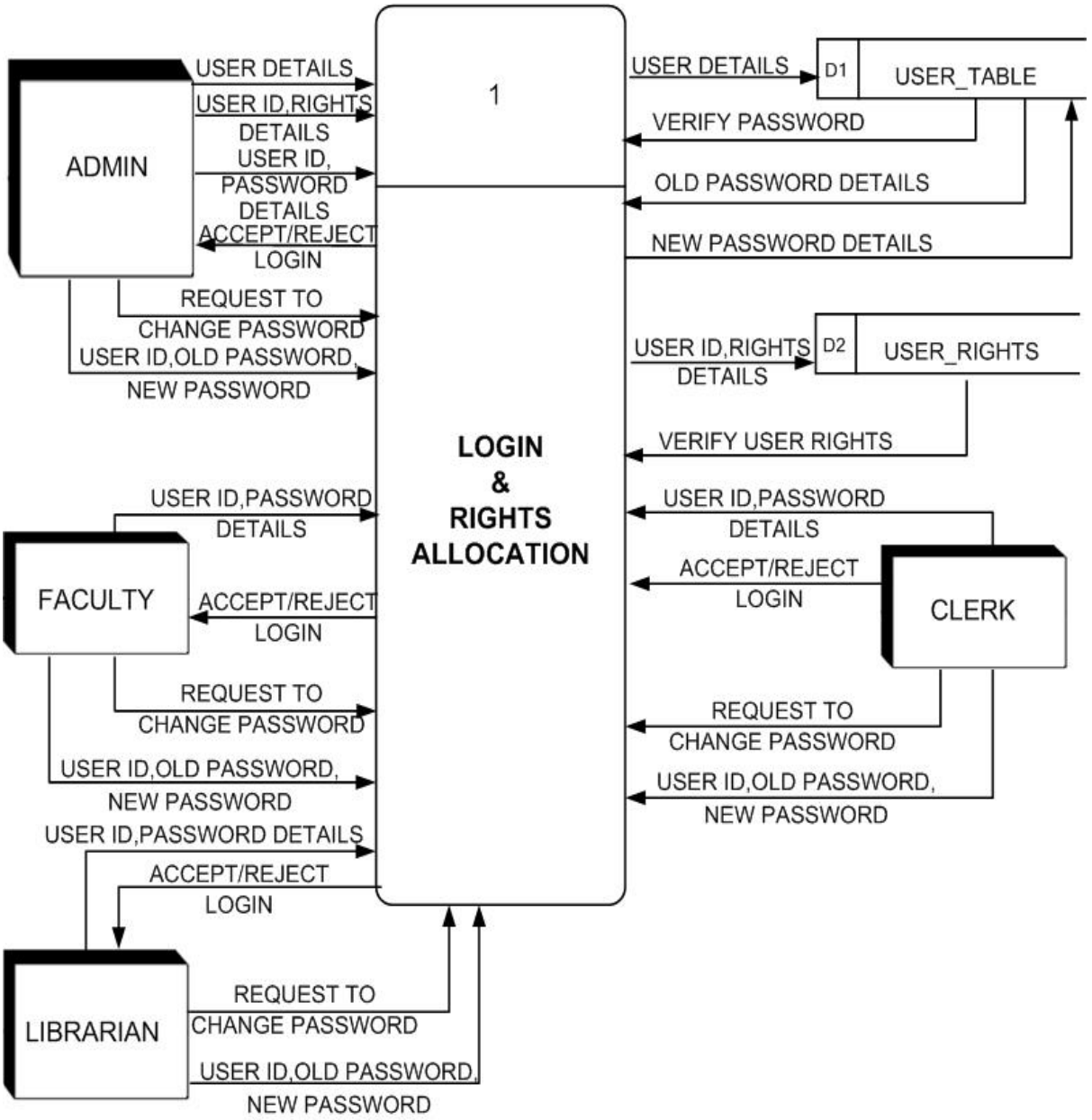


Context diagram of the school management system is shown above. Which has 6 entities Student, Faculty, Admin, Librarian, Driver, and Clerk. They interact with school management system.

Note: Generally, all the first level processes of the system have to be drawn as a single diagram, where only one instance of each entity and data table is placed. But, because of it increases complexity and degrade printing clarity we have shown each first-level process disjointedly. If the system is smaller and all

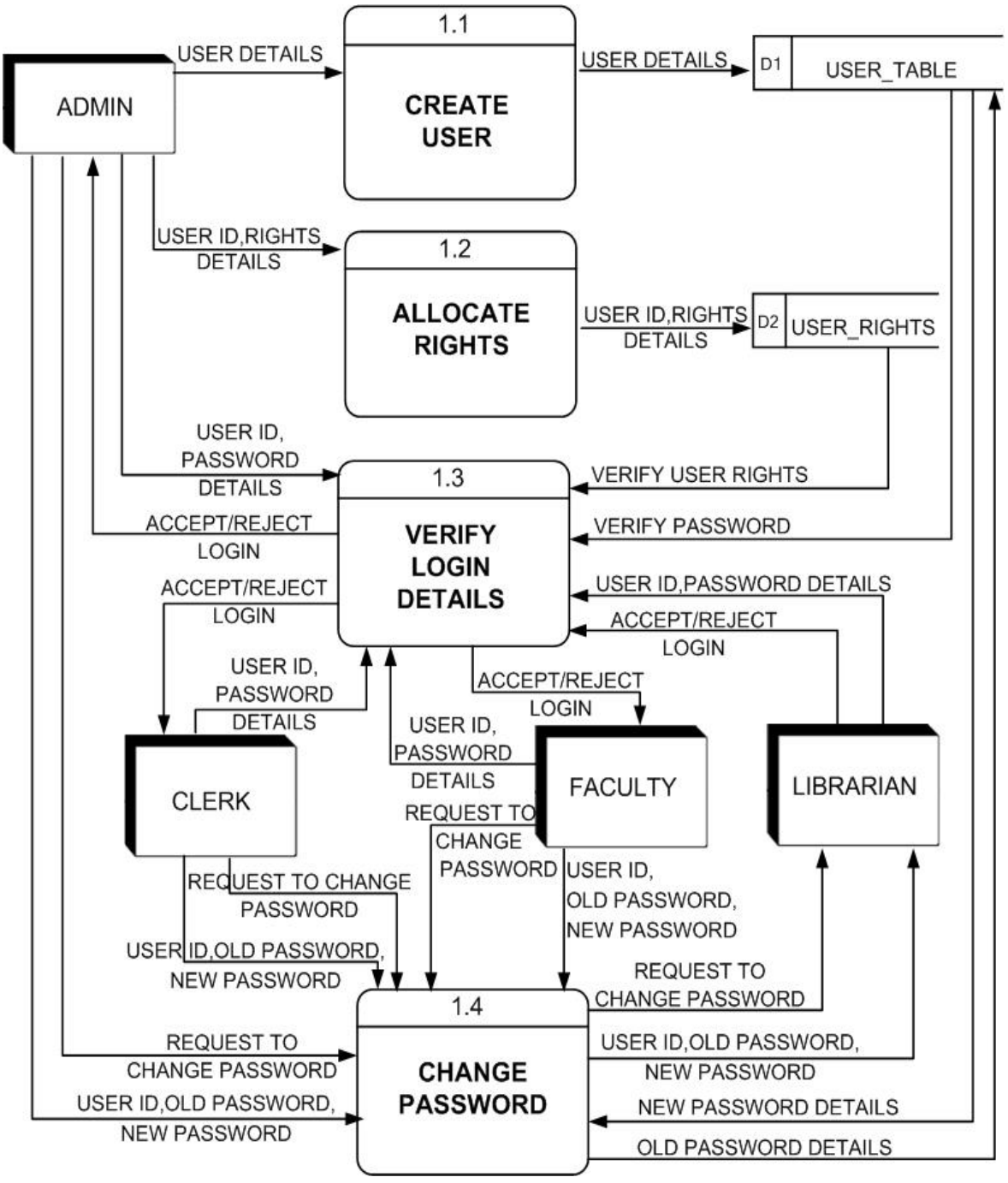
first level processes can be lodged in a page then it is preferable that you draw a single first level DFD which accommodate all the 1st level processes in a diagram.

Login & Rights Allocation (1st – LEVEL):



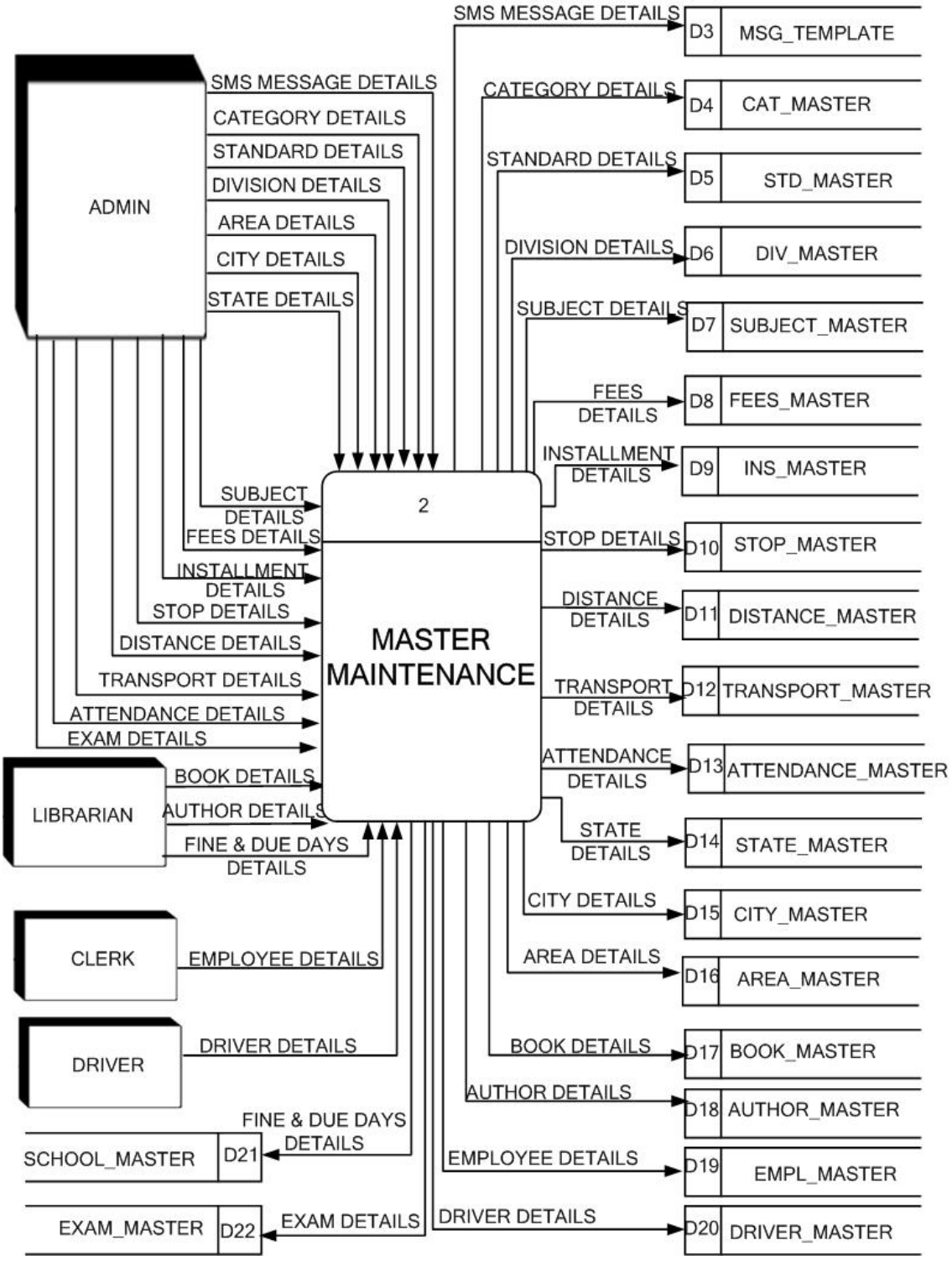
The above figure shows the 1st process of the 1st Level DFD. Different users like administrators, faculty, librarian etc. can log into the system and based on their role, system will give privileges (authorization) to the user. The process is complex and its 2nd level decomposition is needed. Its 2nd level DFD is shown in the next figure.

Expansion of Login & Rights Allocation (2nd LEVEL):



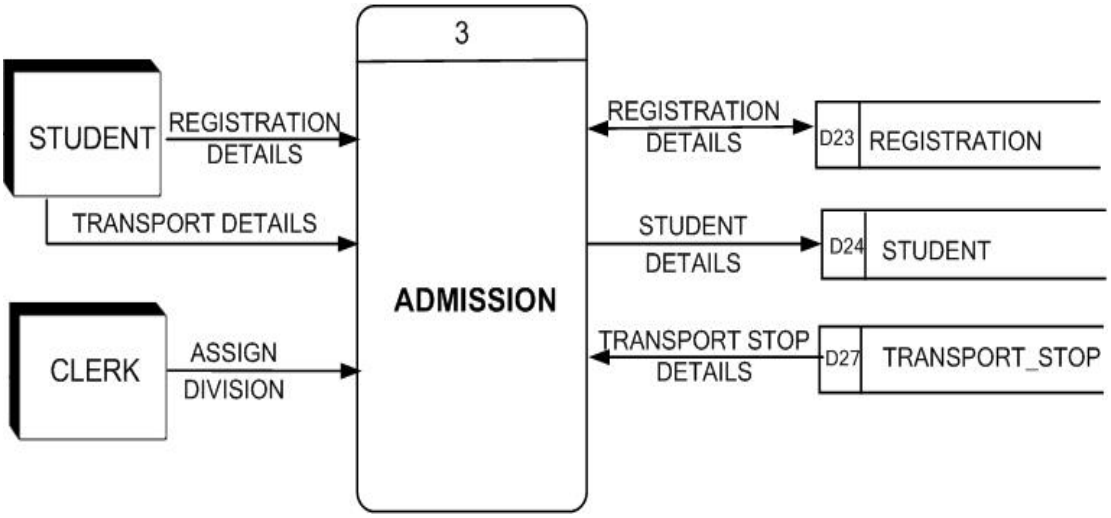
The entire login process of 1st Level, we have decomposed into Create user, allocate rights, verify login details and change password sub-processes. Next process in the DFD, is to record master details for all master tables in the database. The process is very simple; hence second level transformation is not necessary.

Master Maintenance (1st – LEVEL):

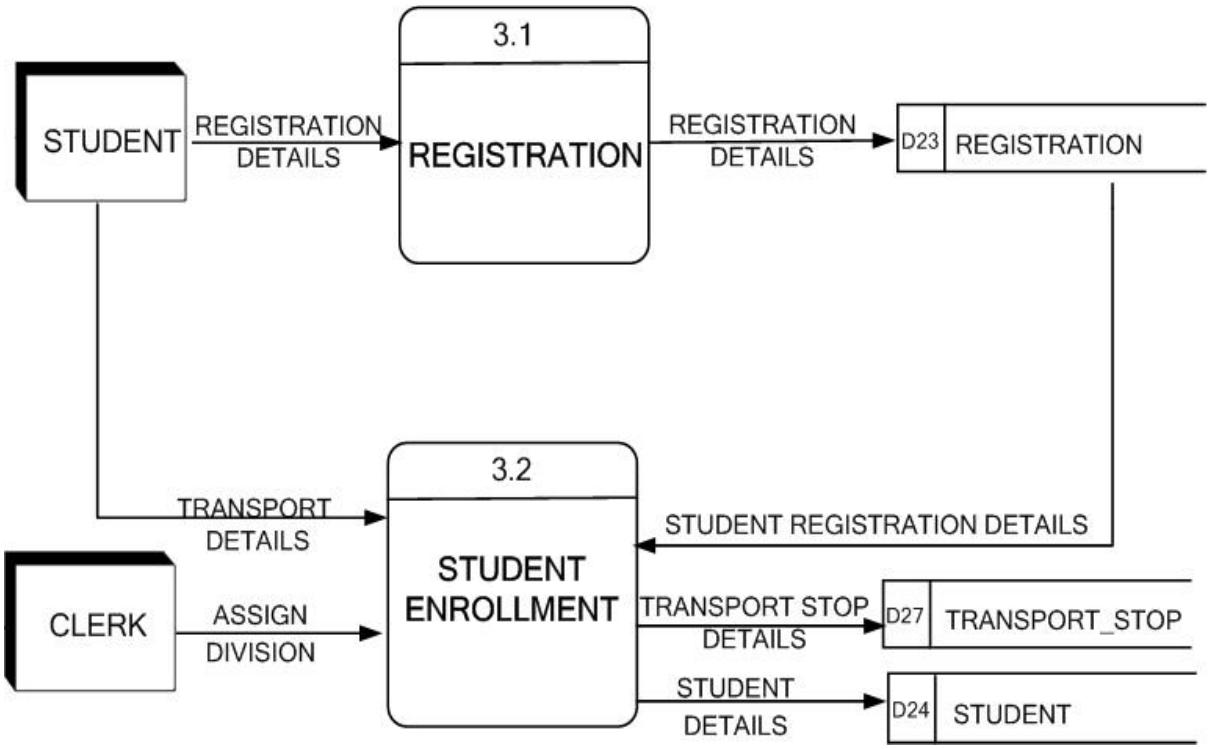


The following process is Admission process at 1st level of DFD, which we have further decomposed at 2nd level, into Registration, and Student enrolment processes.

Admission (1st – LEVEL):

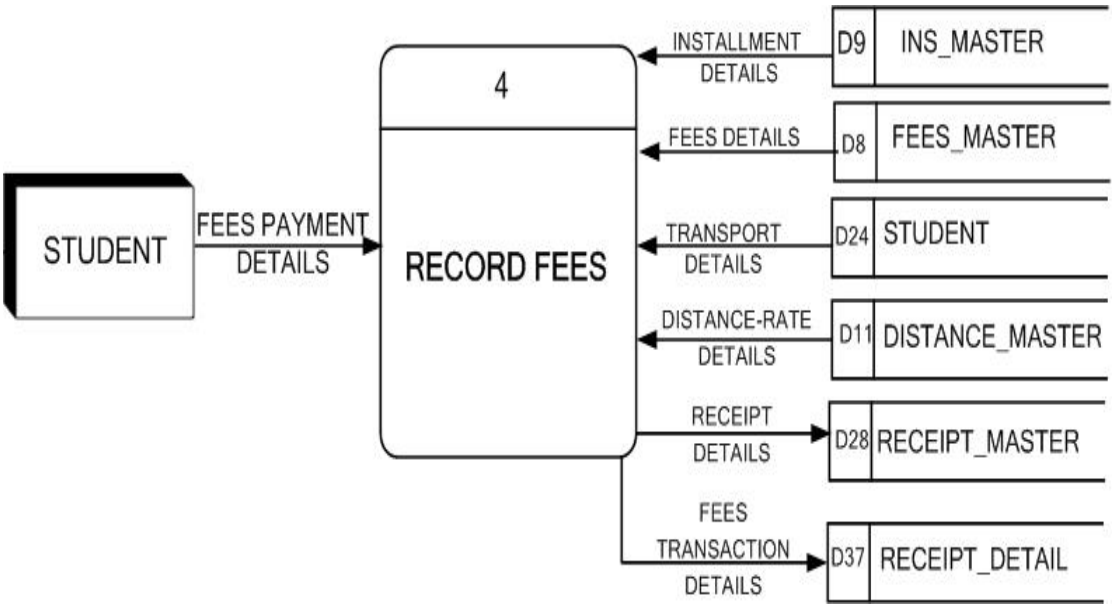


Admission Expansion (2nd – LEVEL):

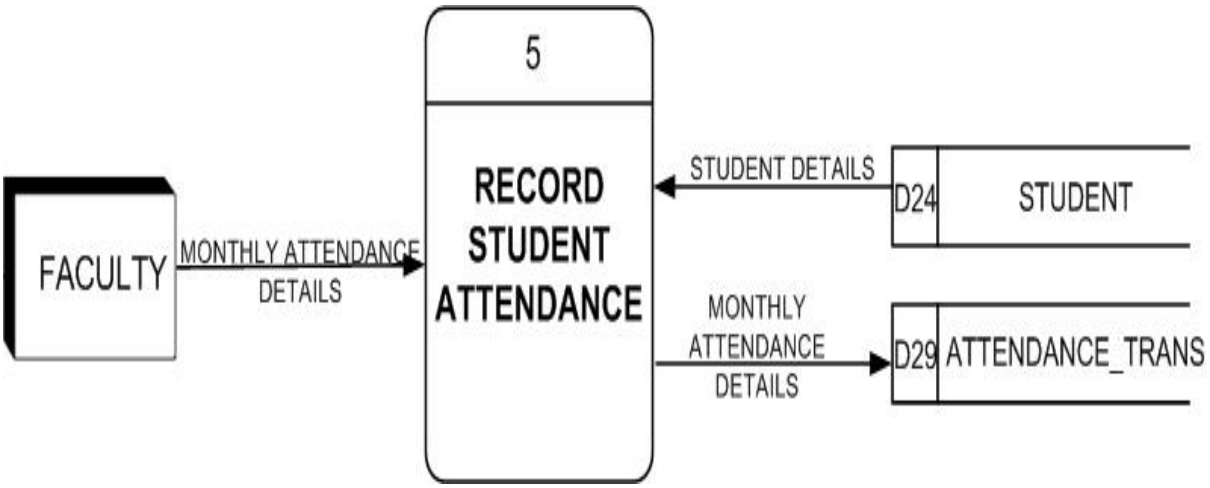


Another two processes are Record attendance, and Record fees. The Process is simple enough and no further expansion in 2nd level DFD is required. So, we keep this process at 1st level only.

Record Fees (1st – LEVEL):

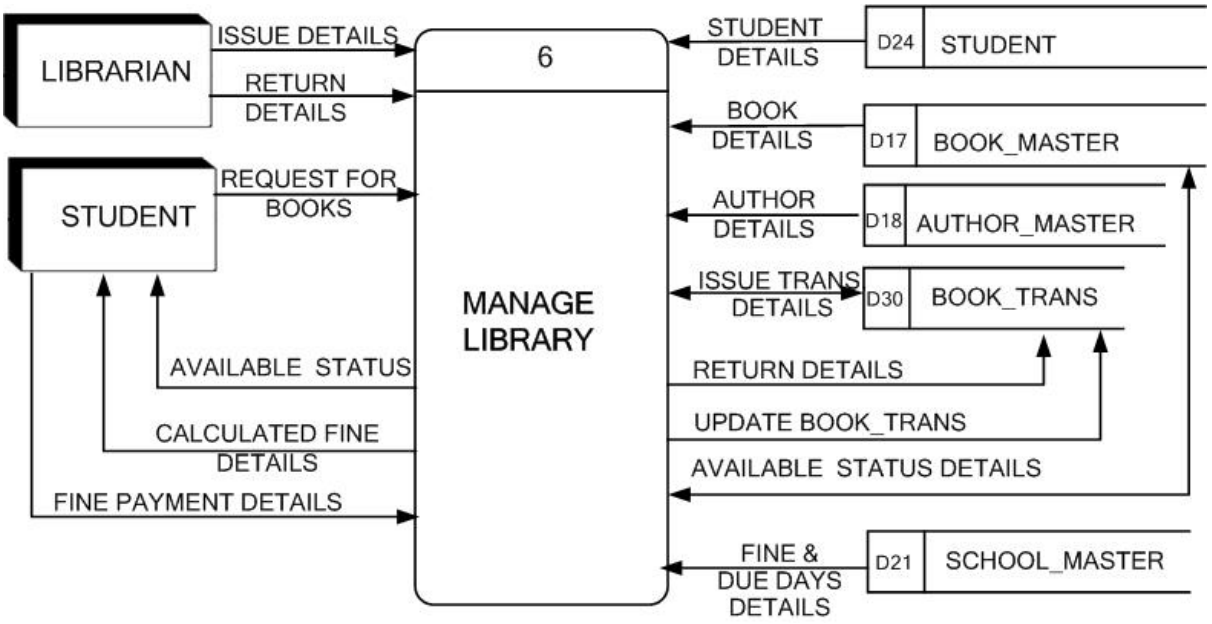


Record Student Attendance (1st – LEVEL):

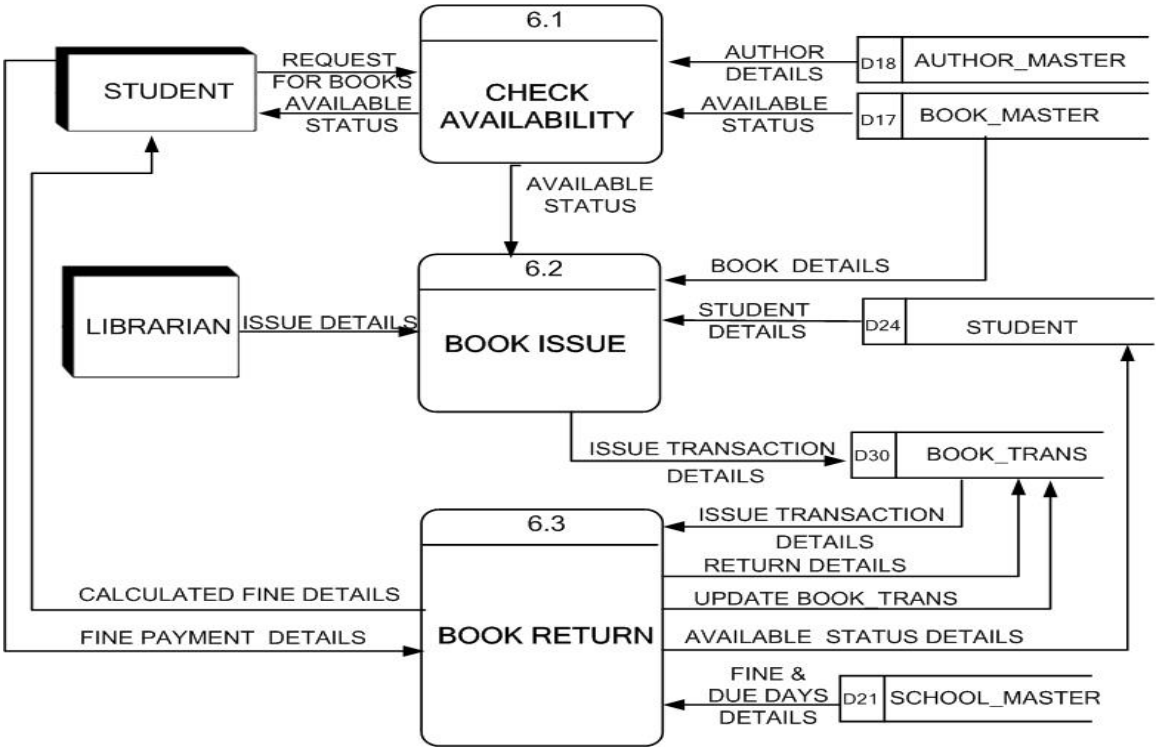


Next process in the sequence is 'Manage Library'. Because this process is complex, it requires to be decomposed into 'Check availability', 'Book Issue' and 'Book Return' sub-processes in the 2nd level of DFD.

Manage Library (1st – LEVEL):

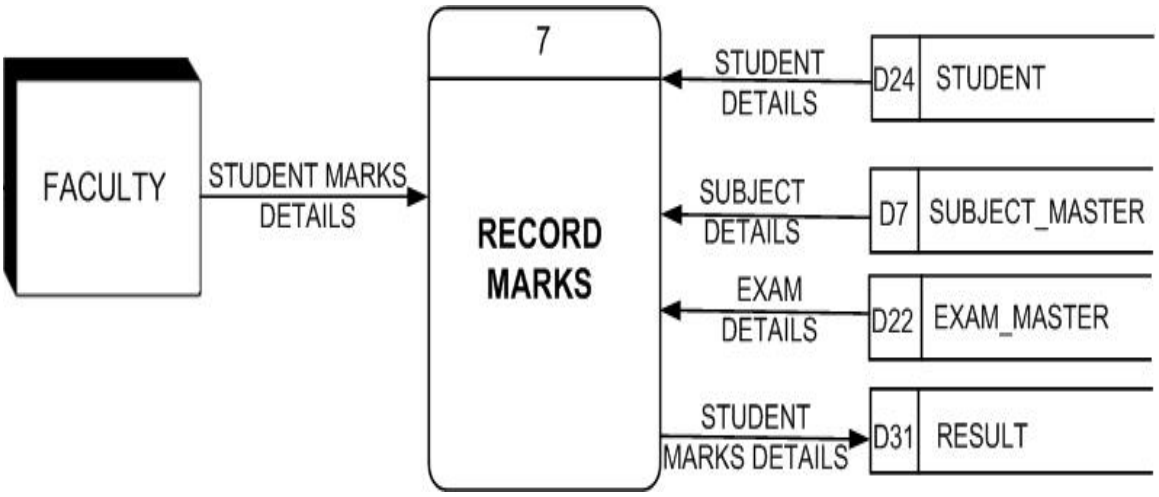


Expansion of Manage Library (2nd - LEVEL):



Next process in the 1st level, is to ‘Record Marks’ in which, Faculty members of the school will be entered the marks obtained by the students in the examination. Here faculty members need to refer the Student, Subject and Exam data tables and marks will be recorded in the Result table.

Record Marks (1st – LEVEL):



In the same way, one process for ‘making schedule’ of the exam and class, and one more process for ‘producing reports’ can be made. We hope by studying this Data-Flow Diagram (DFD), students will have sufficient knowledge of how to prepare DFD. In our example demonstrated above, we have followed Gane & Sarson Notations. But if you want to use Nordan notations too. Make sure, you are not allowed to use mixed notations in your Data-Flow Diagram.

7.6 ENTITY RELATIONSHIP DIAGRAM

Data-Flow Diagram represents the black-box view of the front-end system. It represents the application or interface part of the system. To denotes the back-end part (Database) in the structured analysis of the system, we use Entity Relationship Diagram (ERD). ER-diagram is detailed logical representation of the data for an organization. ERD represents data-oriented model, while DFD denotes function-

oriented model of the system. ER-diagram is used to represent data while DFD is used to represents flow of the data.

Before going into the detail of how an ER-diagram can be organized or prepared, we will discuss several terms related to the ER-diagram, which makes ERD learning process much easier.

1. **Data:** Data is unstructured row material and unstructured facts, which provides necessary input to the computer system.
2. **Attributes:** Attributes are characteristics or properties of an entity. It is represented by an oval symbol in the ER-diagram.
3. **Entity:** It is an important elementary item of an organization about which data is to be maintained. In the ER-diagram entities are represented by rectangular box.
4. **Data table / Entity set:** In the Database, data having similar attributes are stored as single unit called data table. Data table is composed of multiple rows (tuples). Each tuple or row of the data table represent one entity, so and so forth data tables are also known as entity sets. For example, Customer is an entity, where Customer code, name, address, date of birth, email are attributes. When data is entered in the table like (1, “Shiv Narayan Joshi”, “45, Vrindavan society”, 22/08/1976, shivnarayan@gmail.com), then it is called record. Each value in the record is called property and Shiv Narayan Joshi is the entity. Because we can store multiple records (rows or tuples) in a datable table it is also called an entity set. In the Relational Database Management System (RDBMS) data table can be an entity or relation between two entities.
5. **Master Table:** The term is used for those types of data tables which stores data of the permanent type of nature. Master tables are those tables, in which data are used frequently but updated or changed (insert, update, delete) very rarely. For example, “Employee” table is a master table, which store information like Employee code, name, address, date of birth, email, mobile number etc.
6. **Transaction Table:** Transaction tables those database tables, which record those data which are not of permanent type, it is generally useful to store day to day transactions. Transaction tables are those which are used rarely but update frequently. For example, “Salary” table where we store all particulars of the salary given to employee every month are stored.

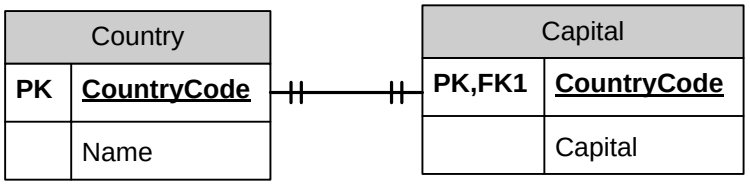
7. **Primary Key:** Primary key is a key column of the data table which is uniquely identifying each row of the table. Once the primary key is given on any particular field or column of the table then it will not accept duplicate or null value.
8. **Foreign key:** Foreign key is a key column which must be primary key of another table or relation.
9. **Composite key:** When the primary key given on more than one field is called composite key. Here two or more fields collectively, uniquely identify each and every tuple (row) in the database table.
10. **Relationship:** In RDBMS, entities are connected to each other by relationships. It represents how two entities are associated with each other. In ER-diagram a diamond notation is used to represent relationship, and name of that relation is specified in the diamond. Number of entity types which are participated in relationship is called degree of the relationship.
11. **Cardinality:** The cardinality simply represents the relationship between two entities. If we focus on the relationship between state and city, then it is one-to-many relation, as there are many cities in one state. Here, cardinality of a relation is the number of instances of entity city that can be associated with each instance of entity state. In the situation where there can be no instance of second entity, then it is called optional relation.

7.7 CARDINALITY OF RELATIONSHIPS

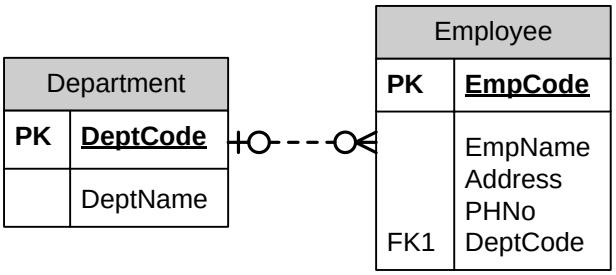
There are three types of relationships are there:

1. One-to-One (1: 1)
2. One-to-Many (1: M)
3. Many-to-Many (M: N)

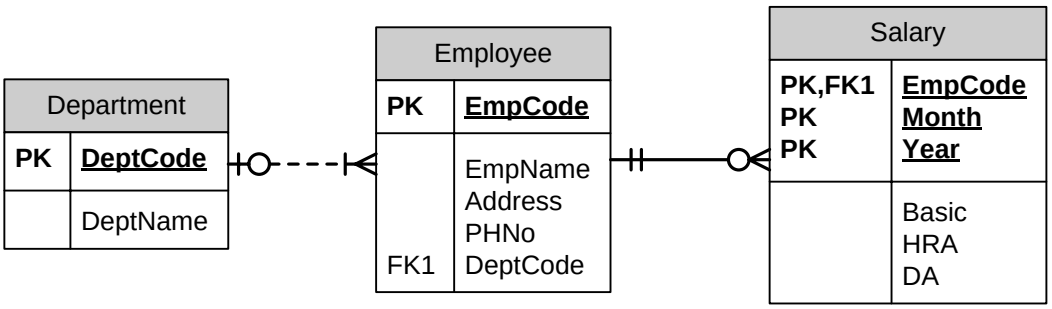
A **One-to-One or 1:1** cardinality exists when exactly one entity of first entity set relate with maximum one entity of the other entity set. For example, if we consider county as first entity and capital to be a second entity then, the relation or cardinality is one-to-one, which means that one country has only one capital.



A **One-to-Many or 1: M** exists when one entity of first entity set relates with two or more entities of another entity set. For example, one department of an organization has many employees. In this example DeptCode is common attribute. DeptCode is primary key in Department table and foreign key in the Employee table.

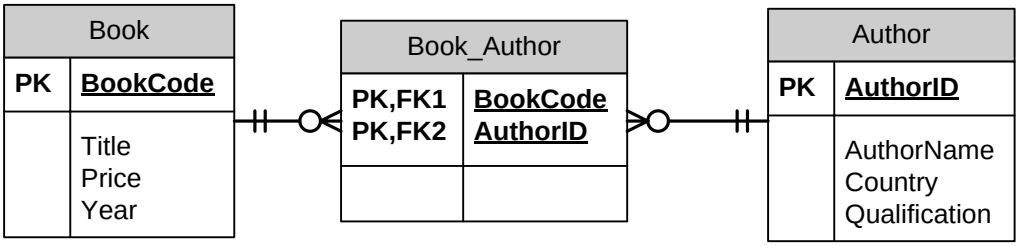


In this example cardinality is one department has zero or more employees. In the above diagram relationship with dotted line indicates weak entity relationship. Consider another one-to-many relation between Employee and Salary. All employees get salary every month (one-to-many). Here 'EmpCode' is a common attribute between Employee and Salary tables. EmpCode is primary key in Employee table and foreign key in the Salary table, not only that but in the Salary table EmpCode, Month, Year fields are act as a composite primary key. That means that duplication in these three fields is not allowed. Either EmpCode or Month or Year any one attribute has to be differed (One employee in the same month and year cannot get more than one salary).



A **Many-to-Many relationship (M: N)** exists when one entity of first entity set can relate with many entities of second entity set, as well as one entity of second entity set can also relate with many entities of the first entity set then it is called as many-to-

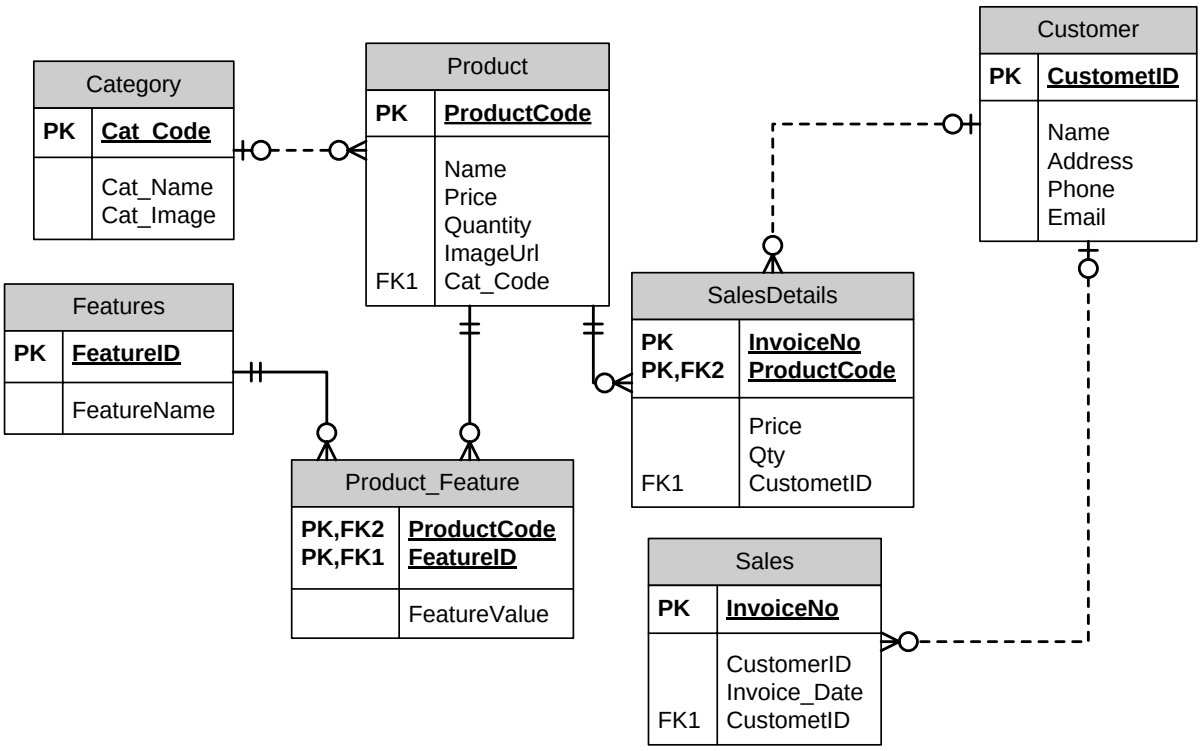
many cardinality. For example, Book and Author is the example of many-to-many relationship. One author can write many books as well as it is also possible that one book may be written by more than one authors.



In the one-to-one cardinality we can merge two tables into one. For example, in the example of Country and Capital, both tables can be merged into one i.e., Country (CountryCode, CountryName, Capital). In the one-to-many cardinality two tables are needed. Similarly, in many-to-many relation three tables should be there, in which two tables represents entities and third table act as bridge table.

7.8 EXAMPLE OF ER-DIAGRAM

Consider an application of ‘Online Sales’. Company sales different category of products. One category should have many products. One product should have many features. Similarly, one feature should be there in the many products. Customer can place many orders and also one sales order can have multiple products. We are limiting our discussion up to here to make the example easier, simpler and having less complexity. Readers of this book can include supplier and purchase details, product feedback, rating, sales and purchase return etc. as per their own assumptions and knowledge.



Exercise:1 Fill in the blanks

1. Primary keys applied on more than one field is called _____.
2. Relationship between Country and State is _____.
3. Types of the relationships are _____, _____, and _____.
4. In the DFD, Entity can be denoted by _____ symbol.
5. _____ symbol is used to denote process in the DFD in Yourdon notation.
6. In the ER-Diagram attributes can be represented by _____.

7.9 Let us sum up

In this unit we have seen that requirement analysis can be done in two ways. (1) Structured analysis and (2) Object-oriented analysis. In the structured analysis, an engineer depicts data flow diagram to represent the flow of the data, and Entity Relationship diagram to represent database structure. We hope, after learning this unit student can prepare ER-diagram and DFD of any given system.

7.10 Check your progress: Possible Answers

Exercise: 1

1. Composite key
2. One-to-Many
3. One-to-One, One-to-Many, Many-to-Many
4. rectangle
5. Circle
6. oval

7.11 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

7.12 Activities

1. Draw a data flow diagram of 'Online Sales Management system'.
2. Prepare ER-Diagram for school management system. Make suitable assumptions if required.

Unit 8: Object Oriented Analysis and Design

8

Unit Structure

8.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know how software system can be analyse with object-oriented approach
- Understand object-oriented analysis related basic terms
- Understand UML diagram of object-oriented analysis and design
- How to draw different types of UML diagrams?
- Know how the quality software product can be determined

8.2 INTRODUCTION

In the previous unit we have seen that an engineer can do the analysis of the system using any of the method from (1) Structured analysis or (2) Object-oriented analysis. In the previous unit, we have seen that, if structured analysis is chosen then data flow diagram (DFD) and entity relationship diagram (ER diagram) are the basic tools. In this unit we will focus on object-oriented analysis.

Object-oriented analysis and design are extremely popular now a days. In this section we will emphasis on some basic ideas of object-oriented analysis. We will emphasis on UML (Unified Modeling Language) which can be understood as a standard for object-oriented systems. Before we start doing object-oriented analysis, and learn how to design UML diagrams, first we will give attention on some important concepts of object-oriented system.

8.3 BASIC TERMS OF OBJECT-ORIENTED ANALYSIS

OBJECT:

Object is a tangible or physical entity of the real world. Generally, object can have its own data, which is known as property of the object, and in the same way object can also perform some actions, which known as methods of an object. Normally, one object cannot have privileges to access the data of another object. Object itself can have privilege to use its own private data. Object can be anything, any real entity which has something (called as properties or attributes) and it can do something (called as methods). For example, person XYZ is a customer of ABC Bank is an object.

Customer Identification number (CID), Name, Account No, Balance in the account are the properties of the customer object, where the customer can perform some actions, like withdraw the money or deposit the money are the methods of the customer. Similar to customer, teachers, students, accountants are some of the examples of object.

CLASS:

Similar objects collectively create a class. Those objects having similar properties and possessing similar behavior create a class. Class can be thought like a template, which represents multiple objects. Class has variables (properties or attributes) and functions (methods). Object is an instance of the class. When the object is created or initiated, from the class then variables declared in the class will become properties of the object and functions specified in the class will becomes methods of that particular object. Classes are also known as Abstract Data Types (ADTs).

ABSTRACTION:

Abstraction is the process of identifying characteristics and methods of an object. Abstraction is the discriminatory examination of certain features of a problem while ignoring the remaining aspects of the problem. In short, we can say that, abstraction is the process by which we concentrate on those aspects of the problem which are relevant and ignore those aspects which are not relevant.

ENCAPSULATION:

Encapsulation is a method in which variables (attributes) of the class and functions (methods) are club or packaged together as a single unit, and data can be accessible by the outsiders through its methods only. Encapsulation provides black-box view to the class, where non-member functions of the class (outsiders) are not permitted to access the data directly. Generally, in the class variables (attributes or data) are declared in the private sections, so that only member functions of that class (those methods written in the class) can access it. If any other function (non-member function) wants to access it, they need to call any member function (method of the class), which verify the request is proper or not. If the request is proper and valid then and then the data will be provided by the method to the requested non-member function. So essentially, encapsulation enforces security to the data elements of the class from the outside world.

POLYMORPHISM:

Polymorphism is Greek word, which means more forms of the same thing. An operation may show different behaviors in different instances. Features like function overloading and operator overloading in an object-oriented programming language is type of Polymorphism. If two or more functions with same name, can be separated by a system at execution time by looking to number of arguments passed to it or datatype of arguments passed. In function overloading, we have different functions but having same name, which is an example of polymorphism. Polymorphism is increasing readability of the program. For example, if we have two functions to do sum, first function to do sum of integer numbers and another function to do sum of float numbers. Because of the nature of the function is same (computing sum) we can give same name 'sum' to both the functions in object-oriented programming languages like Java, C#, C++ etc., and we do not have to choose different names like sum and sum1 to different functions like in C-language.

INHERITANCE:

Inheritance is the process by which developer (programmer) can derive a new class from the existing class (es). Existing class will act as parent or base class, and newly generated class from the parent class is called child or derived class. Derived class receives properties and methods automatically from its base class, so we do not have to repeatedly define those properties and methods which are already present in the base class. Inheritance offers reusability of the code (Code specified in the parent class can be accessible by the child class).

8.4 OBJECT ORIENTED ANALYSIS AND DESIGN

Object-oriented analysis:

Object-oriented analysis is focusing on examine at the problem domain, with the main purpose of producing a conceptual model of the information gathered in the preliminary investigation, feasibility study and requirement gathering and analysis phases of the SDLC, are being studied. Analysis model do not concentrate on implementation part, or how system is to be prepared. Analysis has to be performed before the design.

Written problem statement or formal vision document is the source of the analysis. A system may have divided into multiple domains, representing different business, technologies, or other interest areas. The result of the object-oriented analysis is functional requirements in the conceptual model.

Object-oriented design:

If we consider analysis to be a definition of the problem, then obviously design is the process of defining the solution of the problem. Object-oriented design process describes the components like classes, objects, properties, methods and interfaces which fulfills functional requirement of the system. OOD converts the conceptual level model produced by analysis into the technical or environmental model.

8.5 UML DIAGRAMS:

Unified Modeling Language (UML) consists of different types of diagrams which represent object-oriented analysis. Each diagram focuses on different aspect to describe and examine the system. These diagrams are:

1. **Class diagram:** Class diagram is used represents different classes of the system and relationships among them. Class consists of properties and methods.
2. **Object diagrams:** Objects are the instances of the class. Object diagram represents the relationships among the various system objects. Single class-diagram can have multiple object diagrams.
3. **Use-case diagram:** Similar to DFD in structured analysis, use-case represents external behavior in object-oriented analysis of the system. A use-case diagram consists of several actions known as use-cases, and their interactions with different actors (people).
4. **Sequence diagram:** Sequence diagram represents exchanges between actors (users) over time period. A sequence diagram is detail behavior with respect to the time of each use-case (action) of use-case diagram. That means for each case denoted in the use-case diagram, a separate sequence diagram has to be designed.

5. **Collaboration diagram:** Collaboration diagram represents the exchanges of objects of the system with respect to the relationships among the objects.
6. **State chart diagram:** State chart diagram represents various states of the software system in response to the different events triggered by the user. A state chart diagram shows, how the state of the system changes in response to any internal or external events.
7. **Activity diagram:** Activity diagram represents explanation of the behavior of the system. Activity diagram shows details performance of the single function.

8.6 USE CASE DIAGRAM

Use-case diagram provides a fast and simple and efficient way to describe the purpose of the project. Recently many software engineers are using it for their ongoing projects, to record high-level purposes of the project in its early phase of development.

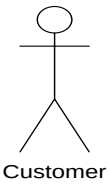
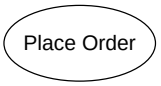
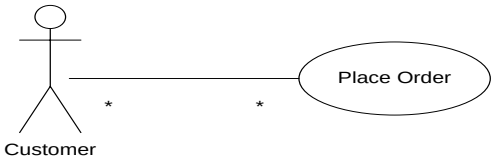
Use-case diagram is used to recognize different processes (use-cases) as well as primary elements (actors) of the system. The primary elements are also identified as *actors* and processes of the system are identified as *use-cases*. Use-case diagram represents how the different actors of the system are interacting with the different use-cases of the system.

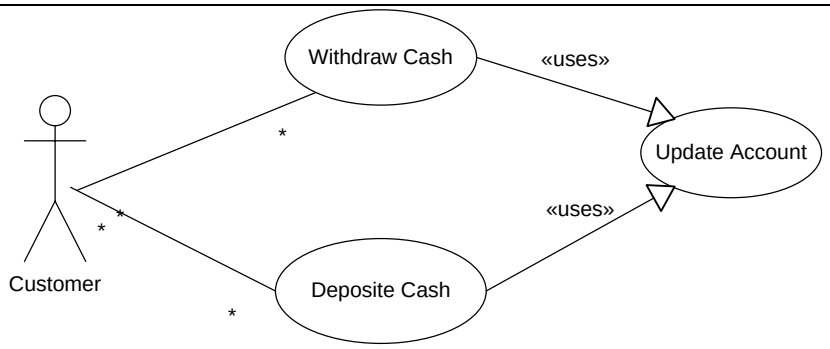
Use-case diagram concentrates on functional requirements of the system. It represents the graphical vision of the system functions (use-cases) and users (actors).

ELEMENTS OF USE-CASE DIAGRAM:

It is easier to draw use-case diagram, if you have proper knowledge of its different components. The components of the use-case diagrams are:

1. Actors
2. Use-Case
3. Relationship between Actor and Use-case
4. Relationship between Use-cases
5. Relationship between Actors
6. System boundary.

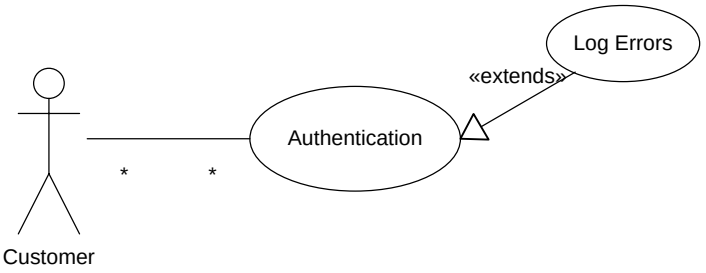
Sr. No.	Symbol	Description
1		**Actors:** An actor can be a user or role (group of users) which directly interact with the system by invoking different use-cases (functions) of the system. Generally, Actor can be human, a hardware device, any department or another system which operates or communicate with the system. Actors are always external to the system (outside of system boundary). Actor can provide data or obtain information from the system by interacting with different use-cases in use-case diagram.
2		**Use-Cases:** Use-Cases in the Use-Case diagram, represents set of sequences of actions that system performs. A use-case describes what a system, sub-system, class or interface does, but not describe how it does.
3	**Relationship between Actor and Use-cases:** Relationship between the Actor and Use-case is communication between the instance of Actor and instance of the Use-case. It is demonstrated by a straight line between the contributing Actor and requested Use-case.  Customer place order	
4	**Relationship between Use-cases:** There are two types of relationship are there between Use-Cases: [1] Uses / Includes: A Uses / Include relationship between two use-cases describe that the sequence of actions described in the sub use-case is included in the sequence of the base use-case. A sub use-case is called included and base use-case is called including use-cases. For example, customer can 'Withdraw' or 'Deposit' money in cash, but in both the cases account has to be reflected.	



Use-cases Withdraw Cash and Deposit Cash includes Use-case Update Account

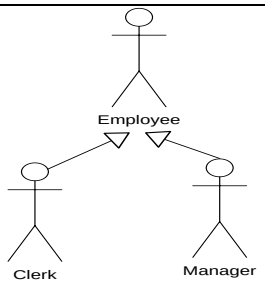
Note: Uses / Includes enforce compulsion. In the case of withdraw or deposit account must be compulsorily (mandatorily) updated.

[2] Extends: Extend is normally used to extend the functionality if one Use-case by another. For example, in the case of authentication process, if any error occurs then, it must be logged into log files. Make sure here the sub use-case will perform if error occurs. If the process of Authentication done positively then sub use-case 'log error' should not performed. Thus, extended Use-cases represents optional activities.



Use-case 'Authentication' extends sub use-case 'Log Errors'

5	**Relationships between Actors:** Some time different Actors of the system has to be generalized into one generalized Actor. It is useful while the roles of different actors are overlapped.

	 Generalization
6	**System Boundary:** System boundary is a rectangular box, which denote whole system. All the Use-cases has to be placed inside this rectangle box, to denote Use-cases are in the system. Actors are the external entities, so they are placed outside of the rectangle (system).

8.7 CLASS DIAGRAMS

Depending upon the purpose, class diagram can be planned of two ways: (1) Analysis oriented class-diagram or (2) Design oriented class-diagram. Analysis model provides just overview of the class, that is names of the attributes and functions. Whereas, Design model represents attributes with its data type, and functions with arguments and its return type. So, Design oriented class-diagram is more detailed version of the class diagram.

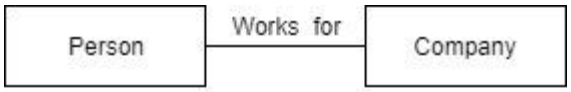
Analysis Class Diagram	Design Class Diagram
Order -OrderID -OrderDate -DeliveryDate +CalculateTotal() +CalculateGST()	**Order** -OrderID : Integer -OrderDate : Date -DeliveryDate : Date +CaculateTotal() : Single +CalulateGST(in Total : Integer) : Single

ELEMENTS OF CLASS DIAGRAM:

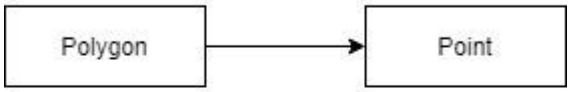
1. **Class:** As depicted in the figure given above, in a class-diagram, class has to be represented in rectangular shape divided into three sections. First section represents the name of the class. Second section lists all the attributes of the class and third section list all the methods of the class.
2. **Relationships:** One class can relate with another class in the class-diagram, with different types of relationships. The types of relationships in the class diagram are discussed below with examples:

[A] Association:

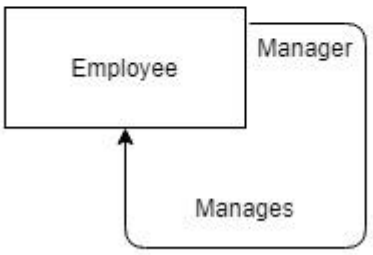
Conceptual or physical connections between the classes can be demonstrated as association (simple line). In the class-diagram association relationship should be labeled as verb. For example, teaches, work for, manages etc. can be valid names for the association relationship. Association relationship can be unidirectional, bidirectional or reflexive.



Bidirectional Association



Unidirectional Association



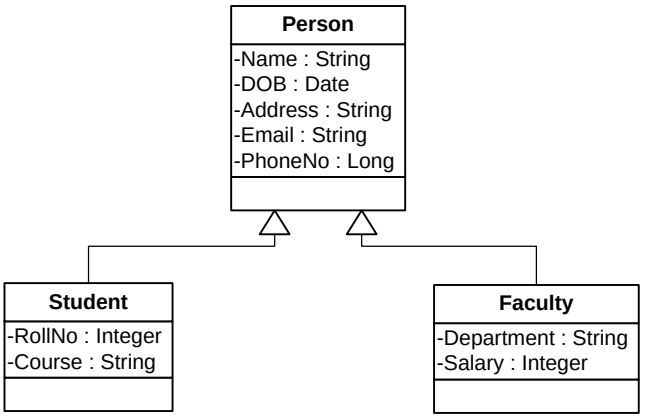
Reflexive Association

[B] Generalization:

Generalization is the process of generating base class, If two or more classes have common attributes or methods. Common attributes and method are placed in the

generalized class (super or base class). For example, if student class has attributes like, Name, Date of Birth, Address, Email, Phone number, Roll No, course etc. and Faculty class has attributes like, Name, Date of Birth, Address, Phone number, Department, Salary etc. Then generalized class Person can be formed with common attribute. Here, base class (Person) and derived classes (Student, Faculty) have 'is a' type of relationship. For example, each student or each faculty is a person. Generalization can be represented by symbol:

Generalization relationship symbol

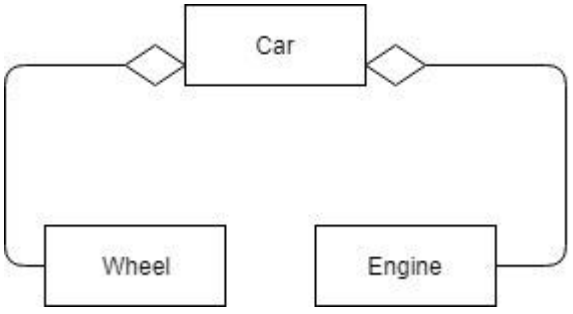


Generalization relationship in class-diagram

[C] Aggregation:

The relationship between aggregated object and its components can be described as aggregation. Aggregation is a kind of association. Aggregation can be represented by symbol:

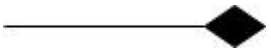
Aggregation relationship



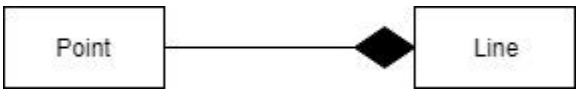
Example of Aggregation relationship (Car made by While Engine etc.)

[D] Composition:

When multiple instances of the same class represent another class then composition is used. The difference between Aggregation and Composition is in aggregation multiple instances of different class represent another class, whereas in composition multiple instances of same class. It is represented by:



Composition Relation



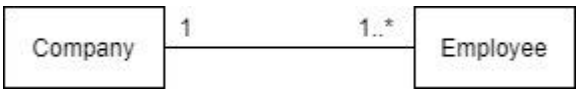
Example of Composition

[E] Multiplicity:

Multiplicity notation is placed near ends of the relationship. It shows how the instances of one class are linked instances of another class.

Indicator	***Meaning***
0..1	Zero or one
1	One only
0..*	Zero or more
*	Zero or more
1..*	One or more
3	Three only
0..5	Zero to Five
5..15	Five to Fifteen
2,4	Two or Four

For example, one company can have one or more employees can be represented as:



8.8 SEQUENCE DIAGRAM

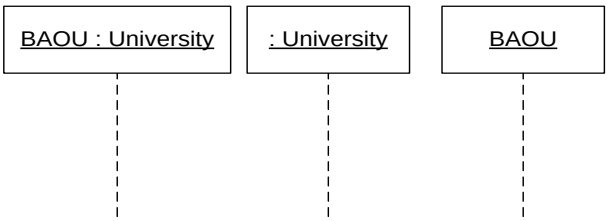
A sequence diagram is used to express each Use-case in details with respect to time. A sequence diagram represents the sequence of actions occurs in a Use-case, and order of each action with respect to time.

ELEMENTS OF SEQUNCE DIAGRAM:

The following elements are used to draw the sequence diagram.

Life Lines:

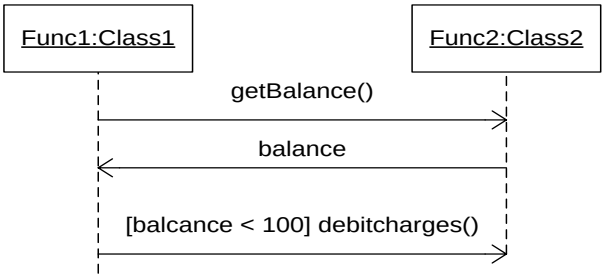
Lifeline represents role or instances which participate in the sequence of interactions. Lifelines are drawn as a rectangle with a dashed vertical line from the center of the rectangle. Inside the rectangle name of the class, name of the instance or both can be specified.



Lifeline of the Sequence diagram

Messages:

Message defines a kind of integration between instances (Actor or Lifeline). Communication (message passing) can invoke by mean of function calls. It is shown in the following figure:

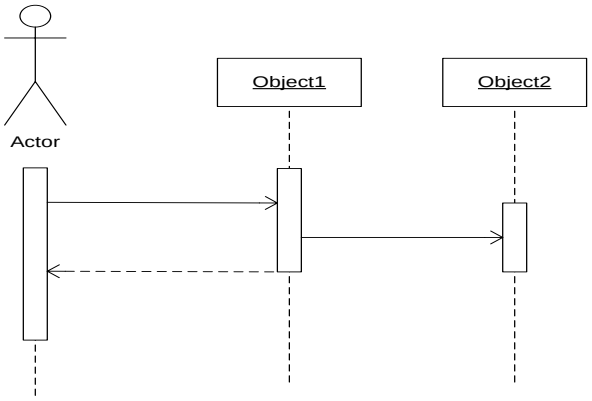


Message passing between Lifelines

The condition placed in the sequence diagram, that is the balance is less than 100 the call function 'debitcharges()' is called guard condition.

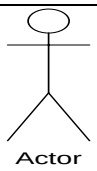
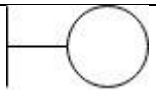
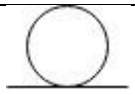
Activation:

Activation is represented as vertical thin boxes on the dotted lines of the Lifelines, which represents the time an object takes to complete the task. Following diagram shows the activation.



Objects:

There are four different types of objects are the, who interact with each other in the sequence diagram. All these objects are described below:

Actor		Actor object initiate the task. Actor is an instance of the class and it is external entity. The role of the actor is same as in Use-case diagram.
Boundary		Instance of the boundary class is used to model the communication between system and external objects like Actor.
Controller		Controller is used to control the behavior specific use-case. It represents logic. Generally, it comes between boundary and entity.
Entity		Entity object is used to store the associated behavior or model information. It represents stores of information in the system.

8.9 Analysis Modeling

Analysis modeling can be organized by it four elements – scenario-based modeling, flow-oriented modeling, class based modeling and behavioural modeling.

1. **Scenario based modeling:** In the scenario-based model, we draw the use case diagram, Activity diagram and Swimlane diagram.

   Use-Case: Use case is used to represent the various scenarios by the user's (Actor) point of view. Use-case diagram is a simple and relatively easier approach to represent what is outside of the system (like Actors) and what system should be performed (use-cases). We have already discussed how to prepare the use-case diagram in section 8.6

   Activity Diagram: In the activity diagram we concentrate on main tasks or functions (use-cases) of the use-case diagram, and represents what Actor can obtains, produces or change in the system. Detailed communications of the different users of the system (Actors) and tasks of the system (use-cases) can be represented by activity diagrams.

   Swimlane Diagram: It is nothing but the useful variation in the activity diagram and allows the modeler to represent the flow of activities described by the user-case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.

   Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

2. **Flow-oriented modeling:** Flow-oriented modeling represents flow of the data in the system. It Represents how data objects are transformed as they move through the system. We have already discussed Data Flow Diagram (DFD), which shows the transitions of the data in the system. To draw the DFD, we need to identify Entity, Process, Data stores and transition of the data among them are represented by arrow. In the Unit:6, we have already discussed the notations and rules of how to draw DFD.

3. **Class-based modeling:** Class based modeling is also known as Object-Oriented Analysis. We have seen in that object-oriented analysis begins by identifying classes. Once the classes are recognized then its attributes and methods are identified. Classes are represented with their relations with the other classes. In this process basic fundamentals of the object-oriented analysis such as Abstraction, Encapsulation, Polymorphism and Inheritance is used. The elements of the class diagram are already discussed in the section 8.7.

4. **Behavioral Model:** It indicate how system will behave or respond to the event triggered by the external entities or actors of the system. To create the model, the analyst must perform the following steps:

    1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
    2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
    3. Create a sequence for each use-case.
    4. Build a state diagram for the system.
    5. Review the behavioral model to verify accuracy and consistency.

To represents the various changing states of the system, state chart diagram is used.

The States of a System

- State—a set of observable circumstances that characterizes the behavior of a system at a given time
- State transition—the movement from one state to another
- Event—an occurrence that causes the system to exhibit some predictable form of behavior
- Action—process that occurs as a consequence of making a transition

Exercise:1 Fill in the blanks

1. In the Use-case diagram user is called _____.
2. _____ diagram is used to show changes in the state of the system.
3. In the Sequence diagram _____ is the interface between user and system.
4. In the class diagram if the construction of the object made by the instance if different classes then _____ and if construction is done by the multiple instances of the same class the _____ is used.
5. In the class diagram process of making new class from two or more classes having common attributes is called_____.
6. In the sequence diagram is message is passed on the basis of condition then condition is known as _____ condition.
7. In the sequence diagram validation is done by _____.

8.10 Let us sum up

In this chapter we have learnt how can we do object-oriented analysis and design using UML diagram. We have seen the rules, symbols and components of Use-

case diagram, Class diagram, Sequence diagram and so on. At the end we have seen the fundamental concept of designing good and quality system. We hope now student can draw UML diagrams of any system if clear and concise requirements are given.

8.11 Check your progress: Possible Answers

Exercise: 1

1. Actor
2. State-chart diagram
3. boundary
4. aggregation, composition
5. generalisation
6. guard
7. controller

3.12 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. Magnifying object-oriented analysis and design by Arpita Gopal and Netra Patil (PHI)
5. Object-oriented modeling and design by James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (PHI)
6. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

3.13 Activities

1. Draw UML diagrams for online library management system. Make suitable assumptions.

Unit 9: UML Diagram of System A Case Study

9

Unit Structure

9.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know how to prepare Use-Case diagram
- Understand how to draw sequence diagram
- Understand how to represent class and object diagrams
- How to draw activity and state-chart diagrams

9.2 INTRODUCTION

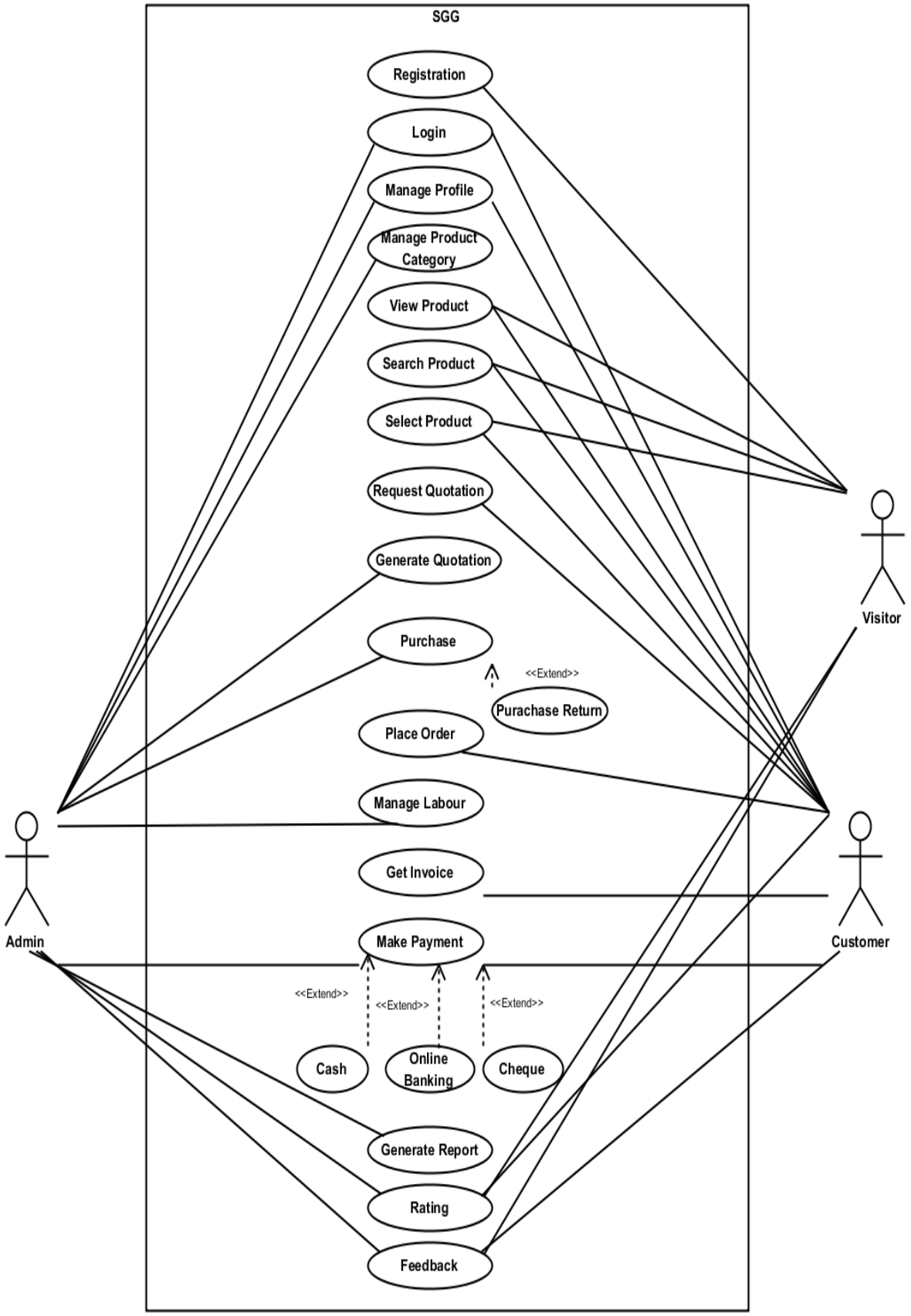
In the previous unit we have seen that how can a software engineer can do object-oriented analysis of the system and represent his/her work with the help of different UML diagrams. We have discussed the methodology and rules to develop different types of UML diagram. Now, in this unit we will try to draw all those diagrams by taking a hypothetical case study.

9.3 UML DIAGRAMS – A Case Study

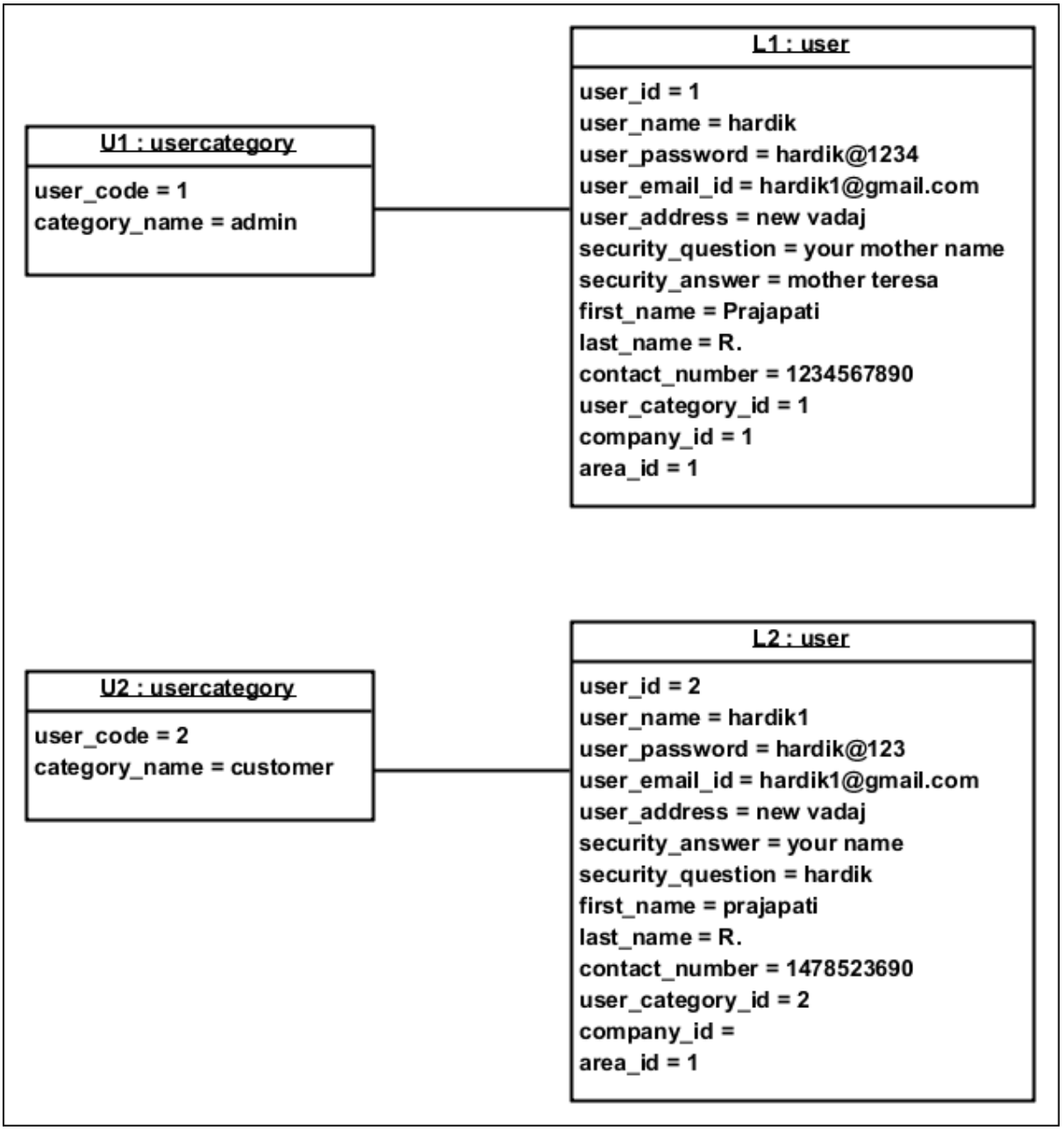
A Shri Shiv-Shakti Glass Traders (SSGT) is a glass-based product selling company, wants to develop website, so that the company can facilitates its customers, to select different types of glasses and place the order online. They also do the work related to partitioning of cabins in the hall using glasses and frames of aluminium. They need a web-based application, so that customer can specify their requirements, by viewing different types of glasses, their thickness, designs on the glass and prices. SSGT wants to send their quotes of the customer's requirements online through the new web-site to be develop. Customer should also place the order online through the website. Day-to-day the administrator of the website, should checks for the customers' requirements and if any requirement is there, administrator will prepare the quotation based on the requirements specified by the customer, and send it to the customer. Once payment is made then Invoice should be generated. Customer should log-in to the system and should also check past transactions with the company.

This online web-application also helps the company to manage their purchases, and purchase return transactions of different raw materials from the supplier. System has to produces different types of reports and helps company (SSGT) to manage schedule of the worker.

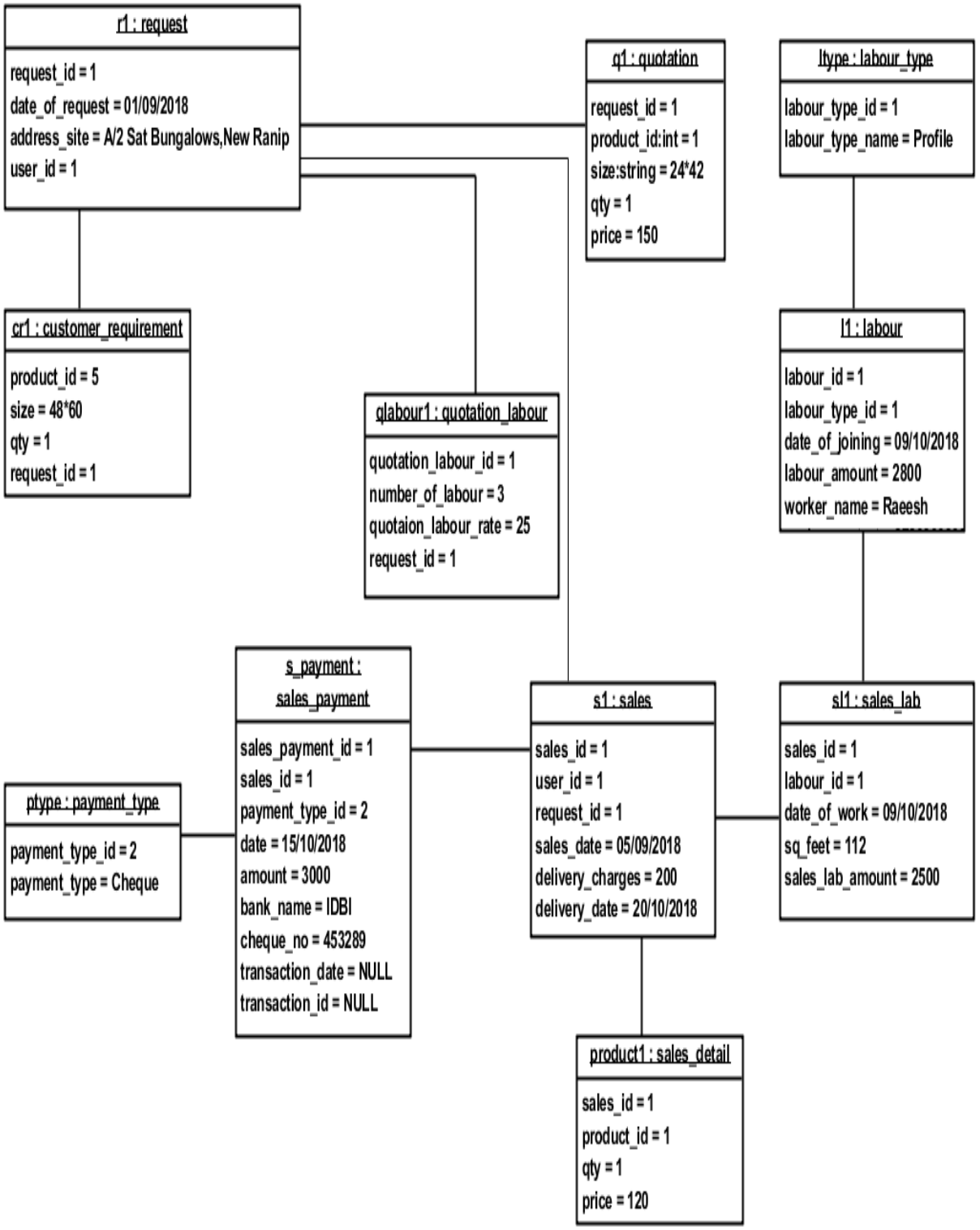
Draw the various UML diagrams for the system described above.



Use-Case diagram of the SSGT system



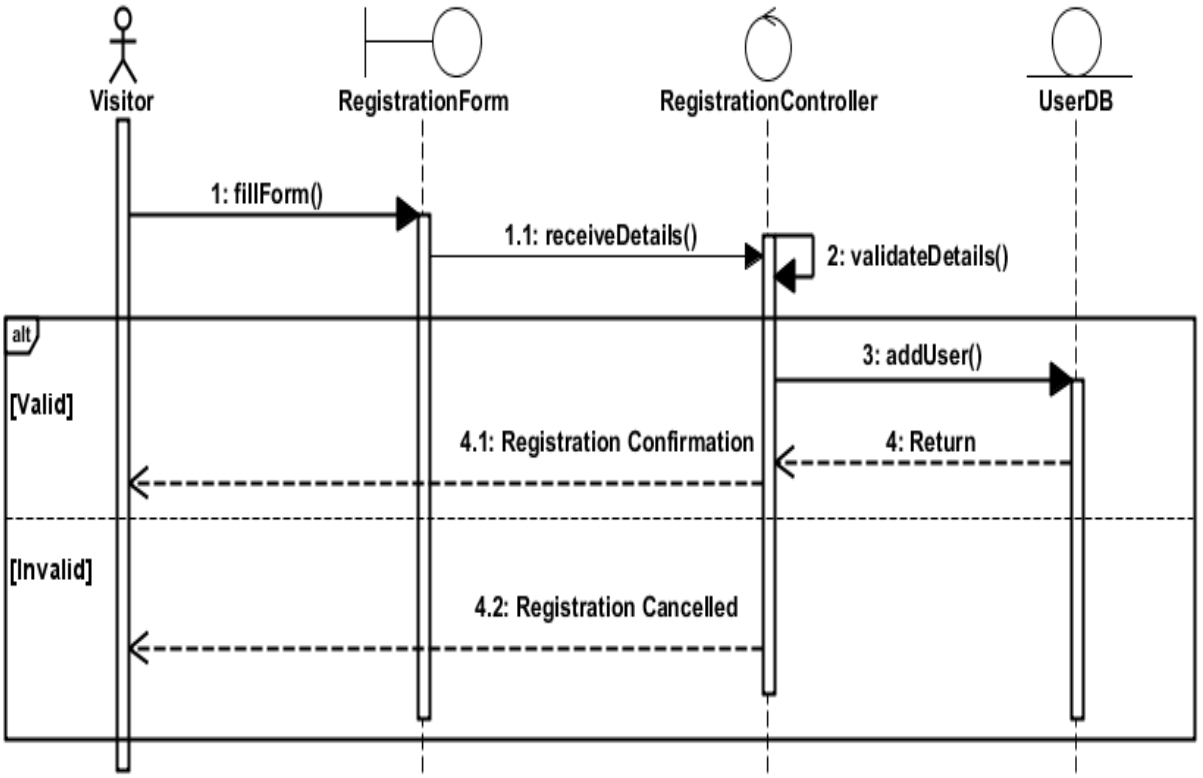
Object Diagram of Login and User Category



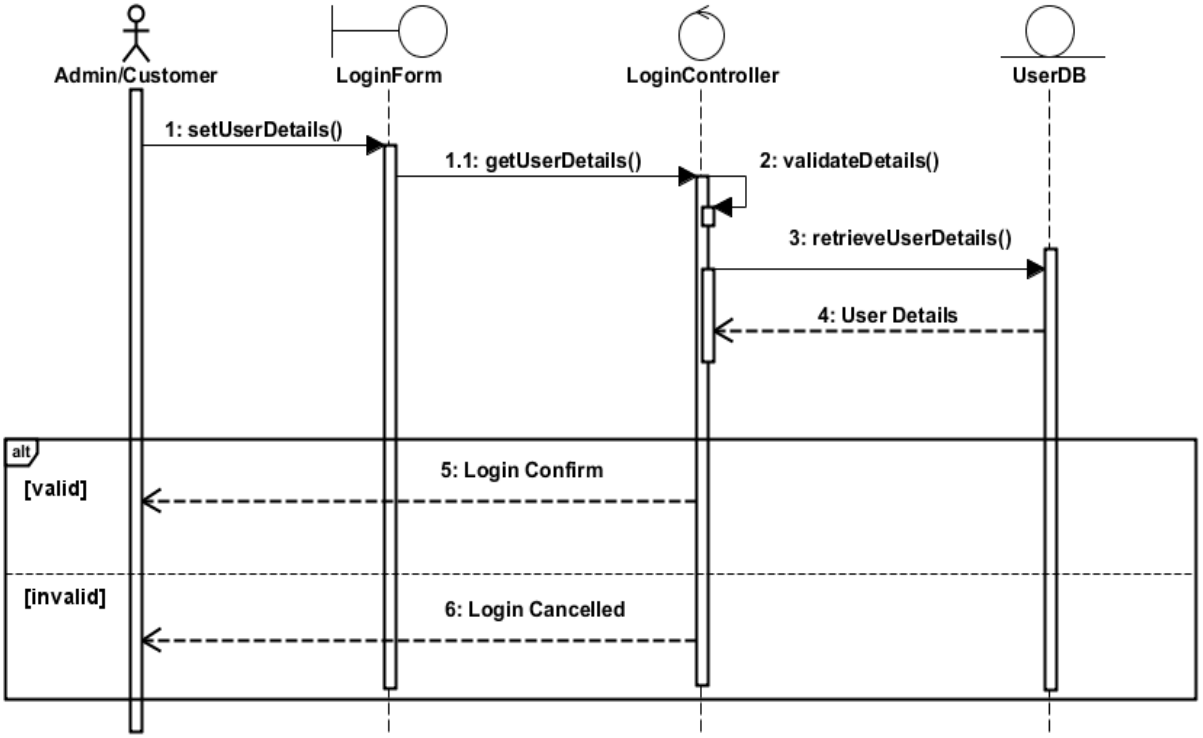
Object Diagram of Sales

SEQUENCE DIAGRAMS

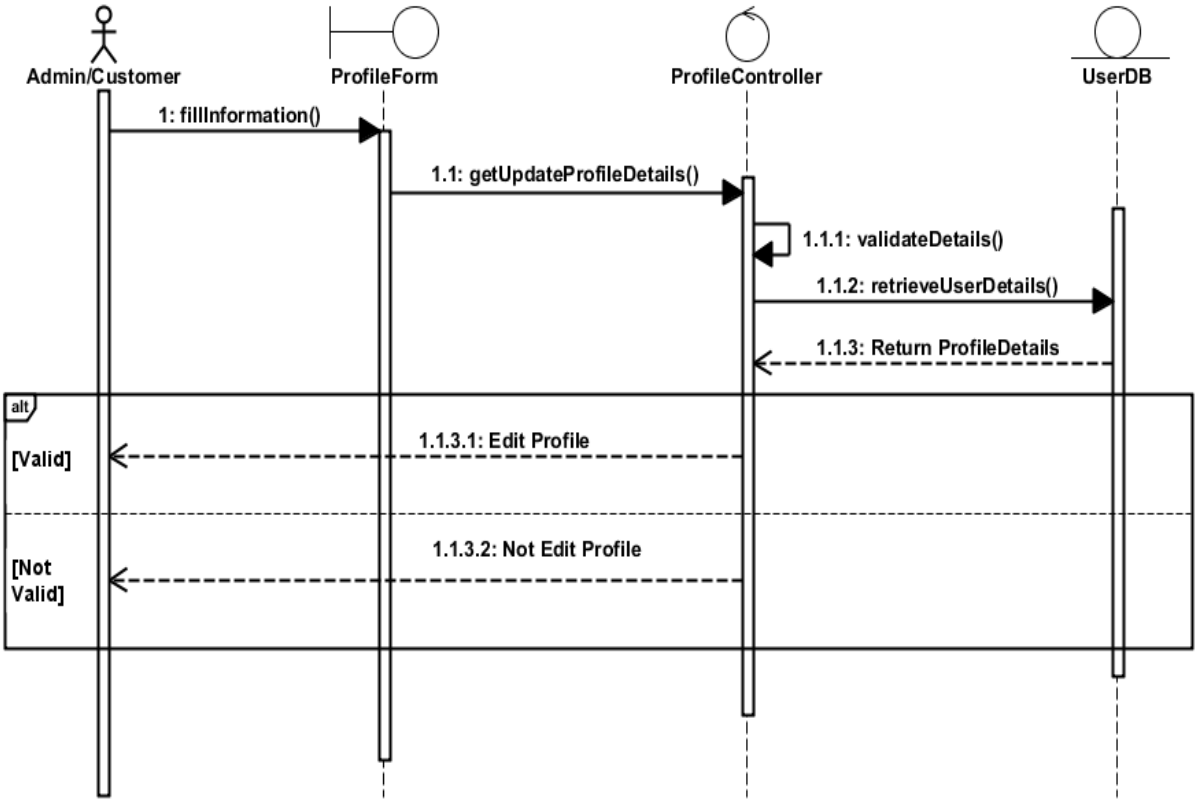
- **Registration**



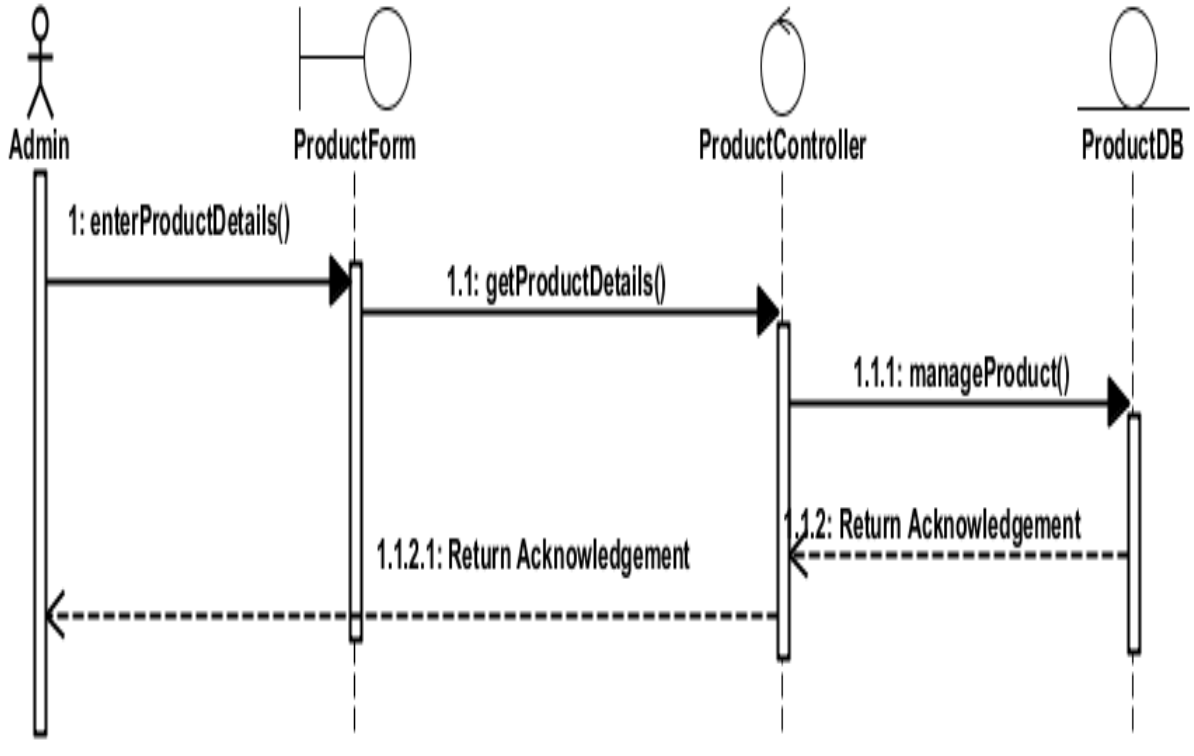
- **Login**



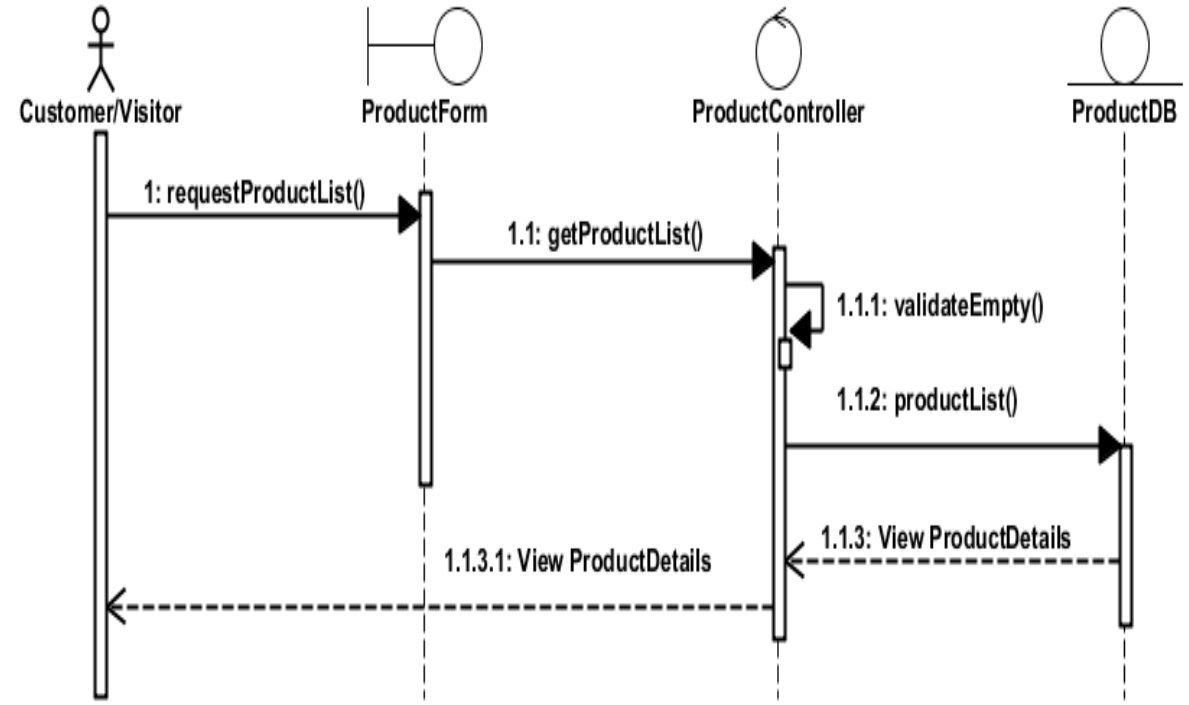
- **Manage Profile**



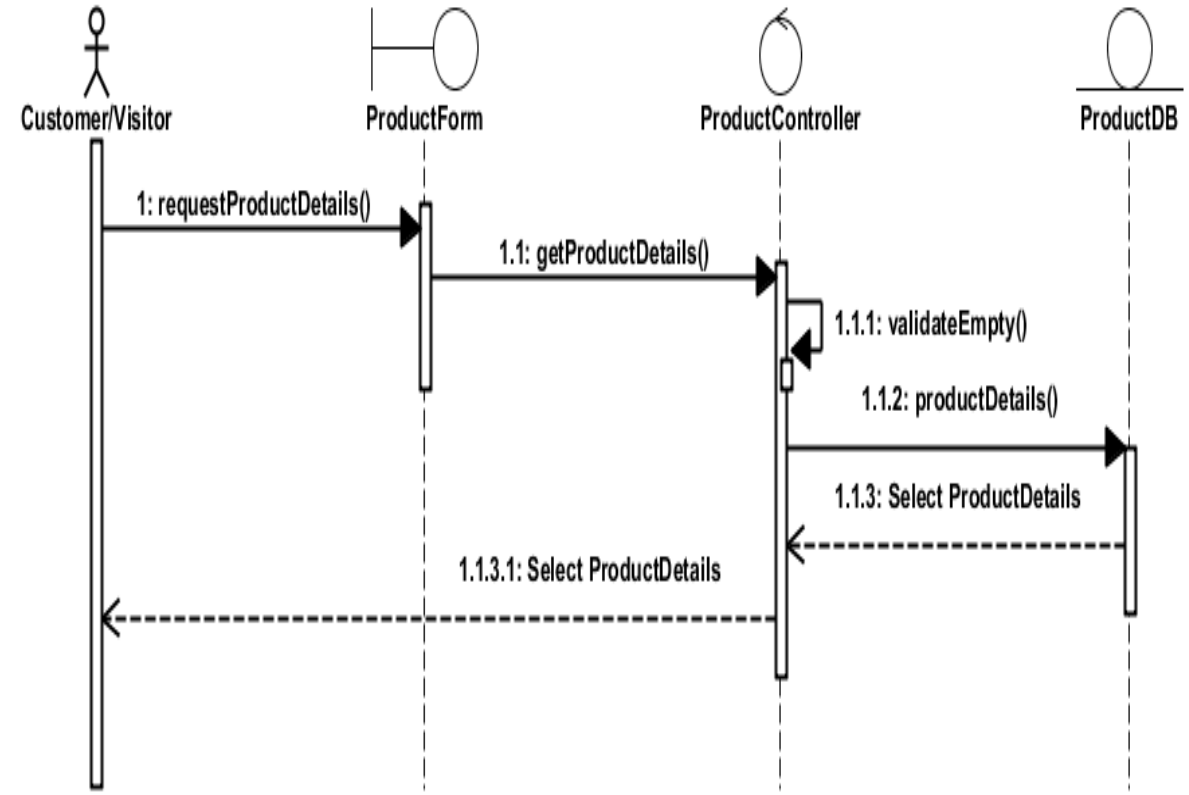
- **Manage Product**



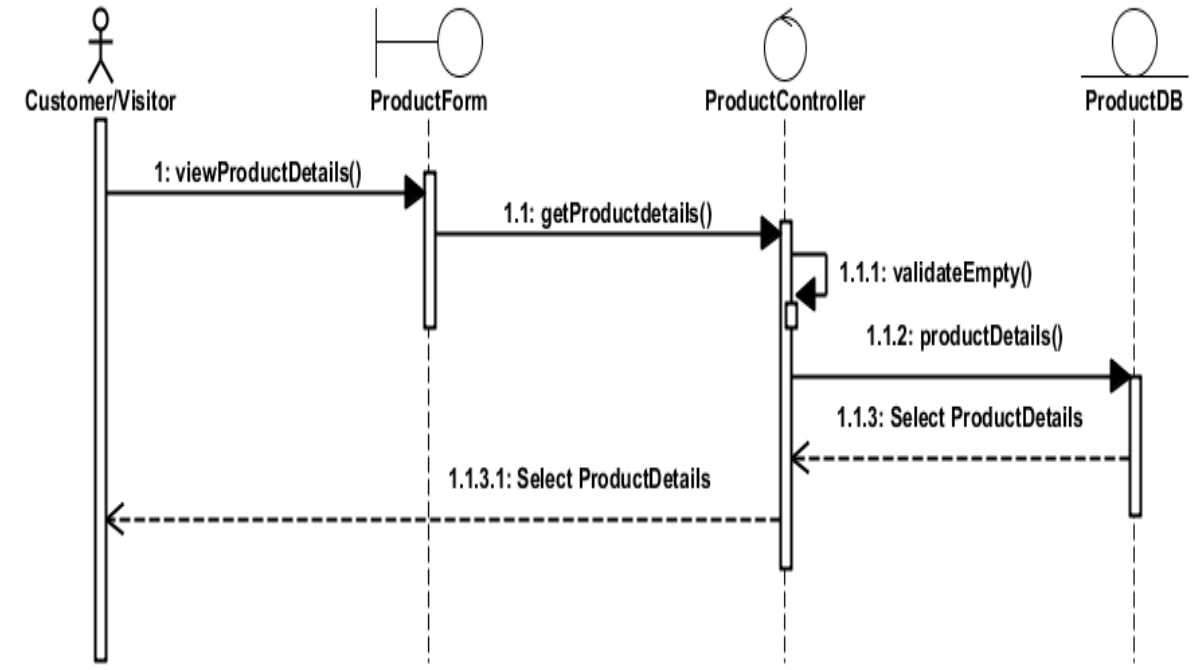
- **View Product**



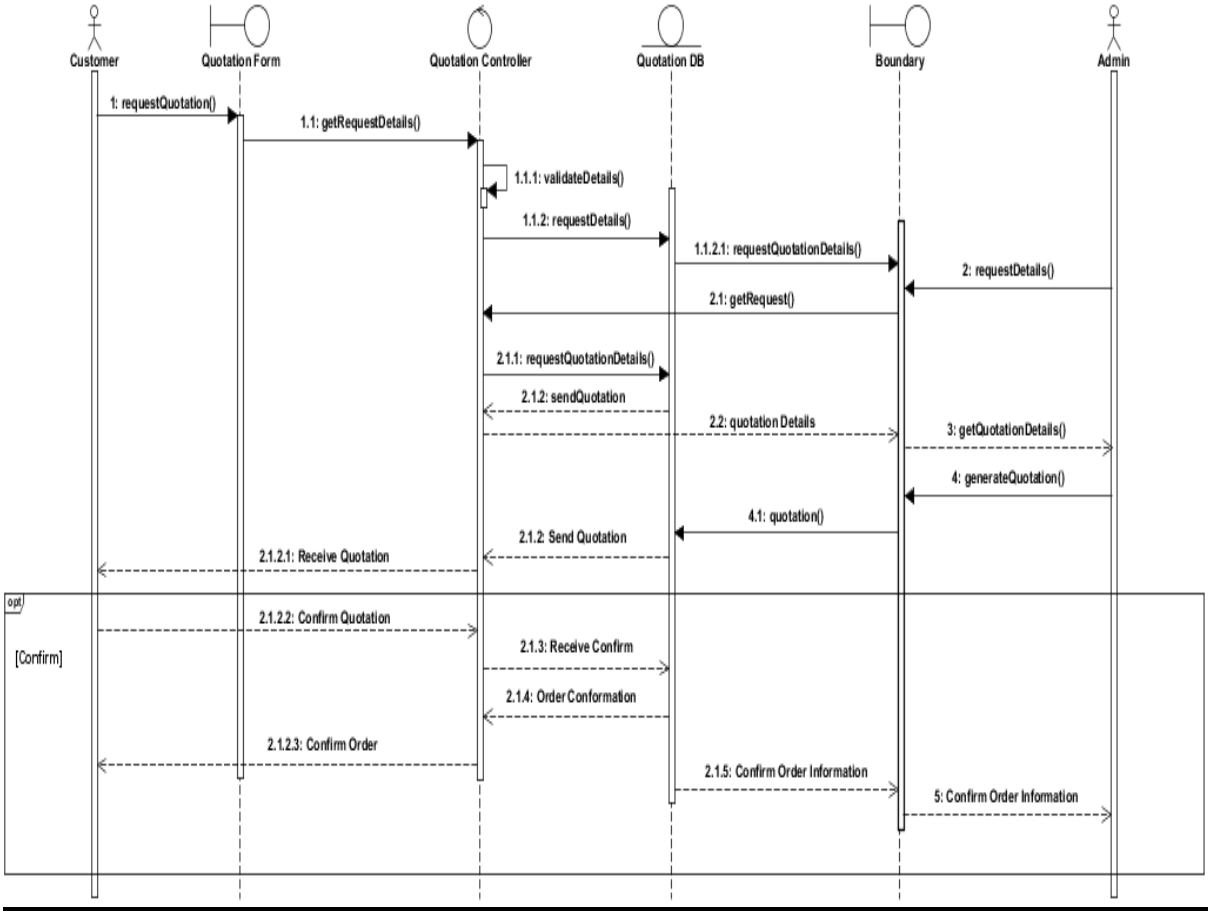
- **Search Product**



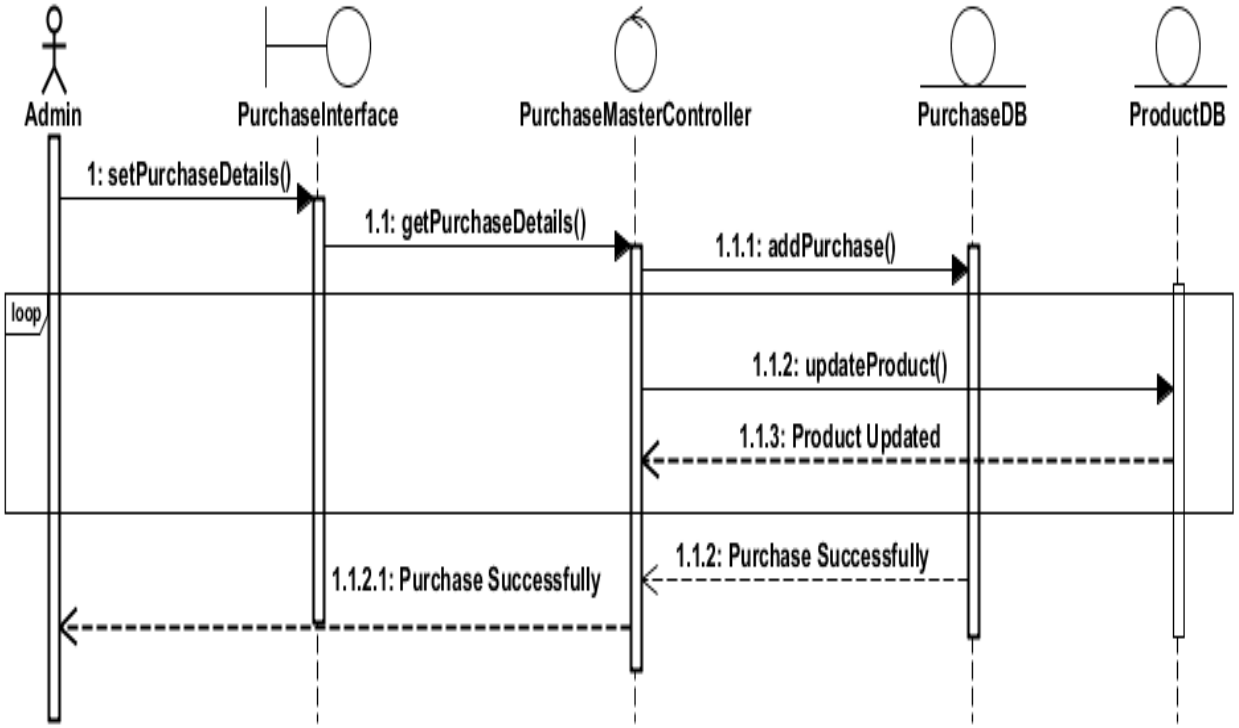
- **Select Product**



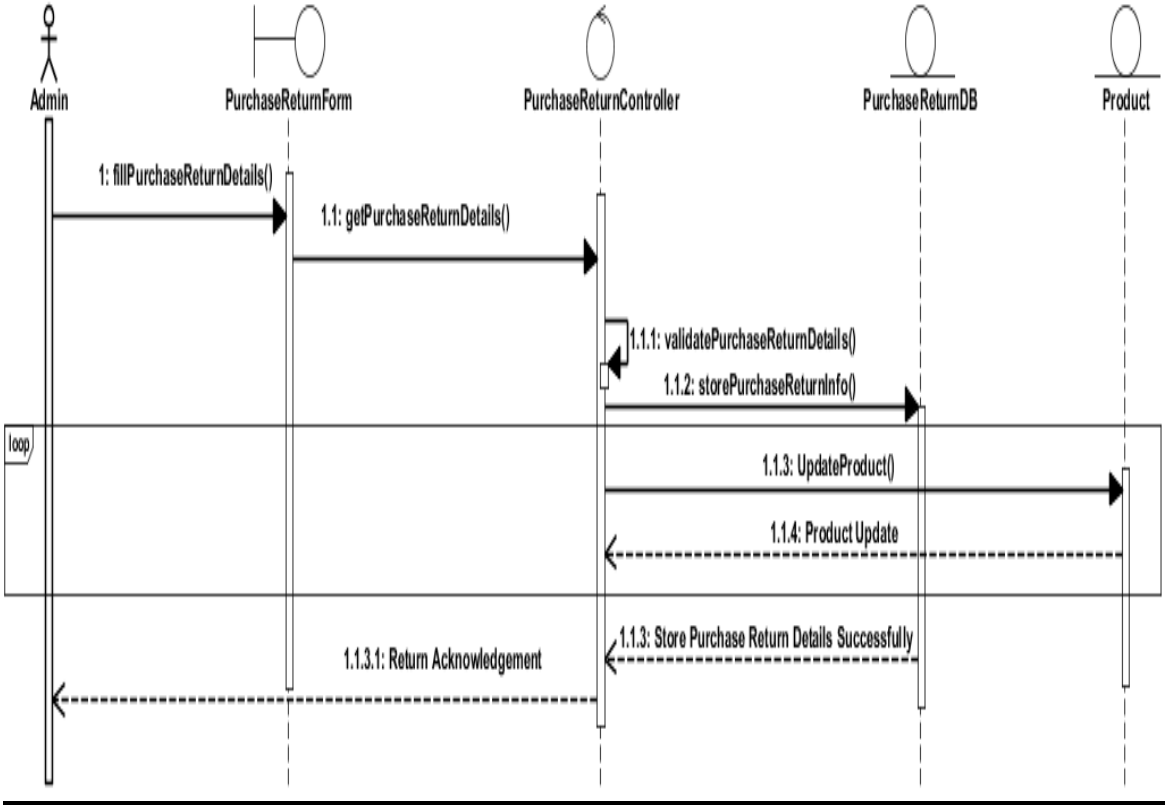
- **Quotation**



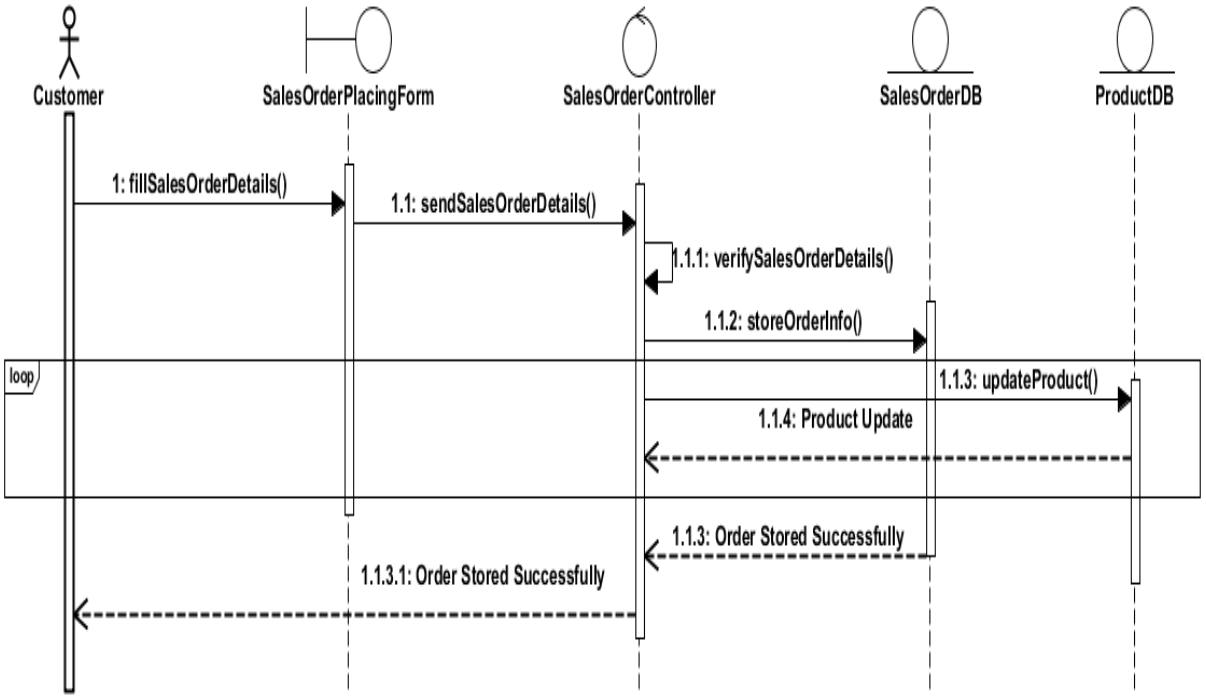
- **Purchase**



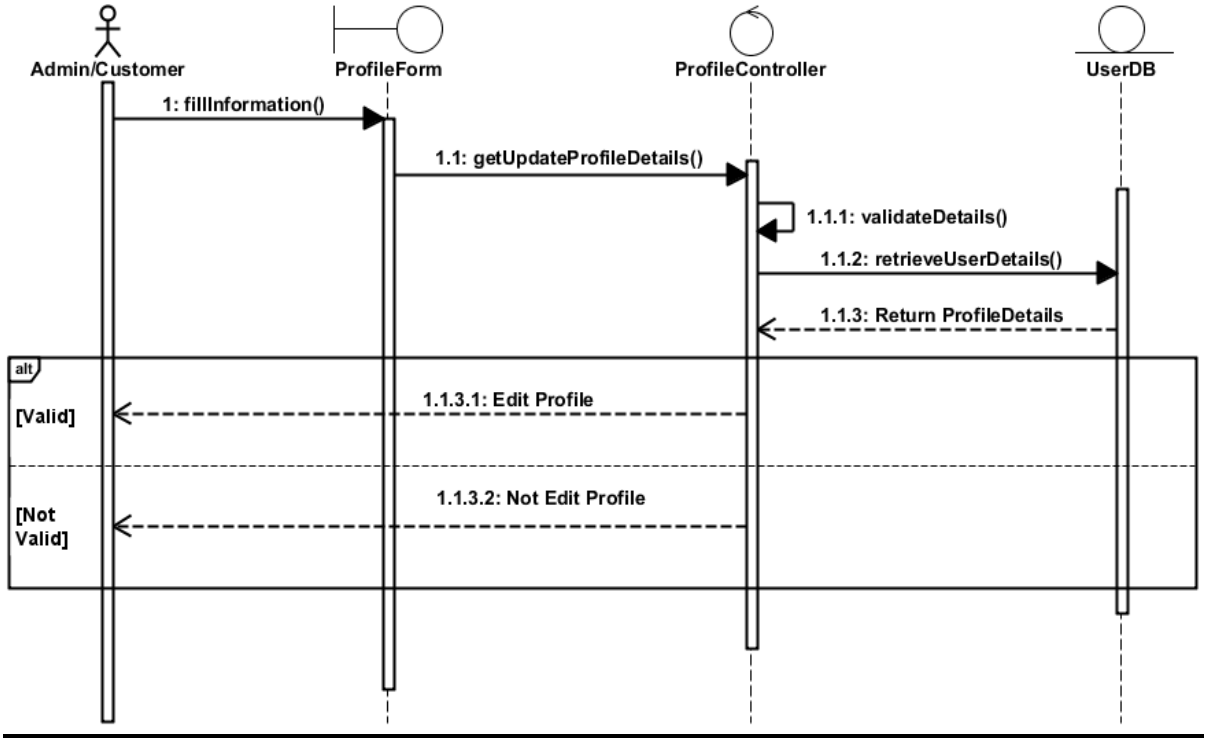
- **Purchase Return**



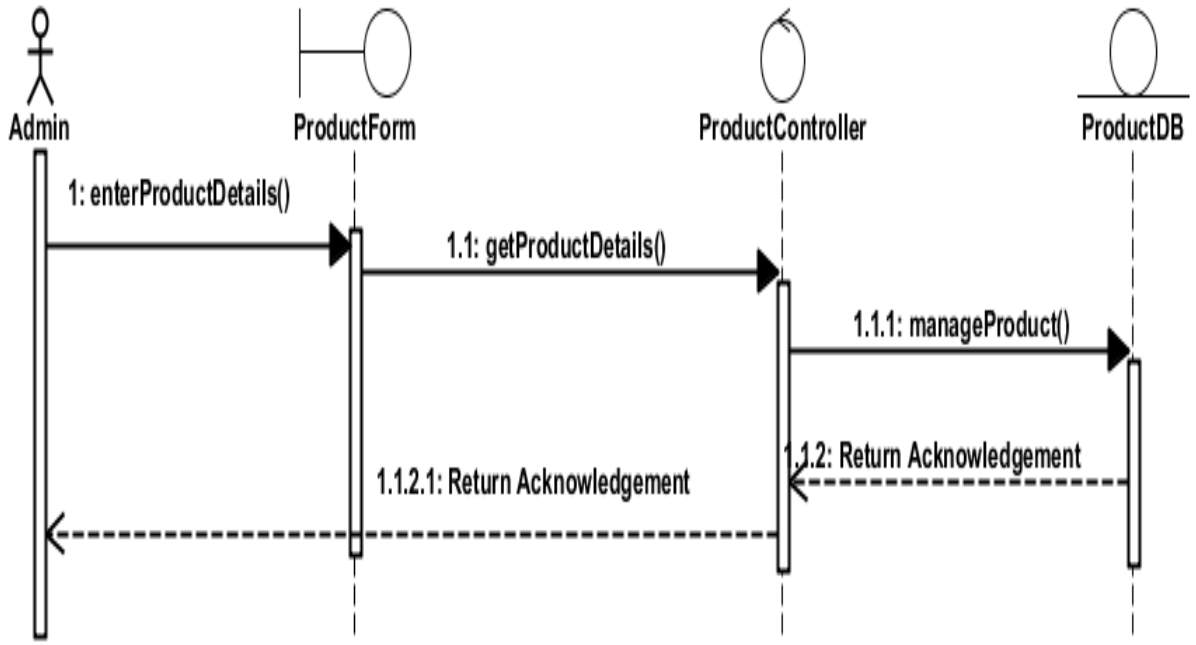
- **Sales Order**



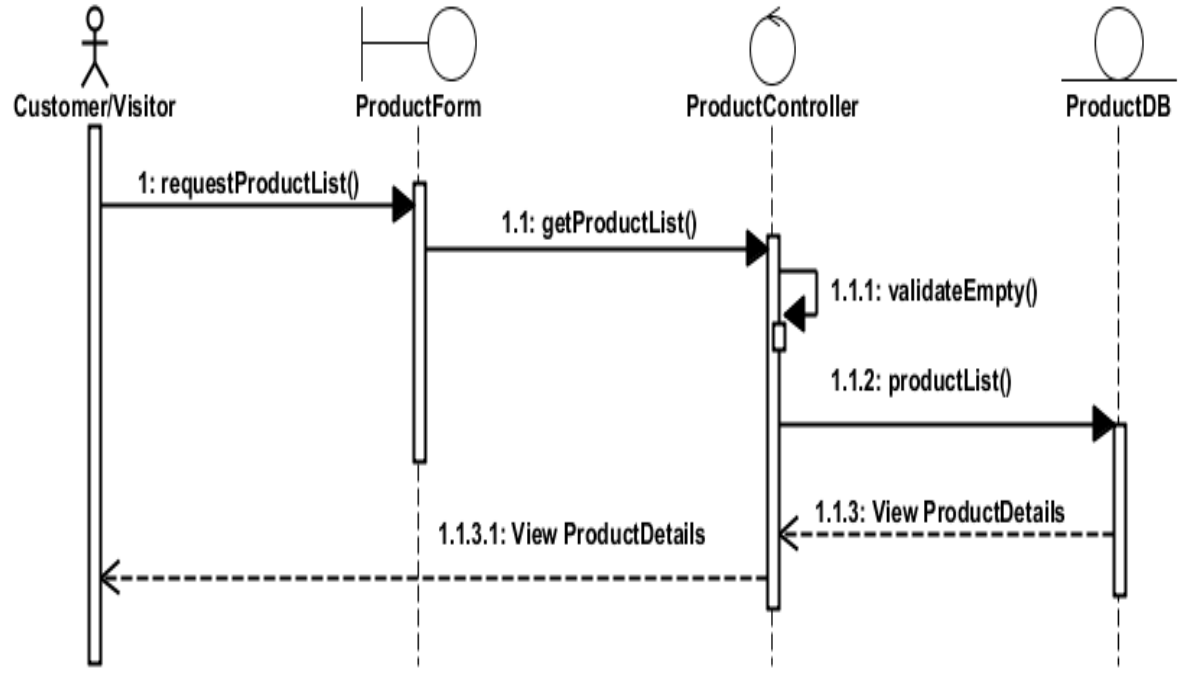
- **Manage Profile**



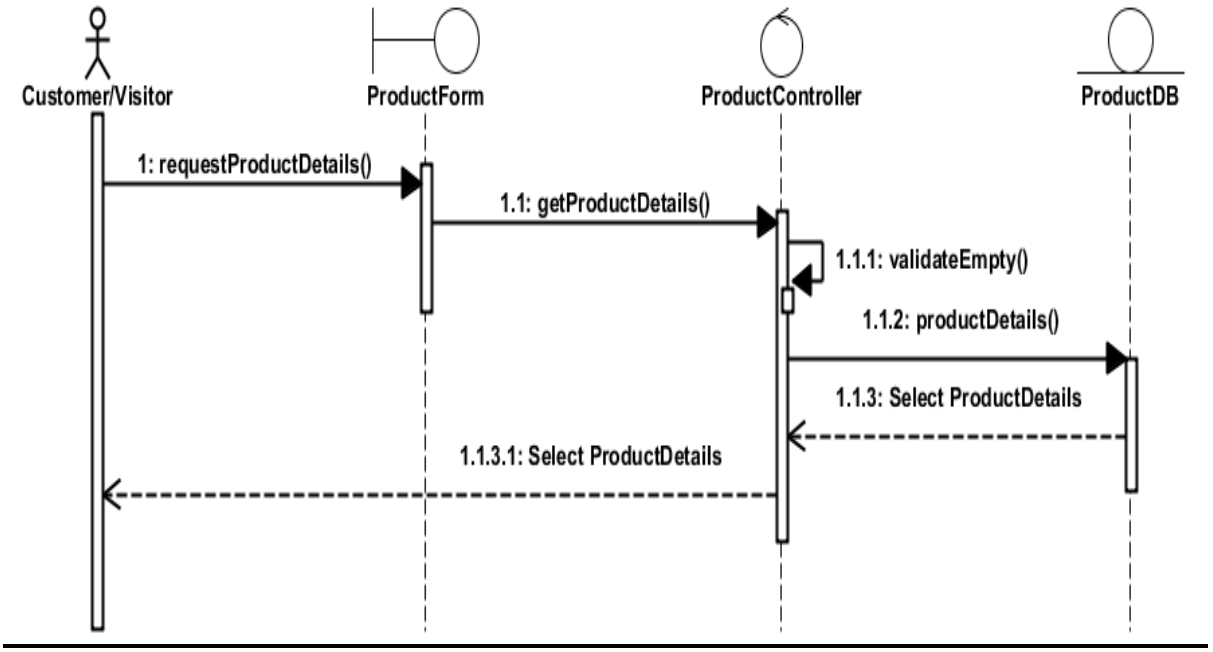
- **Manage Product**



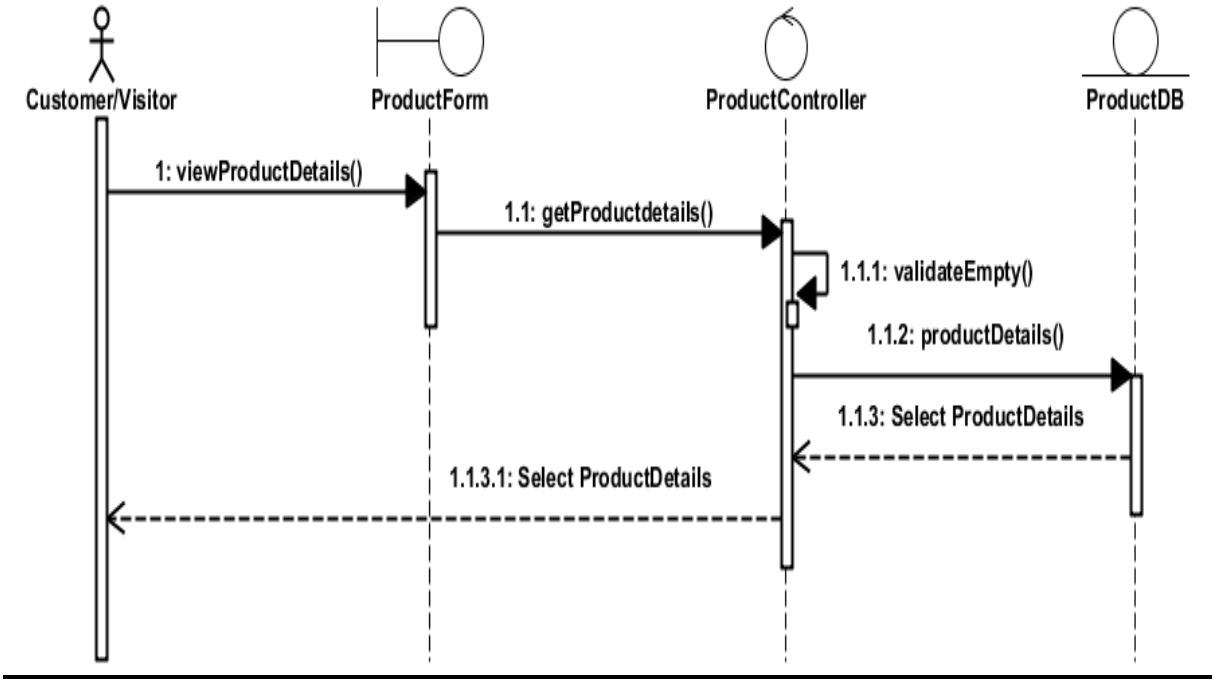
- **View Product**



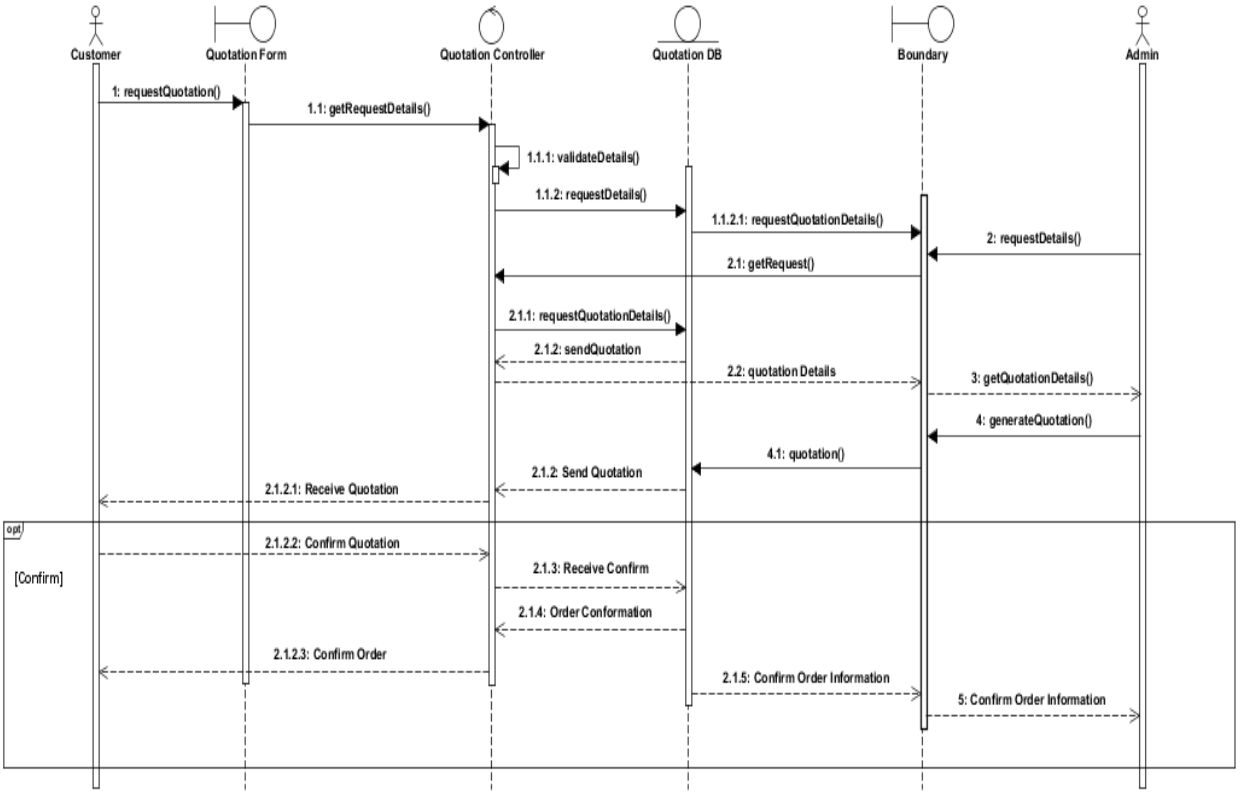
- **Search Product**



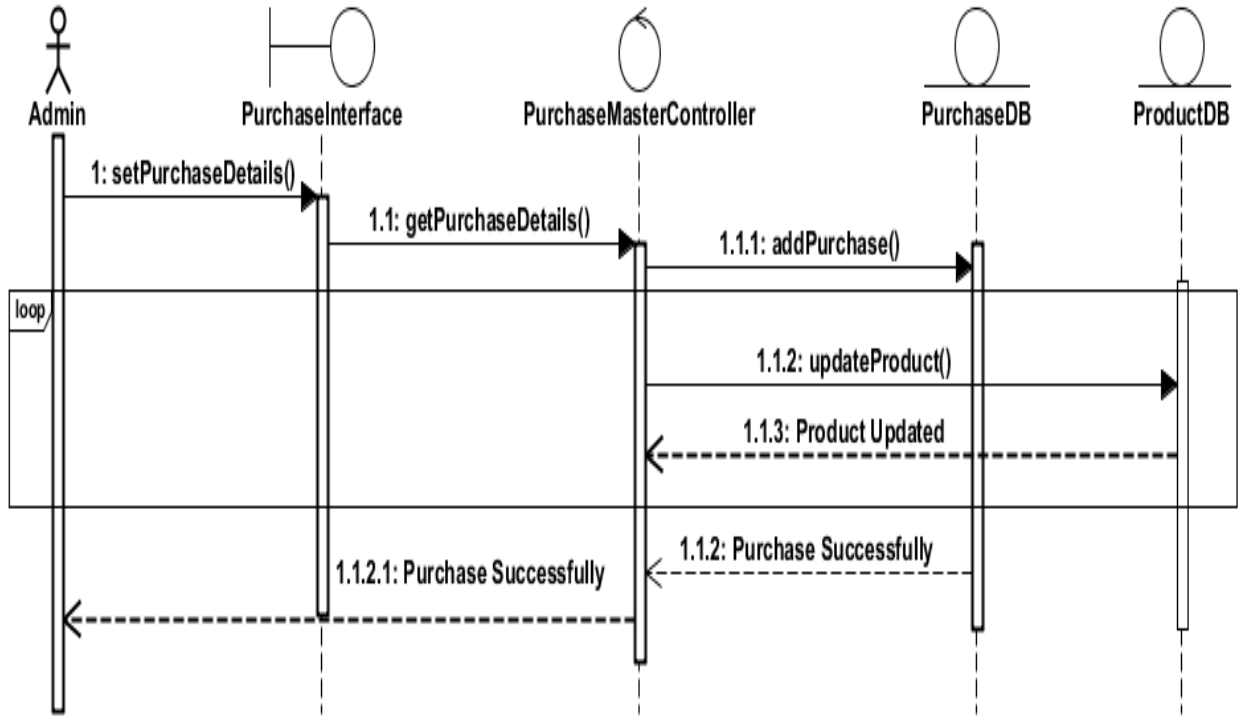
- **Select Product**



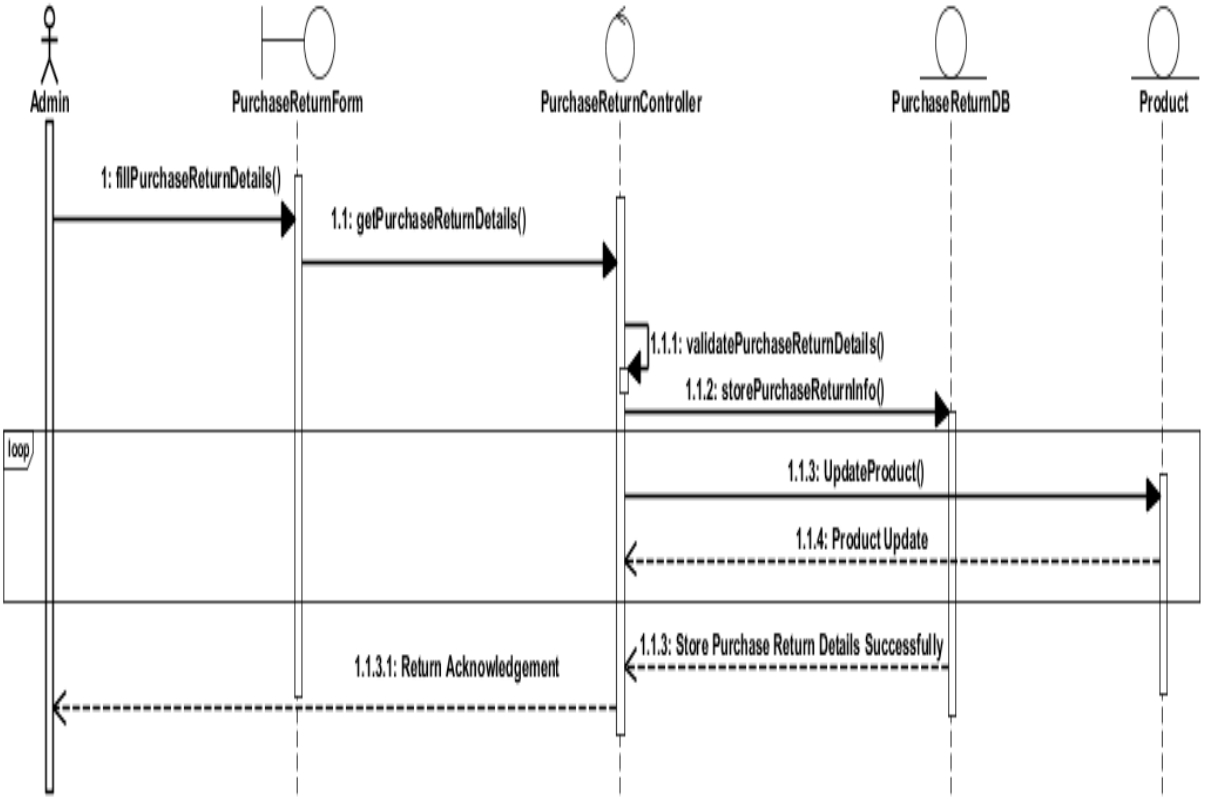
- **Quotation**



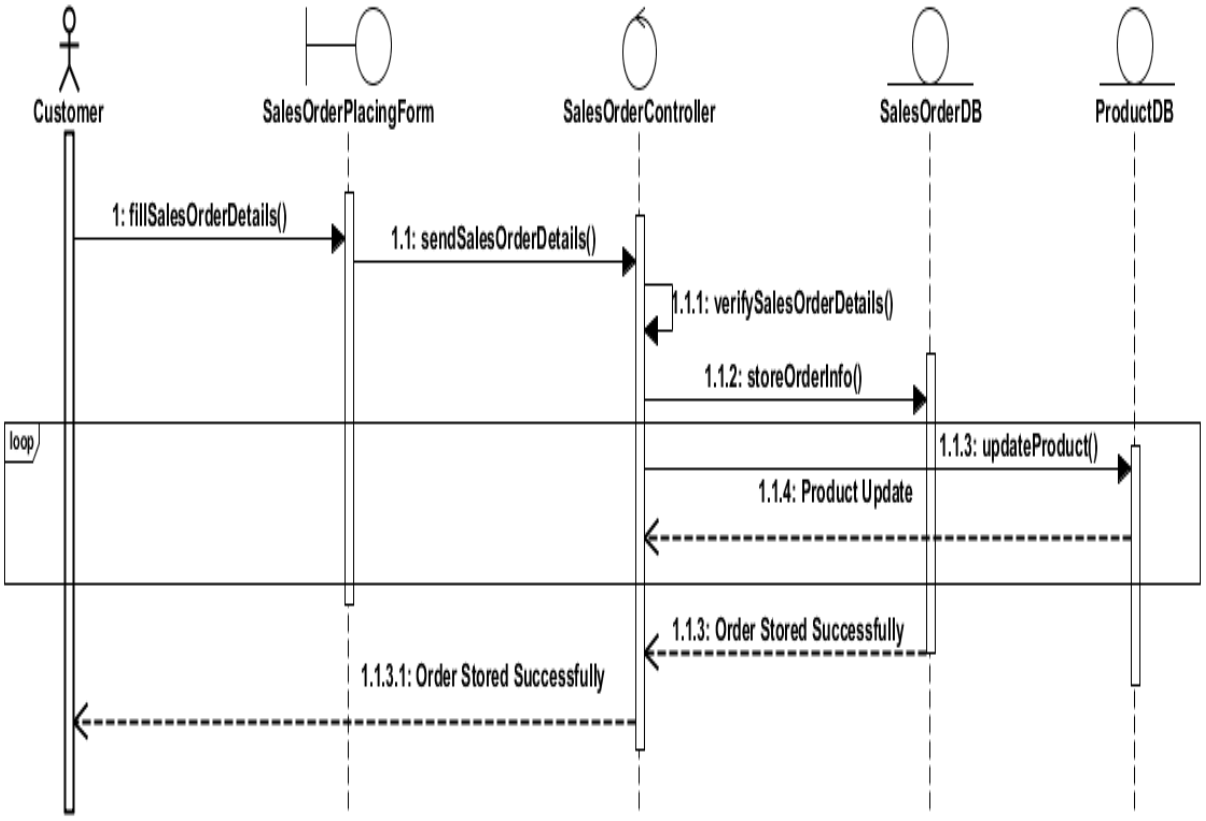
- **Purchase**



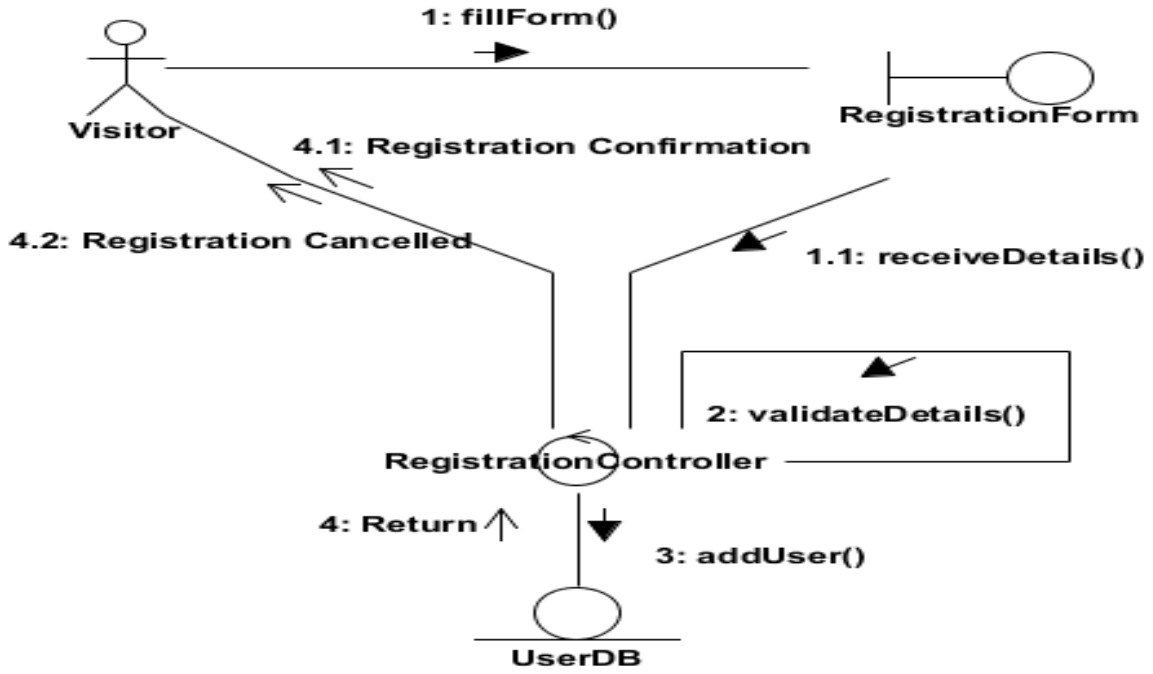
- **Purchase Return**

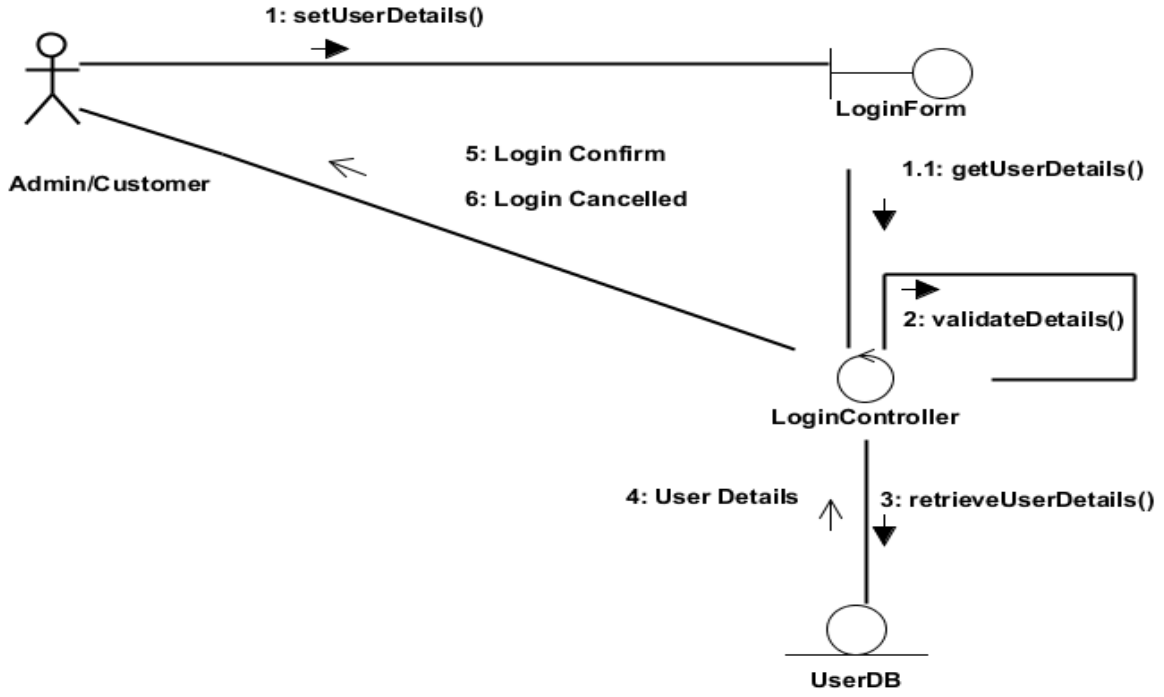


- **Sales Order**

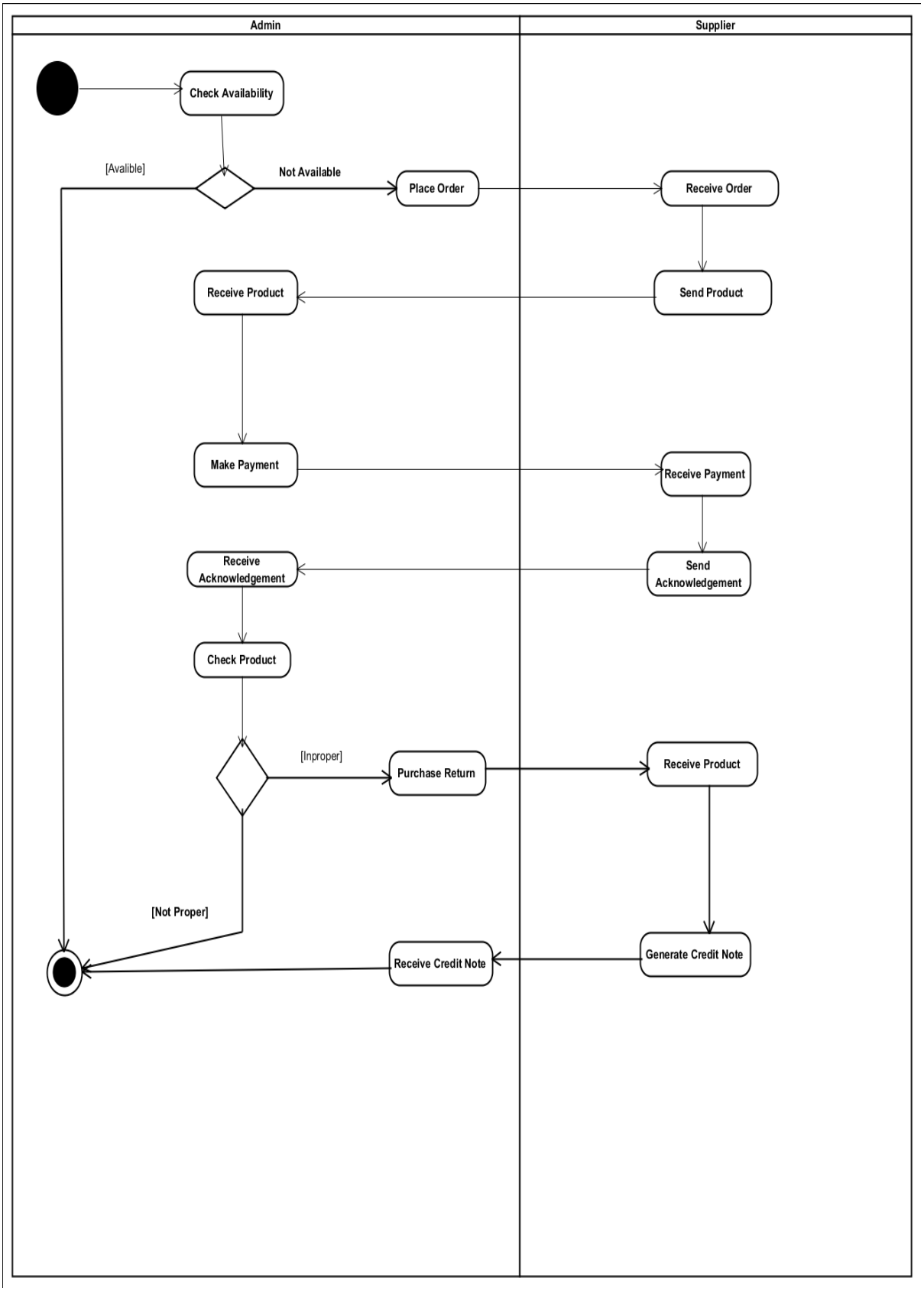


Collaboration Diagrams

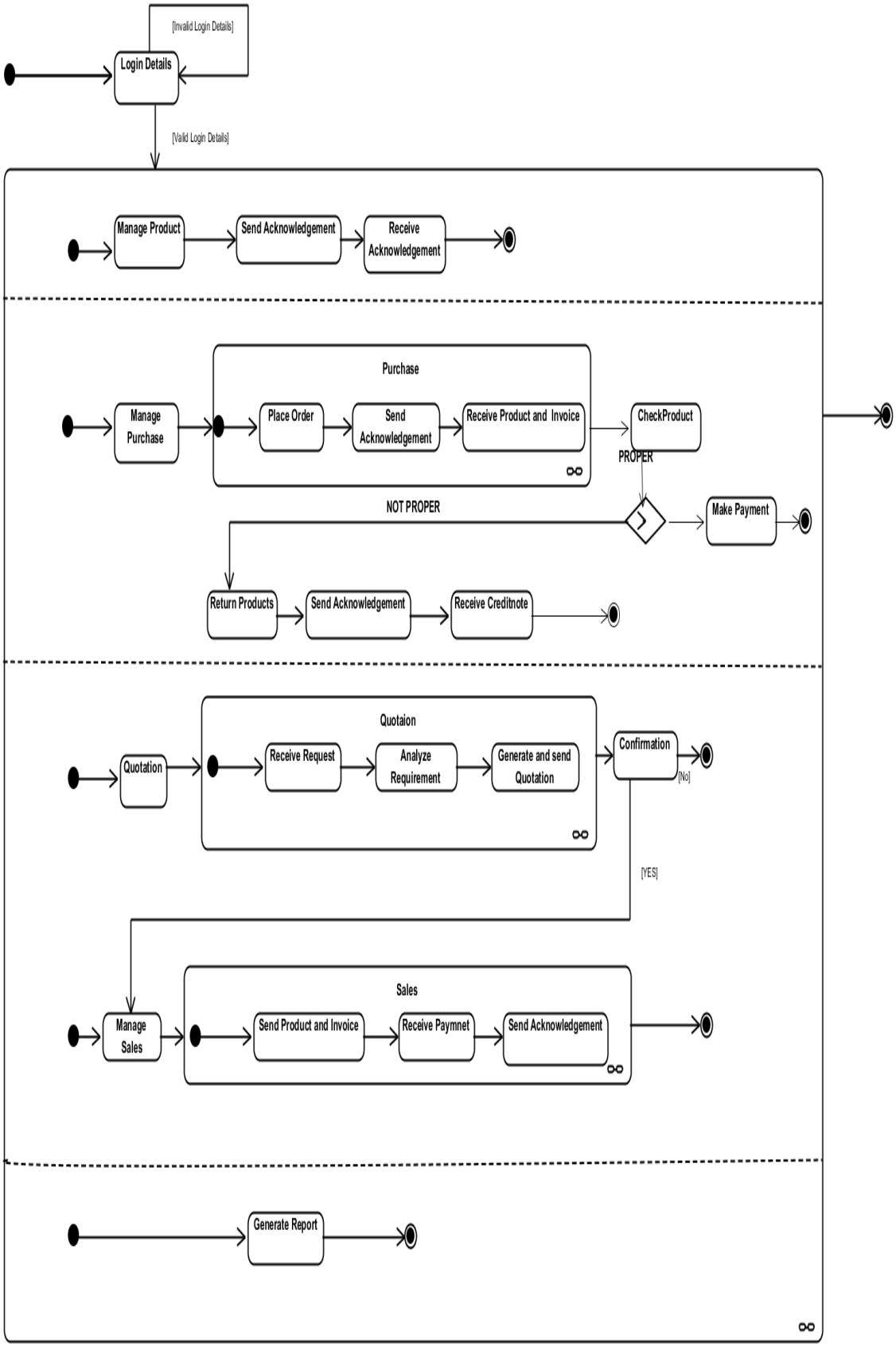




Activity Diagram



Activity Diagram activity Purchase



State-chart diagram for administrator user

Unit 10: Software Design 10

Unit Structure

10.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know – “What is design modeling”
- Understand the method of preparing design model using analysis model.
- Understand rules of preparing good modular design of software
- Understand meaning of coupling and cohesion
- Use patterns in the software design

10.2 INTRODUCTION

The main purpose of the Software Design phase is to convert the requirements of the customer, as stated in the SRS document, in the form of implementable using any programming language. To implement design straightforwardly into any programming language, following items are desired in the design phase:

- Different modules are needed to identified and implement design resolution
- Relationships among these identified modules are required.
- Interfacing between various modules is also needed. Interface classifies the exact data values communicated between the modules.
- Declaring data structures for each module.
- Algorithm needed to implement/code each module.

Remember that the design process of software takes software requirement specification (SRS) document as an input and produce the documents mentioned above. A good software design is rarely to achieve in a single-step procedure, and it requires several iterations. Software design can broadly categorize into two important phases:

- Preliminary or High-Level design
- Detailed or Low-Level design

In the Preliminary design we need to recognize different modules, relationships between these modules, identifies and defines interfaces (method of data exchange) between the modules. The outcome generated by Preliminary design is program structure which also known as system architecture. In the detailed (low-level) design,

data structures and algorithms for different modules are prepared. There are large number of methodologies exists to make good software design. We will discuss only few of them, but before discussing that we should understand what is good software design?

10.3 FEATURES OF GOOD SOFTWARE DESIGN

It is very difficult to feature-out good software design for large number of problems. For different types of application software, the term “Good software design” can varies, and it is dependent on the targeted application. However, many researchers and software engineers approved on few required characteristics, which are discussed below:

- **Correctness:** In ‘Good software design’ all the functionalities of the system should be implemented correctly. If all the functionalities are not properly implemented or should have errors in it then, software design will become worthless. So, it is very essential that the software design should be error-free and acceptable.
- **Efficiency:** A software design should also be efficient.
- **Maintainability:** A software design should easily adopt new features, any updates, or any kind of changes in the system.
- **Understandability:** A ‘Good software design’ is such, which can be easily understandable. If the software design is simple and easy to understand, it will also be easy to implement. If the system design very complex to understand, maintainability of system will become tedious and it increases the efforts.

For ‘Good software design’, having all the features discussed above, a software engineer has to take care for the following things:

- Different design components should have meaningful and reliable names. It should increase the readability and easy to understand.
- The ‘Good software design’ should be modular in nature. By mean of modularity, the larger software has to be divided in to clean set of sub-programs called modules.

- The ‘Good software design’ should not be complex. The modules of the system should be arranged in layered approach in hierarchical (tree shaped) diagram. It is also called well-ordered arrangement of all the modules.

10.4 DESIGN CONCEPTS

In this section we will discuss the concepts of designing the system:

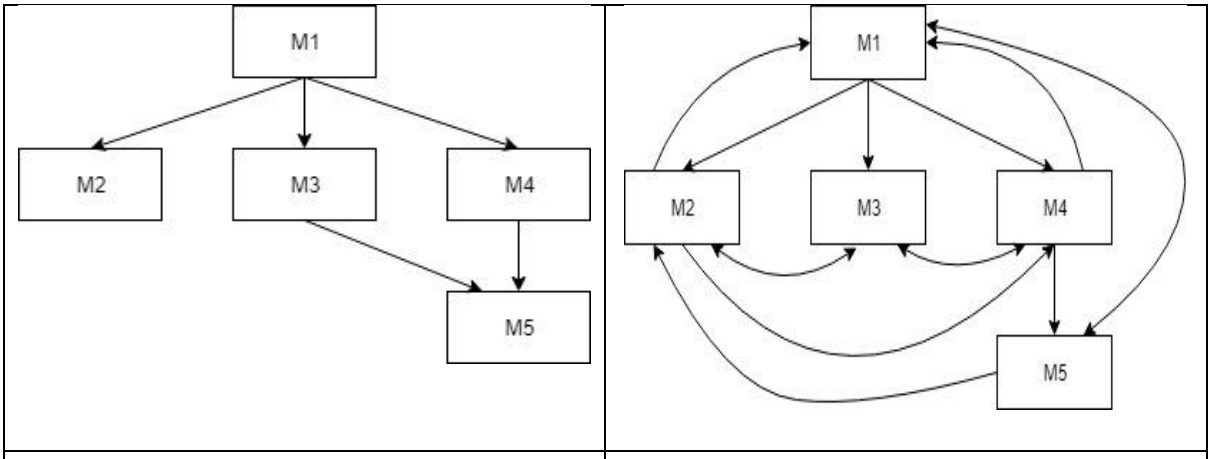
Abstraction: At the uppermost-level of abstraction, solution of the problem situation is described in the detail manner. At lower-level of the abstraction, details solution of the problem is provided. As we move in different levels of the abstraction from uppermost-level to lower-level, we are producing the procedural (sequence of instructions) form of the abstraction. In general, abstraction is the process of identifying attributes and methods. Data abstraction is a named group of data elements that represent a data object.

Information hiding: The term information hiding is related to controlled interfaces. Hiding means that effective modularity can be achieved by defining by a set of independent modules that interconnects with one another in such a way that only required information can be exchanged between them. The different types of data are encapsulated as a single unit which cannot accessible directly by other modules.

Modularity: Large and complex system has to be divided into number of small and manageable sub-components which are individually named and referenceable. The small and manageable sub-components are called modules. Larger program developed as a single unit or module is not readable, also difficult to manage and tough to implement. Rather than handling the whole software as a single unit, it is easier to solve a lengthy and complex problem, by breaking it into controllable pieces – “Divide and conquer”. Modularity reduces the complexity and enhances readability of the problem.

Clean decomposition: The concept of clean decomposition is focused on separation of different modules. How two modules can be separated from each other? Obviously, the answer of this particular question is – by their functionalities. The identifications of different modules have to done by detecting their functionality. Module should have many functions, which are strong enough to perform the job either independently or

with the support of other functions from the same module. Function calls outside the module (external functions) is not a good design. ***In the software design modules should be high cohesion and low coupling***. ***It means modules should more self-dependent and less interdependence on each other.*** Consider the following figure:10.1, to distinguish the good software design and bad software design, in which M1, M2, M3 …etc. are the modules, and their dependencies are denoted by arrows.



[I] Good software modular design | [II] Poor software modular design

Figure:10.1 Modular designs

Functional Independence: Functional independence is a key concept for good software design, and good design is the key concept for quality software product. Function independence represents the strength of the function. The function has to be robust enough and it should not be reliant on other functions of other modules. The module decomposition should have high cohesion and low coupling. Cohesion and coupling are discussed in the next topic (9.5) in greater details.

A module having high coherence and low coupling is said to be a functional independence of other modules.

Module arrangement: In a good design of software, the modularity should have following features:

- Layered solution
- Low fan-out
- Abstraction

Patterns: Design pattern brings the spirit of a proven design solution to a recurrent problem within a certain situation within computing concerns. A design pattern

provides a design structure that solves a particular design problem within the specific situation, and within “forces” that may have an impact on the manner in which the pattern is applied. The commitment of each design pattern is to deliver a description that enables a designer to determine:

[1] Is pattern can be applicable to the current work?

[2] Is the patter reusable?

[3] Are the pattern serves as a guide for developing a similar, but structurally or functionally different pattern?

10.5 COHESION AND COUPLING

We have discussed above that the “Good Software Design” suggests clean decomposition of the modules of the given problem. To obtain this, arrangement of the module is most needed. The terms cohesion and coupling are related to the arrangement of modules. In the arrangement of the modules, we have to follow thumb rule – “High cohesion and Low coupling”. Cohesion is a measure of the functional strength of a module (function of one module should not be dependent on other functions of other modules) and coupling is a measure of the degree of interaction or interdependence between the modules. High cohesion and low coupling means modules have to be functionally independent of other modules. Cohesive module performs single task of function. It should not be depended on the functions of other modules. Functionally independent module has less interaction with other modules.

If the modules are functionally independent then, that design is considered to be a – “Good software design” and it helps us in following ways:

Reusability: It allows us to reuse the module, because each module has well-defined and precise functions, which are not dependent to other modules. Therefore, cohesive module can easily be taken out from one project and can easily reuse it into the other projects when it is needed.

Understandability: If the modules are arranged with high cohesion and low coupling, have simpler and less complex design which is easier to understand. This makes error separation much simpler.

Isolation of Errors: If the modules are dependent on each other, then error of one module, will be propagated into other modules. If the modules are independent then chances of propagating error will be reduced. It is easier to find the error and eliminate it from the function.

10.6 DESIGN MODELING

A design model is an object-based picture(s) that represent the use cases for a system, or represent it in another way. It represents the system implementation and source code in a diagrammatic fashion. The couple of advantages of the design models are:

1. Representation is much simpler that it represented by words.
2. Any person can see the diagrammatic representation and quickly get the general idea of the system.

As we know, the design modeling is based on the analysis, it also involves number of steps:

1. Data design elements

The data design element produced a model of data that represent a high level of abstraction. This model is then more refined into more implementation specific representation, which is processed by the computer-based system. The structure of data is the most important part of the software design.

2. Architectural design elements

The architecture design elements provide us overall view of the system. The architectural design element is generally represented as a set of interconnected subsystems that are derived from analysis packages in the requirement model.

The architecture model is derived from following sources:

- The information about the application domain to build the software.
- Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
- The architectural style and pattern as per availability.

3. Interface design elements

The interface design elements for software represents the information flow within it and out of the system. They communicate between the components defined as part of architecture.

Following are the important elements of the interface design:

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface between various components.

4. Component level diagram elements

The component level design for software is similar to the set of detailed specification of each room in a house. The component level design for the software completely describes the internal details of each software component. The processing of data structure occurs in a component and an interface which allows all the component operations. In a context of object-oriented software engineering, a component shown in a UML diagram. The UML diagram is used to represent the processing logic.

5. Deployment level design elements

The deployment level design element shows the software functionality and sub-system that allocated in the physical computing environment which support the software. The components arrived at during the component level design step are groped for the purpose of delivery to their final destination.

10.7 PATTERN BASED SOFTWARE DESIGN

Developing software is challenging task, and developing a software that can be easily reused in the other projects is even harder. The designs for the various sections of the software coding, should be general enough so that it can be utilized in the future problems. Pattern are useful in designing the software in determining the appropriate granularity and in designing the system architecture that can be reused in the future projects as well as easy to update or change in the future. At a design level, patterns

allow large-scale reuse of software architectures by capturing the expert's knowledge of pattern-based software development.

DESIGN OF PATTERN TEMPLETE:

Name of the Pattern: Describes the essence of the pattern in a short but expressive name.

Intent: Describe pattern and what it does.

Also-known as: List of the similar words to the pattern.

Motivation: Describe the example of the problem

Applicability: Record design solutions in which the pattern can be applied.

Structure: Describe the structure of class to implement pattern

Participants: Describe the responsibility of the class, we have designed into the implementation of pattern.

Collaboration: Describes collaboration of the participants to carry responsibilities.

Consequences: Focuses on the considerable potential trade-offs, in the implementation of pattern.

Related patterns: Provides references to the related pattern designs.

DESCRIBING PATTERN DESIGN:

1. Good designers of any field have ability to see patterns that characterized a problem and related pattern that can be implement to create a solution.
2. Description of the pattern design can be considered a set of design forces.
    a. All non-functional requirements (e.g., portability, ease of maintainability) associated the software for which the pattern is to be applied, is described by Design forces.
    b. Design forces describes conditions and environment, that may exist in the pattern design.
    c. Design forces also describes the constraints that may restrict the manner in which the design pattern is to be implemented.
3. To accommodate a variety of problems, the attributes of the patterns (classes, collaborations etc.) are adjusted.

4. The attributes which represent the characteristics of the pattern may store in the data base, so that based on the attributes we can search the pattern.
5. Guidance related to any complications should be provided in the pattern design.
6. Pattern design should have appropriate name.

HOW TO USE PATTERNS IN DESIGN?

After developing the analysis model, software designer can examine detailed representation of the problem to be solved. Also designed has to focused on the contrarians that are imposed by the problem. Design patterns can be used throughout the software design. Examination of the problem description at each level of description opens one or more different types patterns, which are discussed below:

Architectural patterns: Architectural pattern defines the overall structure of the software. The overall structure of the software indicates components (subsystems) or the software, relationship between subsystems, and rules which specifies relation among packages, classes, components, or subsystem of the architecture.

Design patterns: Design pattern addresses a specific element of the design such as an aggregation of components to solve some design problems, relationships among components, or the mechanisms for effecting component-to-component communication.

Idioms: Idioms are also known as coding patterns, these language-specific patterns generally implement an algorithmic element of a component, a specific interface protocol, or a mechanism for communication among components.

Exercise: 1

1. _____ is a measure of the functional strength of a module.
2. _____ between two modules is a measure of the degree of interdependence or interaction between the two modules.
3. A module having high _____ and low _____ is said to be a functional independence of other modules.
4. _____ design concept suggests to divide the large unmanageable system, into number of manageable sub-systems.
5. _____ system analysis model, focuses on events, state and state transitions.

10.8 Let us sum up

In this chapter we have learnt how can we make design model from the analysis model. We have discussed that design modeling is prepared from the analysis model. Each abstraction level of analysis modeling is translated in the design modeling where the components of the software such as packages, classes, and relationship among them is prepared. We hope student can now understand design modeling and its use in software production.

10.9 Check your progress: Possible Answers

Exercise: 1

1. Cohesion
2. Coupling
3. Cohesion, Coupling
4. Modularity
5. Behavioral Model

10.10 Further Reading

1. Software Engineering – A Practitioner's Approach by Roger S. Pressman (McGraw-Hill international edition).
2. Fundamentals of Software Engineering by Rajib Mall (PHI)
3. System Analysis and Design Methods by Gary B. Shelly, Thomas J. Cashman, Harry J. Rosenblatt (CENGAGE Learning)
4. Magnifying object-oriented analysis and design by Arpita Gopal and Netra Patil (PHI)
5. Object-oriented modeling and design by James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lor
6. "Software Engineering" by Dr. Ruchita Shah, Dr. Kamesh Raval, Mr. Nitin Shah. ISBN No: 978-81-942146-4-9 From: Dr. Babasaheb Ambedkar Open University

Dr. Babasaheb Ambedkar Open University

BCAR-402

Software Engineering

BLOCK 4: SOFTWARE TESTING

BLOCK 4: SOFTWARE TESTING

Block Introduction

In this block-4 of the Software Engineering, we have discussed about software testing. We will discuss What is software testing? Why software testing is important for producing quality software products? How many types of testing? And What are the level of the testing?

With the advancement in software industry, demand of quality products has been increasing significantly. To develop quality products software testing plays an important role. Before anyone else is finding error or bug from the software, it is essential to find out the bugs and eliminate it from the software product become essential. Study software testing explain different methodology so that an engineer can find the bugs from the software product.

In this block, we will learn how software testing can be done and also what different types of approaches are used to find out bugs from the software in the software industry.

Block Objective

The main objective of the block is to explain importance of software testing and how different types of testing like black-box and white-box testing can be done. Now a days, software testing is done as a separate phase of SDLC, but it is included in each phase of SDLC.

Mainly software testing is divided into two categories: [1] Static testing and [2] Dynamic Testing. Inspection, Walkthrough are the methods of static testing. Where as Dynamic testing can be divided in two categories. [1] Black-Box testing where the code to implement software is not considered and [2] White-Box testing where the code to implement software functionality is considered in testing. Different methods are used in the black-box and white-box testing are used in the industry. Few widely used methods are explained in this block.

Testing process for the software is also done on different level for example unit testing, integration testing, system testing, acceptance testing. This block aim to give brief idea about all these different types of testing techniques.

Block Structure

BLOCK 4: SOFTWARE TESTING

UNIT 11 SOFTWARE TESTING CONCEPTS

Objectives, Introduction, SDLC, SDLC models, Quality concepts, Verification and Validation, Goals of software testing, Static and Dynamic testing, Let Us Sum Up

UNIT 12 BLACK-BOX TESTING

Objectives, What is black-box? Need for black-box testing, Advantages and disadvantages of black-box testing, Boundary value analysis, Equivalence class partitioning, Decision tables testing, Let Us Sum Up

UNIT 13 WHITE-BOX TESTING

Objectives, White-box testing, Need of white-box testing, Advantages and disadvantages of white-box testing, Black-box vs White-box testing, Logic coverage criteria, Basis path testing, Let Us Sum Up

UNIT 14 SOLVED PROGRAMS-III

Objectives, Unit testing, Integration testing, System testing, Performance testing, Acceptance testing, Let Us Sum Up

Unit 11: Software Testing Concepts

11

Unit Structure

11.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know various testing concepts
- Understand SDLS
- Learn how different SDLC models works
- Learn different quality management terms

11.2 INTRODUCTION

Testing is the process of demonstrating that there are no errors. The purpose of testing is to show that the software performs its required functions correctly. It is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements. Testing is not an intuitive activity rather than a process. Therefore, testing should be performed in a planned way. Testing is the process of executing a program with the intent of finding errors. It is a concurrent lifecycle process of engineering, using and maintaining test-ware in order to measure and improve the quality of the being tested. Software testing should be effective, not exhaustive.

11.3 SOFTWARE TESTING CONCEPTS

- **Software Testing**: Software testing is the process of demonstrating that there are no errors.

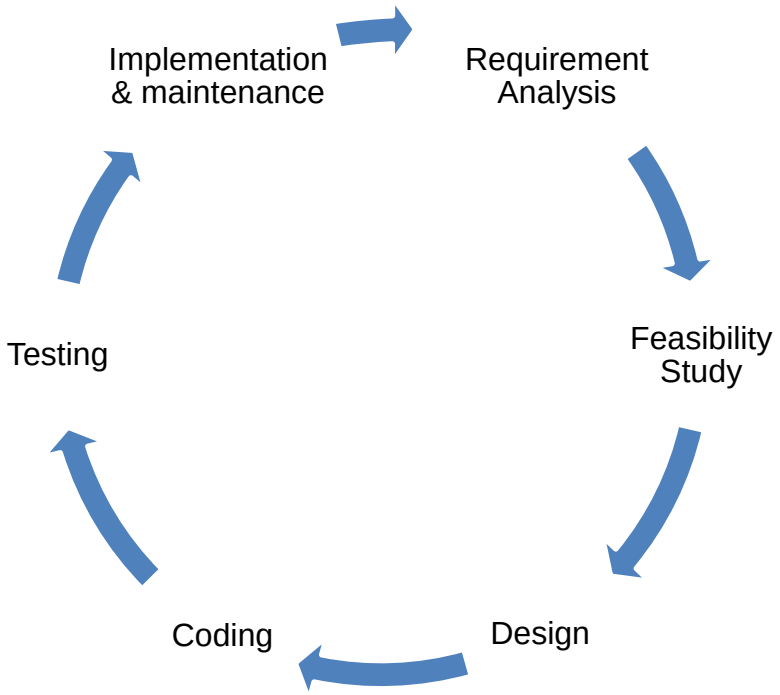
- **Errors:** Whenever a development team member makes a coding mistake then errors are produced.

- **Fault:** When error exists, fault occurs. A fault, also known as a bug or defect, is a result of an error which can cause system to fail.

- **Failure:** It is an inability of a system or component to perform a required function according to specification.

11.4 SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process. It is a pictorial and diagrammatic representation of the software life cycle.



Activities covered by SDLC are,

- Requirement Analysis
- Feasibility study
- Design
- Coding
- Testing
- Implementation and Maintenance

Requirement Analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. This activity gives the view of scope of the entire project. Requirements Gathering stage need teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

Feasibility Study:

Feasibility study is the series of tests that defines whether to proceed further to develop a system or not. It is performed by the project team. If the system is feasible with all criteria, then the organization accepts the system to be developed. Feasibility study can be,

- *Economical feasibility* looks into number of expenses and income.
- *Technical feasibility* checks for required hardware, software and network resources to develop a system.
- *Operational feasibility* ensures that the system will be easily operatable once it is developed.
- *Schedule feasibility* ensures that the system will be developed within the specified time duration.

Design:

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define overall system architecture. In the initial stage of design, basic outline plan for the entire system is created. Once the basic architecture is designed, detailed design of the system is started in which each module is designed in detail.

Coding:

Once the system design phase is over, the next phase is coding. In this phase, developers start building the entire system by writing code using the chosen programming language. In the coding phase, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software

Development Life Cycle process. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.

Testing:

Once the software is complete, and it is deployed in the testing environment. The testing team starts testing the functionality of the entire system. This is done to verify that the entire system works according to the customer requirement. Individual modules are tested first. After that, all modules are integrated together and entire system is being tested.

Implementation and Maintenance:

Once the software testing phase is over and no bugs or errors left in the system then the final deployment process starts. Based on the feedback given by the project manager, the final software is released and checked for deployment issues if any. The software is delivered to the client and installed on the specified hardware platform and also provides maintenance services

11.5 SDLC MODELS

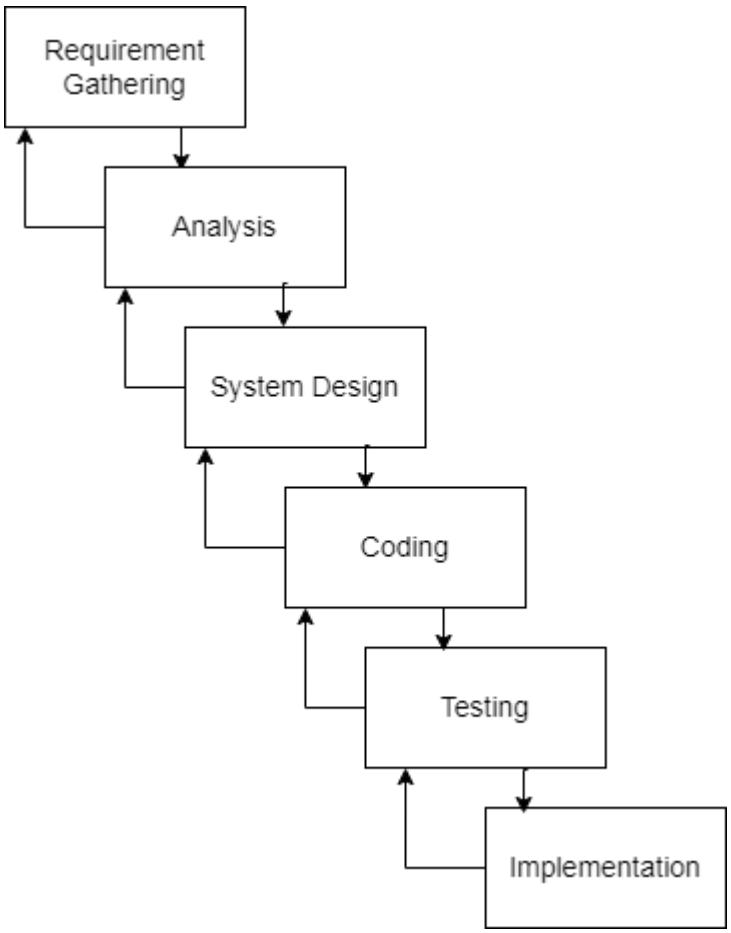
There are different software development life cycle models used in software development process. Each process model follows a series of phase unique according to its type in the step of software development.

- Waterfall Model
- Spiral Model
- V-process model

11.5.1 Waterfall Model:

The waterfall model is the most commonly used software development process model. It consists of series of activities with some arrows pointing downwards and some pointing upwards. A project starts with an initial phase and once it is completed, it moves to the next phase. The down arrow indicates that when one activity is

completed, next activity can be started. The back arrow indicates that if any changes are required than it can be performed only in the previous activity.



Waterfall Model

- **Requirement Gathering**: During this phase, requirements from the client are collected. There are various requirements gathering techniques through which customers' requirement are collected.
- **Analysis:** Once the requirements are collected, there are analysed and SRS is generated from it. Different flow graphs are prepared.
- **System Design:** During this phase, basic outline plan for the entire system is prepared. And then each module and forms are designed in detail with proper validations.
- **Coding:** After completion of design phase, the programmers write code with the specified programming language.
- **Testing:** Testing is the activity in which different modules are tested first (unit test). Once unit testing is done, the testing process proceeds to the integrated

system in which all modules are combined and the entire system is tested again.

- **Implementation:** This is the last phase of waterfall model where the system is deployed onto some hardware platform.

Advantages:

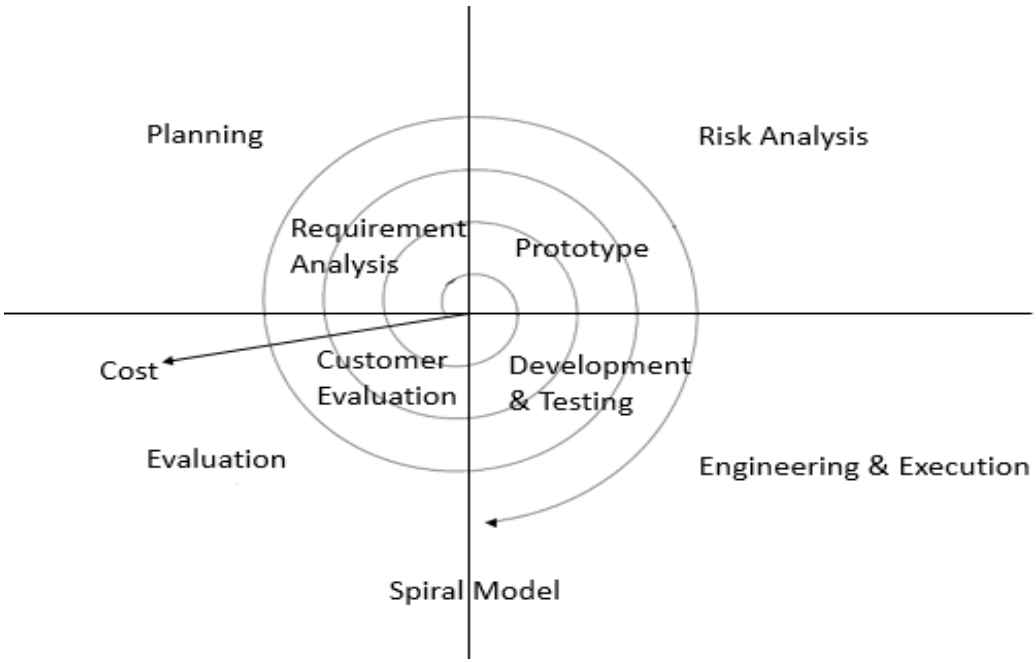
- ✓ Simple and easy to understand and use
- ✓ Works well for smaller projects where requirements are very well understood.
- ✓ Phases are processed and completed one at a time.

Disadvantages:

- × High amounts of risk and uncertainty.
- × Cannot accommodate changing requirements.

11.5.2 Spiral Model:

The spiral model is an iterative model in which the requirement gathering, design, coding and testing are performed iteratively till all requirements are met. A software project repeatedly passes through these phases in iterations called Spirals. Using the spiral model, the software is developed in a series of incremental releases. During the early iterations, the additional release may be a paper model or prototype. During later iterations, more and more complete versions of the engineered system are produced. The spiral model has four phases.



Advantages:

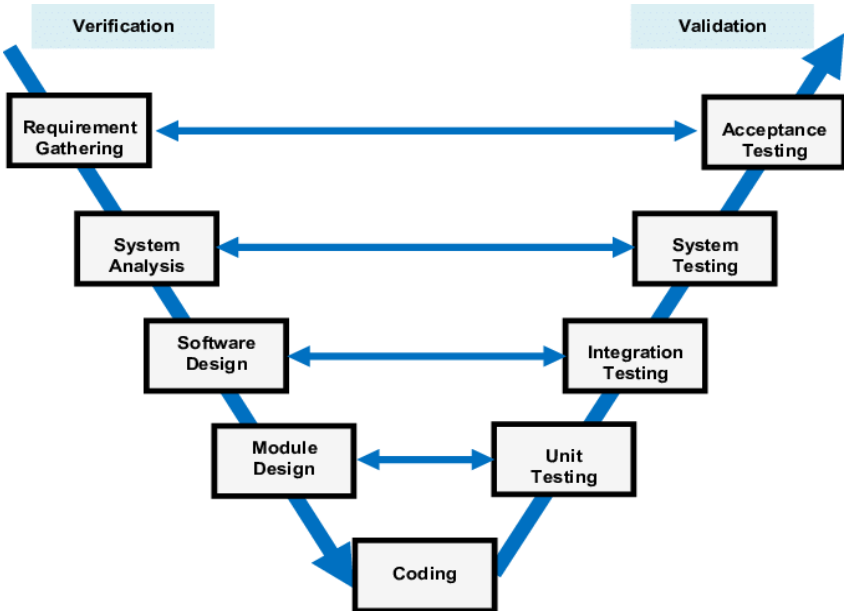
- ✓ Changing requirements can be accommodated.
- ✓ Requirements can be captured more accurately.
- ✓ Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.

Disadvantages:

- × Not suitable for small or low risk projects and could be expensive for small projects.
- × Process is complex
- × Spiral may go on indefinitely.

11.5.3 V-Process Model:

V-Model also referred to as the Verification and Validation Model. In this, each phase of SDLC must complete before the next phase starts. It follows a sequential design process same as the waterfall model.



There are the various phases of V-model:

- Requirement gathering: This is the first step where product requirements understood from the customer's side. This phase contains detailed communication to understand customer's expectations and exact requirements.
- System Design: In this stage system engineers analyse and interpret the business of the proposed system by studying the user requirements document.
- Architecture Design: The baseline in selecting the architecture is that it should understand all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology detail, etc. The integration testing model is carried out in a particular phase.
- Module Design: In the module design phase, the system breaks down into small modules. The detailed design of the modules is specified, which is known as Low-Level Design
- Coding: After designing, the coding phase is started. Based on the requirements, a suitable programming language is decided. There are some guidelines and standards for coding.

Advantages:

- ✓ Simple and easy to understand and use.
- ✓ Works well for small plans where requirements are easily understood.

Disadvantages:

- × Not suitable for a complex project.
- × Once an application is in the testing stage, it is difficult to go back and change its functionality.

11.6 Quality Concepts

Quality: Quality is all about meeting the needs and expectations of customers with respect to functionality, design, reliability, durability, & price of the product.

Software Quality: Quality software refers to software which is reasonably bug or defect free, is delivered in time and within the specified budget, meet the requirements and/or expectations, and is maintainable. In the software engineering context, software quality reflects both functional quality as well as structural quality.

Software Functional Quality − It reflects how well it satisfies a given design, based on the functional requirements or specifications.

Software Structural Quality − It deals with the handling of non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, and the degree to which the software was produced correctly.

Quality Assurance – It is defined as a procedure to ensure the quality of software products or services provided to the customers by an organization. Quality assurance focuses on improving the software development process and making it efficient and effective as per the quality standards defined for software products. Quality Assurance is popularly known as QA Testing.

Quality Control - Quality control popularly abbreviated as QC. It is a Software Engineering process used to ensure quality in a product or a service. It does not deal with the processes used to create a product; rather it examines the quality of the “end products” and the final outcome.

11.7 VERIFICATION AND VALIDATION

Verification:

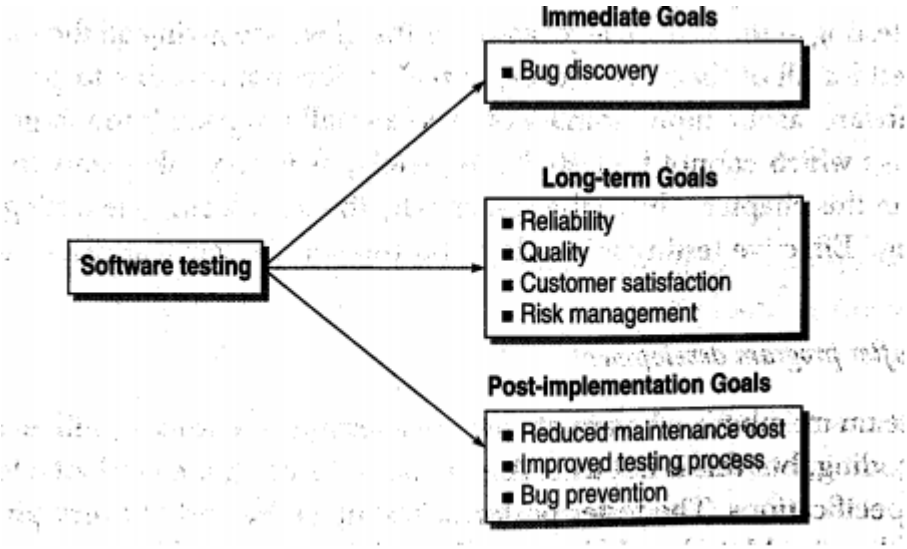
Verification is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of verification process is to ensure quality of software application, design, architecture etc. Verification activities are performed at the end of each phase of SDLC. It is also known as static testing, where we are ensuring that "**we are developing the right product or not**". The verification process involves activities like reviews, walk-through and inspection.

Validation:

Validation is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. Validation activities are performed at the end of one cycle of SDLC. Validation testing is also known as dynamic testing, where we are ensuring that **"we have developed the product right."** The validation process includes activities like black box testing, white-box testing.

11.8 GOALS OF SOFTWARE TESTING

Software testing is the process of finding errors. It helps in finalizing the software application or product against business and user requirements. The goals of software testing may be classified into three major categories:



11.8.1) Short-term or immediate goals:

These goals are immediate results after performing testing. These goals may be set in the individual phases of SDLC.

Bug discovery: The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.

11.8.2) Long-term goals:

These goals affect the product quality in the long run when one cycle of the SDLC is complete. Some of them are discussed here.

- **Quality:** Since the software is also a product, its quality is primary from the users' point of view.
- **Reliability:** Reliability is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in the reliability, in turn, increases the quality
- **Customer satisfaction:** the prime concern of testing is customer satisfaction only. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements, which are otherwise understood. A complete testing process achieves reliability, which enhances the quality, and quality, in turn, increases customer satisfaction

- **Risk management:** Risk is the probability that undesirable events will occur in a system. Software testing may act as a control, which can help in eliminating or minimizing risks.

11.8.3) Post-implementation goals:

These goals are important after the product is released.

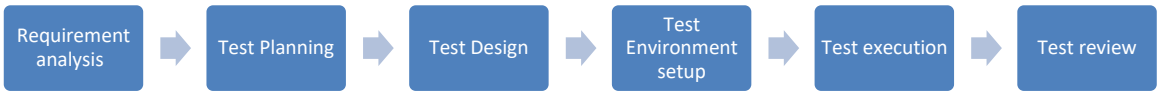
- **Reduced maintenance cost:** The maintenance cost' of any software product is not its physical cost, as the software does not wear out. If testing has been done effectively, then the chances of failure are minimized and, in turn, the maintenance cost is reduced.
- **Improved software testing process:** A testing process for one project may not be successful and there may be scope for improvement. Thus, the long-term post-implementation goal is to improve the testing process for future projects.
- **Bug prevention:** It is the consequent action of bug discovery. Though errors cannot be prevented to zero, they can be minimized. Thus, bug prevention is a superior goal of testing.

11.9 SOFTWARE TESTING LIFE CYCLE (STLC)

Software Testing Life Cycle (STLC) is a sequence of specific activities conducted during the testing process to ensure software quality goals are met. STLC involves both verification and validation. This systematic execution of each step will result in saving time and effort. The major contribution of STLC is to involve the testers at early stages of development. The STLC also helps the management in measuring specific milestones.

STLC consists of following phases:

- Requirement Analysis
- Test Planning
- Test Design
- Test execution
- Test reviews



1. Requirement Analysis: When the SRD is ready and shared with the stakeholders, the testing team starts high level analysis concerning the AUT (Application under Test). System requirements include functional and non-functional specifications, both of which present opportunities to test and validate. Requirement analysis often includes brainstorming sessions, identifying blind spots or unclear areas in the requirements, and prioritizing certain assessments.

2. Test Planning: The goal of test planning is to take into account the important issues of testing strategy like resources, schedules, responsibilities, risks and priorities. Following activities are performed during test planning.

2.1 Defining the test strategy

2.2 Estimating number of test cases, their duration and cost

2.3 Planning the resources required like tools and documents required

2.4 Identifying areas of risks

2.5 Defining the test completion criteria

After test planning, *test plan* and *effort estimation* document will be the deliverables.

3. Test Design: One of the major activities in testing is the design of test cases. It is a well-planned process. It involves the creation, verification and rework of test cases & test scripts after the test plan is ready. A good test case is one that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority. Thus, the test design phase includes developing test objectives, identifying test cases and creating their specifications and then developing test case procedure specification.

After test design, *test cases* and *test data* will be the deliverables.

4. Test Environment Setup: The test environment provides the setting where the actual testing occurs. This is crucial software testing life cycle phase, and it requires help from other members of the organization. Testers must have access to bug

reporting capabilities, as well as the application architecture to support the product. Without these elements, testers might not be able to do their jobs.

After this stage, *smoke test* (a software testing process that determines whether the deployed software build is stable or not.) results are deliverables.

5. Test Execution: In this phase, all test cases are executed including verification and validation. Verification test cases start at the end of each phase of SDLC. Validation test cases start after the completion of a module. The process consists of test script execution, test script maintenance and bug reporting. If bugs are reported then it is reverted back to development team for correction and retesting will be performed.

After test execution, *defect reports, test log, summary reports* are deliverables.

6. Test Reviews/Test Closure: This phase is completion of test execution which involves several activities like test completion reporting, collection of test completion matrices and test results. Testing team members meet, discuss and analyze testing artifacts to identify strategies that have to be implemented in future.

After test closure, *test closure (summary) report* and *test metrics* are deliverables.

11.10 STATIC AND DYNAMIC TESTING

Static Testing:

Static Testing is a type of software testing in which software application is tested without code execution. It is called as static because we never execute the code in this method. Manual or automated reviews of code, requirement documents and document design are done in order to find the errors. The main objective of static testing is to improve the quality of software applications by finding errors in early stages of software development process. Static testing can be done manually or with the help of tools to improve the quality of the application by finding the error at the early stage of development; that why it is also called the verification process. The documents review, high and low-level design review, code walkthrough take place in the verification process.

Advantages of static testing:

- ✓ It is a fast and easy way to find and fix the errors

- ✓ With automated tools, it is quick and easy to review the software
- ✓ It finds errors at the early stage of SDLC which reduces cost also
- ✓ With automated tools, it takes less time to check the system

Disadvantages of static testing:

- × It is time consuming process if done manually
- × Automated tools may work with few programming languages as well as only scan the code

Dynamic Testing:

Dynamic Testing, a code is executed. It checks for functional behavior of software system, memory/CPU usage and overall performance of the system. The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This testing is also called validation testing. It executes the software and validates the output with the expected outcome. It is performed at all levels of testing and it can be either black or white box testing. It is used to check whether the application or software is working fine during and after the installation of the application without any error.

Advantages of dynamic testing:

- ✓ It finds bugs which are not discovered during static testing
- ✓ It identifies vulnerabilities in a runtime environment
- ✓ It can be applied with any application

Disadvantages of dynamic testing:

- × It requires time and cost to check the system

Difference between static and dynamic testing:

	Static Testing	Dynamic Testing
1.	Code is not executed	Code is executed
2.	It means review and examine software	It means running and testing the software
3.	It performs verification process	It performs validation process
4.	It is performed before compilation	It is performed after compilation

5.	It is about prevention of defects	It is about finding and fixing defects
6.	It takes less time to check the system	It takes more time to check the system
7.	It is done at early stage of SDLC	It is done after completion of one cycle of SDLC
8.	It is more cost-effective method	It is less cost-effective method
9.	Inspection, walkthroughs and code reviews are methods of static testing	Black-box and white-box testing are methods of dynamic testing

Exercise:1 Fill in the blanks

1. Testing is the process of _____ errors.
2. _____ testing checks the system without executing the code.
3. _____ testing requires knowledge of a programming language.
4. _____ activities are performed at the end of one cycle of SDLC.
5. Bug discovery is the _____ goal of software testing.

11.11 Let us sum up

In this chapter we have learnt basic concepts of software testing. We have also seen how the software development life cycle is used in software development. Along with these, we have seen different terms related to software quality. We learnt goals of software testing. We have seen how STLC is used in software testing.

11.12 Check your progress: Possible Answers

Exercise: 1

1. Waterfall Model
2. Software development life cycle
3. Verification, Validation
4. Long-term goals of software testing
5. Static & dynamic testing

11.13 Further Reading

1. Software Testing: Principles and Practices by Naresh Chauhan (OXFORD)
2. Software Testing: Principles and Practices by Srinivasan Desikan, Gopalaswamy Ramesh(PEARSON)
3. Software Engineering: A Practitioner's Approach by Roger S. Pressman(Mc Graw Hill Education)

11.14 Activities

1. Give difference between Verification and Validation.
2. Differentiate Waterfall and spiral model.

Unit 12: Black-Box Testing 12

Unit Structure

12.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about black-box testing
- Know why, when and how black-box testing is done
- Understand black-box testing methods
- Learn how to create test cases using different methods

12.2 INTRODUCTION

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This method considers only the functional requirements of the software or a module. That's why it is also called *functional testing*. It is obvious that in black-box testing, test cases are designed based on the functional specifications. Input test data is given to the system, which is a black-box to the tester, and results are checked against expected outputs after executing the software. In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.

12.3 WHAT IS BLACK-BOX TESTING

Black-box testing is one type of dynamic testing, taking no notice of its internal structure. It only focuses on the expected output. So, to implement black-box testing, knowledge of programming language is not required. So, it is also called *Closed-box* technique. Black-box testing is done from the customer's viewpoint. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. This makes it possible to identify how the

system responds to expected and unexpected user actions. It is also known as *Behavioral Testing or functional testing*.

Example:

Let us consider the login screen of any website. It includes two fields' username and password. The tester would check these two fields with the required set of inputs and outputs only. The tester would not check any logic behind this screen that how it is implemented. That is called black-box testing.

12.4 NEED FOR BLACK-BOX TESTING

Black box testing helps in the overall functionality verification of the system under test. It helps in identifying any incomplete, inconsistent requirements as well as any issues involved when the system is tested. It also addresses the stated requirements as well as implied requirements. Black-box testing handles valid and invalid inputs. Because it is not sufficient to simply handle valid inputs. It encompasses the end user perspectives as it is integral part of black-box testing. The tester may or may not know the technology or the internal logic of the product. However, knowing the technology and system internals helps in constructing test cases specific to the error prone areas.

Finally, we can summarize that black-box testing considers following aspects,

- It is done based on requirements
- It addresses stated as well as implied requirements
- It handles valid and invalid inputs
- It is done from the end user perspectives

12.5 ADVANTAGES & LIMITATIONS OF BLACK-BOX TESTING

Advantages of Black-box testing:

- ✓ Test cases can be designed as soon as the functional specifications are complete.
- ✓ Tests will be done from an end user's point of view, because the end user should accept the system.
- ✓ Tester can be non-technical.
- ✓ No knowledge of programming language is required.
- ✓ Efficient when used on large systems.
- ✓ Since the tester and developer are independent of each other, testing is balanced
- ✓ Code access is not required.

Disadvantages of Black-box testing:

- × Test cases are difficult to design without having clear functional specifications.
- × It is difficult to identify all possible inputs in limited testing time. As a result, writing test cases may be slow and difficult.
- × It is difficult to identify tricky inputs if the test cases are not developed based on specifications.
- × It is not possible to test all the functionalities of the application with the help of black box testing.

12.6 WHEN TO DO BLACK BOX TESTING

Black-box testing activities require involvement of the testing team from the beginning of the software project life cycle. Testers can get involved right from the requirements gathering phase for the system under test. Test scenarios and test data are prepared during the test construction phase of the test cycle, when the software is in the design phase.

Once the code is ready and delivered for testing, test execution can be done. All the test scenarios developed during the construction phase are executed. Since, only the functional requirements are tested, black box testing is started after the system or a module is designed.

Once the system is designed, black-box testing can be carried out by using various techniques. The following sections defines different methods of black-box testing.

12.7 BOUNDARY VALUE ANALYSIS (BVA)

An effective test case design requires test cases to be designed such that they increase the probability of finding errors. Boundary value analysis solves this issue. It has been observed that test cases designed with boundary input values have a high chance to find errors. BVA is applicable when the module to be tested is a function of several independent variables. BVA is considered a technique that uncovers bugs at the boundary of input values. Here, boundary means the minimum and maximum value taken by the input domain.

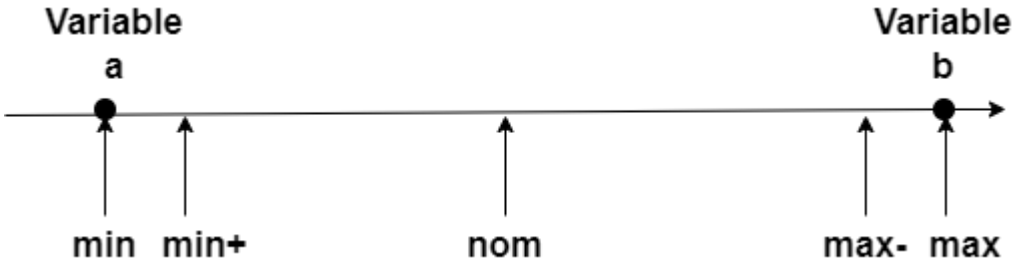
For example, if A is an integer between the range 10 to 100, then boundary checking can be done on 10(9,10,11) and on 100(99,100,101).

Boundary Value Analysis offers several methods to design test cases.

12.7.1 Boundary value checking (BVC):

One of the methods of BVA is Boundary Value Checking (BVC). In BVC, the test cases are designed by holding one variable at its extreme value and the other variable at its nominal value in the input domain. The variables can consider values at:

- Minimum value (Min)
- Value just above the minimum value (Min+)
- Nominal value
- Maximum value (Max)
- Value just below the maximum value (Max-)



It can be generalized that for n input variables in a module, 4n+1 test-case can be designed using Boundary Value Checking method.

Example:

A program reads an integer number within range [1,100] and checks whether it is odd number or even. So, the boundary value checking can be performed as follows:

Since there is only one variable, the total number of test cases will be 4n+1=5. The set of minimum and maximum values can be,

Minimum value (Min) = 1

Above the minimum value (Min+) = 2

Maximum value (Max) = 100

Value below maximum value (Max-) = 99

Nominal value = 50 – 55

Boundary Value Analysis		
Minimum (Min, Min+)	Nominal	Maximum (Max, Max-)
1,2	50-55	100, 99

Using these values, following test cases can be designed using BVC.

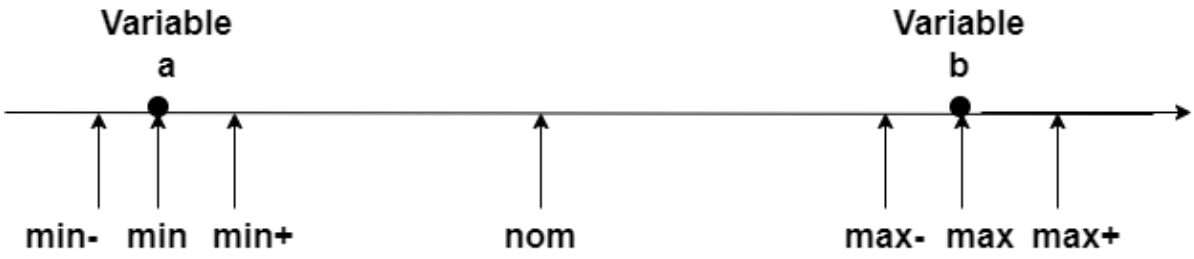
Test case Id	**Integer variable**	**Expected output**
1	1	Odd Number
2	2	Even Number
3	99	Odd number

4	100	Even number
5	52	Even number

12.7.2 Robustness Testing Method:

The idea of BVC can be extended such that the boundary values can be selected as,

- Value just below the minimum value(Min-)
- Minimum Value(Min)
- Value just above the minimum value(Min+)
- Nominal value
- Maximum value(Max)
- Value just below the maximum value(Max-)
- Value just above the maximum value(Max+)



It can be generalized that for n input variables in a module, 6n+1 test-case can be designed using robustness testing method.

Example: A program reads an integer number within range [1,100] and checks whether it is odd number or even. So, the robustness testing can be performed as follows:

Since there is only one variable, the total number of test cases will be 6n+1=7. The set of minimum and maximum values can be,

Minimum value (Min) = 1

Above the minimum value (Min+) = 2

Below the minimum value (Min-) = 0

Maximum value = 100

Value below maximum value = 99

Value above maximum value = 101

Nominal value = 50 – 55

Boundary Value Analysis		
Invalid (min-1)	Valid (min, min+, max-, max)	Invalid (max+)
0	1,2,99,100	101

Using these values, following test cases can be designed using robustness testing.

Test case Id	**Integer variable**	**Expected output**
1	0	Invalid input
2	1	Odd number
3	2	Even number
4	99	Odd number
5	100	Even number
6	101	Invalid input
7	52	Even number

12.8 EQUIVALENCE CLASS PARTIONING

Equivalence partitioning is a method for deriving test cases where classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. So, instead of testing every input, only one test case from each class can be executed. It means only one test case in the equivalence class is sufficient to find errors. If one test case in equivalence class detects a defect, then all other test cases in that class have the same probability of finding defects. So, instead of taking every value in one domain,

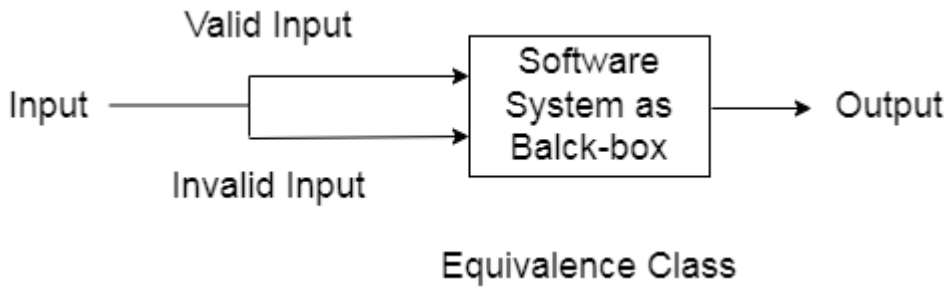
only one test case is selected from one class. In this way, it reduces the total number of test cases. Similarly, it is a way to find maximum errors with the small number of test cases.

Equivalence class partitioning method has following two goals:

- Completeness: we can complete the testing domain without executing all the test cases.
- Non-redundancy: The goal of equivalence partitioning method is to reduce redundant test cases.

Two types of classes can be identified.

- Valid equivalence class: These classes consider valid inputs to the program.
- Invalid equivalence class: These classes consider invalid inputs that will generate error conditions.



Equivalence Class

Guidelines for forming equivalence classes:

- If the requirement is a range of values, then derive the test case for one valid and two invalid inputs.
- If the requirement is a set of values, then derive the test case for one valid and two invalid inputs.
- If the requirement id Boolean (true/false), then derive the test case for both true/false values.
- Boundary value analysis can help in identifying the classes.

Advantages of equivalence class testing:

- ✓ With the help of equivalence class testing, the number of test cases greatly reduces maintaining the same test coverage.

- ✓ This testing technique helps in delivering a quality product within a minimal time period.
- ✓ It is perfectly suitable for software projects with time and resource constraints.
- ✓ It is process-oriented.

Disadvantages of equivalence class testing:

- × In the case of complex applications, it is very difficult to identify all set of equivalence classes
- × All necessary inputs may not cover.
- × The whole success of equivalence class testing relies on the identification of equivalence classes.

Example:

A program accepts 10-digit mobile numbers n from user and prints the success of error message. Design test cases using equivalence class testing.

Here, we have three different test scenarios.

Equivalence Class testing		
Invalid (Number n < 10)	Valid (Number = 10)	Invalid (Number n >10)

Class A1 = {<n>: n<10}

Class A2 = {<n>: n=10}

Class A3 = {<n>: n>10}

The test cases designed using equivalence class partitioning method are,

Test Case Id	**Input variable n**	**Expected Output**	**Class covered**
1	123456789	Error!!!	A1
2	1234567890	Success	A2
3	1234567890123	Error!!!	A3

12.9 DECISION TABLE BASED TESTING

Boundary value analysis and equivalence class testing methods do not consider combinations of input conditions. There may be some critical behaviour to be tested when some combinations of input conditions are considered.

Decision table is another useful method to represent the information in a tabular format. Decision table helps to check all possible combinations of conditions for testing and testers can also identify missed conditions easily. This is a systematic approach where the different input combinations and their corresponding system behaviour (Output) are captured in a tabular form. That is why it is also called as a *Cause-Effect table* where Cause and effects are captured for better test coverage.

Decision Table Testing is Important because it helps to test different combinations of conditions and provides better test coverage for complex business logic. When testing the behaviour of a large set of inputs where system behaviour differs with each set of input, decision table testing provides good coverage and the representation is simple so it is easy to interpret and use.

12.9.1 Advantages of Decision table-based testing:

- ✓ The representation of decision table is simple so that it can be easily interpreted.
- ✓ It helps to make effective combinations and can ensure a better coverage for testing.
- ✓ Any complex business conditions can be easily turned into decision tables
- ✓ Decision table testing is the most preferred black box testing and requirements management.
- ✓ Decision tables guarantee coverage of all possible combinations of condition values.
- ✓ Decision tables are easy to understand.
- ✓ Multiple conditions, scenarios and results can be viewed and analysed on the same page by both developers and testers.

12.9.2 Disadvantages of decision table-based testing:

- × Decision tables only present a partial solution
- × Do not depict the flow of logic of a solution
- × Decision tables are quite far away from high level languages
- × When there are too many alternatives, decision table cannot list them all.

12.9.3 Formation of decision table:

A decision table is composed of following components.

- **Condition Stub:** It is a list of conditions upon which decisions are depend. It displays different input conditions.
- **Action Stub:** It is a list of resulting actions performed based on the input condition.

		Case 1	**Case 2**	**Case 3**	**Case 4**	**-------**
Condition Stub	Condition 1	True	False	False	True	
	Condition 2	False	False	True	True	
Action Stub	Action 1	✓	✓			
	Action 2			✓	✓	

- **Condition Entry:** It is a specific entry in the table based on the condition. It takes only two values – TRUE or FALSE which is called a *Case* or a *Rule.*
- **Action Entry:** It is the entry in a table for the resulting actions performed when one rule is satisfied.

12.9.4 Guidelines to develop a decision table:

To create a decision table, following rules are followed:

- Define all conditions in the condition stub.
- Define all the actions in the action stub.
- Associate specific sets of conditions with specific actions.
- Define rules by indicating list of actions for the set of conditions.

12.9.5 Test case design using decision table:

While creating test cases, following interpretation should be done:

- Condition stub interprets as input for the test case
- Action stub interprets as output for the test case
- Rule or Case becomes the test case itself

Example:

A program accepts username and password. If both are correct than home page is displayed. If one of them is wrong then error message should be generated. Design decision table and test cases for the given case.

Solution: The condition here is that the user will be redirected to the homepage if he enters the correct username and password, and an error message will be displayed if the input is wrong.

		Case 1	**Case 2**	**Case 3**	**Case 4**
Condition Stub	Username=”abc”	T	T	F	F
	Password = “xyz”	T	F	T	F
Action Stub	Home Page	✓			
	Error Message		✓	✓	✓

Decision Table Interpretation:

Case 1: Username and Password both are wrong, and the user is shown an error message.

Case 2: Username is correct, but the password is wrong, and the user is shown an error message,

Case 3: The username is wrong, but the password is correct, and the user is shown an error message.

Case 4: Username and password both are correct, and the user is taken to the homepage.

Test cases:

Test case Id	Username	Password	Expected output
1	Abc	Xyz	Home Page
2	Abc	abc	Invalid input
3	Xyz	Xyz	Invalid input
4	Xyz	abc	Invalid input

12.10 DIFFERENCE BETWEEN BVA & EQUIVALENCE CLASS TESTING

Boundary Value Analysis	Equivalence Class Testing
Boundary values are those that contain the upper and lower limit of a variable.	It is a technique where the input data is divided into partitions of valid and invalid values.
It will help decrease testing time due to a lesser number of test cases	The Equivalence partitioning will reduce the number of test cases
The Boundary Value Analysis is often called a part of the Stress and Negative Testing.	The Equivalence partitioning can be suitable for all the software testing levels such as unit, integration, system.
It it suitable for large applications	It is suitable for applications which have time and resources constraints
BVA can be performed using boundary value checking and robustness testing	Equivalence class can be performed using valid and invalid input classes.

Exercise:1 Fill in the blanks

1. _____ testing does not check internal structure of the system.
2. Using _____, valid and invalid input classes are created.
3. Black-box testing is also called _____.
4. _____ test cases are there in BVC if there are 5 variables in a module.
5. Condition stub and action stub are components of _____.

12.11 Let us sum up

In this chapter we have learnt about black-box testing. We have seen how it is performed and different methods to implement black-box testing. We have also discussed about test cases and how to create a test case using black-box testing methods.

12.12 Check your progress: Possible Answers

Exercise: 1

1. Boundary Value Analysis
2. Equivalence class partitioning
3. Decision table based testing
4. Robustness testing method

12.13 Further Reading

1. Software Testing: Principles and Practices by Naresh Chauhan (OXFORD)
2. Software Testing: Principles and Practices by Srinivasan Desikan, Gopalaswamy Ramesh(PEARSON)
3. Software Testing: A Craftsman's Approach by Paul C. Jorgensen (AUERBACH)

12.14 Activities

1. A program reads an integer numbers A within the range [1, 50] and checks whether the number is prime or not. Design test cases using BVC and robustness testing method.
2. A program calculates the total salary of an employee with the condition that the working hours are less than or equal to 8 then give normal salary. For the hours over 8, the salary is calculated at the rate of 1.25 of the salary. Design test cases using decision table-based testing.
3. A program reads an integer number A within the range [1, 50] and checks for the positive, negative or zero number. Design test cases using equivalence class partitioning.

Unit 13: White-Box Testing 13

Unit Structure

13.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know about white-box testing
- Understand need of white-box testing
- Know different methods of white-box testing
- Learn to create test cases using white-box testing methods

13.2 INTRODUCTION

Black-box testing does not check internal logic of the program. It has various limitations because there are some bugs which are not revealed by black-box testing. Black-box testing requires testing the functionality of the system. But, to implement black-box testing, first that functionality should be coded and tested. Thus, the test cases for black-box testing can be designed earlier than white-box testing, but they cannot be executed until the code is produced and checked using white-box testing.

13.3 WHITE-BOX TESTING

White-box testing is another effective method of dynamic testing. White Box Testing is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output.

In white box testing, code is visible to testers so it is also called *Clear box testing, open box testing, transparent box testing, Code-based testing* and *Glass box testing*. The tester chooses inputs to exercise paths through the code and determine the expected outputs. In white-box testing, structure means the logic of the program, which has been implemented in the language code. The purpose is to test the logic of the system. It ensures that the internal parts of the software are also tested.

13.4 NEED OF WHITE-BOX TESTING

White-box testing is an essential part of software testing. The supporting reasons for white-box testing are as follows:

1. Some typographical errors are not observed and detected and are not covered by black-box testing techniques. White-box testing methods help to detect such errors.

2. Errors that come from the design phase will also be reflected in the code and therefore, we must execute white-box test cases for code verification.

3. One primary goal of White Box Testing is to verify the working of an application.

4. It is executed at early stages and does not wait for GUI (Graphical User Interface) availability.

5. We often believe that a logical path is not likely to be executed but it may be executed on a regular basis. White-box testing explores these paths too.

6. Since white-box testing is complementary to black-box testing, there are categories of bugs that can be revealed by white-box testing but do not through black-box testing. There may be portion of code that are not checked during executing the test cases, but these will be tested by white-box testing.

7. White-box testing is not an alternative but an essential stage of software testing because without testing code, its functionalities cannot be tested means we cannot perform black-box testing first without performing white-box testing.

13.5 HOW TO DO WHITE-BOX TESTING

Testers employing white box testing typically understand the source code and create test cases and execute. Understanding the source code involves a good working knowledge of the programming languages used in the software that is being tested. Besides, the tester should be aware of the secure coding practices as well, in order to identify security issues and prevent attacks. Also, the tester would develop tests for each process. This is often done by the developer as it requires a strong command

over the code. Other methods employed are Manual Testing and trial and error testing. It is performed in the following manner:

- Examine each source code in a program.
- Create a flow graph from the given code
- Identify possible paths and design test cases
- Execute test cases with the required input and expected output

13.6 ADVANTAGES, DISADVANTAGES OF WHITE-BOX TESTING

Advantages of white-box testing:

- ✓ It allows a finding of hidden errors, to find internal errors because it checks and works by internal functionality.
- ✓ Testing can start even without the graphic user interface
- ✓ Ease of automation is present in White Box testing.
- ✓ It can be started at an earlier stage in SDLC as it doesn't require any interface
- ✓ It requires internal knowledge to do testing that's why it helps in maximum coverage of the code.
- ✓ It helps to find issues and optimize code to adopt different techniques of White Box Testing to test a developed application or website.

Disadvantages of white-box testing:

- × It is complex, expensive, and time-consuming.
- × It requires professional resources and in-depth knowledge of software.
- × It cannot be performed without knowledge of programming language.
- × If code changes, then the test cases need to be redesigned again.

13.7 BLACK-BOX TESTING VS. WHITE-BOX TESTING

Though both black-box and white-box testing methods are type of dynamic testing, both have following differences.

White-box testing	Black-box testing
It is a technique to test internal code of the system.	It is a technique to test the functionality of the system.
The tester must have knowledge of the programming language.	It does not require knowledge of the programming language.
White-box testing is performed by software developers.	Black-box testing is performed by software testers.
It is time-consuming process.	It is less time-consuming process.
It is also called clear-box or glass-box testing.	It is also called closed-box testing.
It is generally implemented to lower levels of testing like unit, integration testing.	It is generally implemented to higher levels of testing like system and acceptance testing.
It is more exhaustive testing.	It is less exhaustive testing.
White-box testing techniques are Code coverage, path testing.	Black-box testing techniques are boundary value analysis, equivalence class and decision table-based testing

13.8 CODE COVERAGE CRITERIA

The code coverage criteria consider the program code based on the logic of the program such that every element of the logic is covered. Therefore, the intention in white-box testing is to cover the whole logic. The basic forms of code coverage are,

13.8.1 Statement coverage:

In a programming language, a statement is nothing but the line of code or instruction for the computer to understand and act accordingly. A statement becomes an executable statement when it gets compiled and converted into the object code and performs the action when the program is in a running mode.

Hence “Statement Coverage”, as the name itself suggests, it is the method of validating whether each and every line of the code is executed at least once.

Example:

```
Scanf(“%d”,&x);
Scanf(“%d”,&y);
While(x!=y)
{
        if(x>y)
                x=x-y;
        else
                y=y-x;
}
Printf(x);
Printf(y);
```

In the above program segment, if we want to cover every statement then following test cases must be designed:

Test case 1: x=y=n

Test case 2: x=n, y=n’

Test case 3: x<y

Test case 4: x>y

13.8.2 Branch Coverage:

Branch Coverage is a white box testing method in which every outcome from a code module (statement or loop) is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once. It helps to measure fractions of independent code segments and to find out sections having no branches.

Example:

```
Branch (int a) {
    If (a> 0)
        Printf (“Positive”);
    Else
        Printf (“Negative”);
```

```
}
```

In the above code segment, following test cases are designed:
Test case 1: a>0
Test case 2: a<0

13.8.3 Condition Coverage:

Condition Coverage or expression coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition. Condition coverage offers better sensitivity to the control flow than decision coverage. In this coverage, expressions with logical operands are only considered.

Example:
If(A&&B)

For the above condition, following test cases are designed:
Test case 1: A = true, B=true
Test case 2: A=true, B=False
Test case 3: A= False, B=True
Test case 4: A=False, B=False

Advantages of code coverage criteria:

- ✓ It helps to evaluate a quantitative measure of code coverage
- ✓ It allows to create extra test cases to increase coverage
- ✓ It allows to find the areas of a program which is not exercised by a set of test cases

Disadvantages of code coverage criteria:

- ✗ Code coverage is also not telling how much and how well you have covered your logic

- In the case when the specified function hasn't implemented, or a not included from the specification, then structure-based techniques cannot find that issue.
- It is not possible to determine whether we tested all possible values of a feature with the help of code coverage

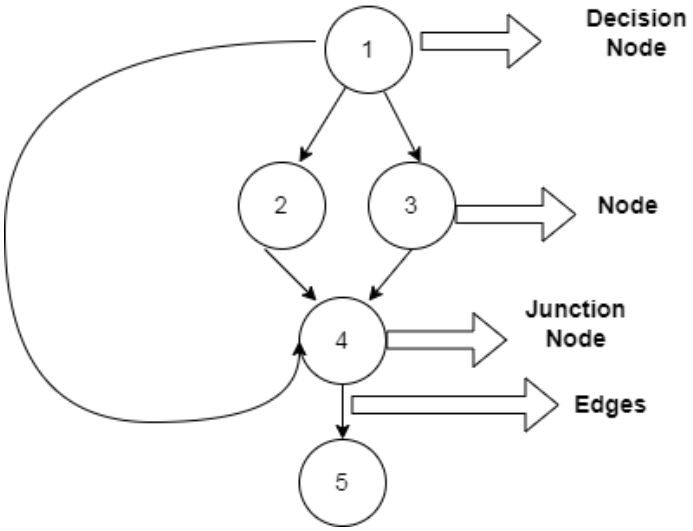
13.9 BASIS PATH TESTING

Basis path testing is the oldest structural testing method. It is based on the control structure of the program. Based on the control structure, flow graph is prepared. Path coverage is more general and useful for detecting errors. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

13.9.1 Control flow graph:

It is a graphical representation of control structure of a program. Flow graph is a directed graph. A directed graph (V, E) consists of set of vertices V and set of edges E. Following terms are used for a flow graph.

- Node – It represents one or more procedural statements and represented by a circle with number or label.
- Edges – It represents the flow of control in a program and denoted by an arrow.
- Decision node – It is a node with more than one arrow leaving.
- Junction node - It is a node with more than one arrow entering.
- Region – Areas bounded by nodes and edges are called region.



13.9.2 Path Testing Terms:

- **Path:**

  A path is a sequence of instructions that starts at the entry node and completes at the exit node.

  In the above flow graph, paths can be 1-2-4-5, 1-3-4-5 and 1-4-5.

- **Segment:**

  Path consists of segments. The smallest segment is a link which displays single process between two nodes.

  In the above flow graph, if we consider the path 1-2-4-5 then the process between 1-2, 2-4 or 4-5 is called a segment.

- **Path segment:**

  A path segment is a succession of adjacent links that belongs to some path.

  In the above flow graph, if we consider the path 1-2-4-5 then the path 1-2, 2-4 or 4-5 is called a path segment.

- **Length of path:**

  The length of path is number of links in a path.

  In the above flow graph, if we consider the path 1-2-4-5 then length of the path will be 3.

- **Independent path:**

  An independent path is any path through the graph that has at least one set of processing statement.

In the above flow graph, 1-2-4-5 , 1-3-4-5 and 1-4-5 all are the independent paths.

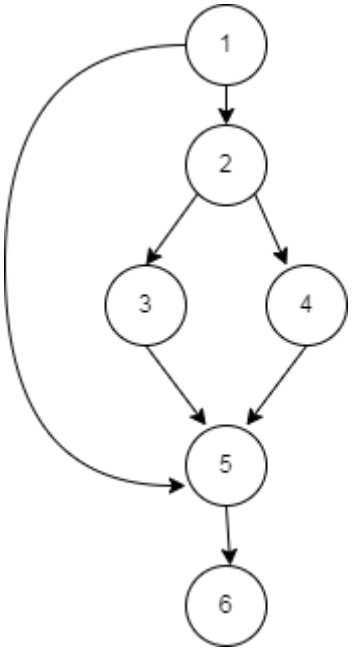
13.9.3 Cyclomatic Complexity:

Cyclomatic complexity is used to find number of independent paths in a given flow graph. An independent path is any path through a program that introduces at least one new set of processing statements (i.e., a new node) or a new condition (i.e., a new edge). The number of independent paths is given by,

V(G) = e – n + 2

This is called the cyclomatic number of a program.

Example:



For the above flow graph, number of nodes are 6 and number of edges are 7. So, the cyclomatic complexity can be,

V(G) = e – n + 2

= 7 – 6 + 2

= 3

Thus, as the cyclomatic number is 3, this flow graph will have 3 independent paths.

Independent paths = 1-2-3-5-6

1-2-4-5-6

1-5-6

13.9.4 Guidelines for Basis Path Testing:

Following steps should be performed for designing test cases using basis path testing.

- Draw flow graph using the given code.
- Calculate cyclomatic complexity from the flow graph.
- Find independent paths from the cyclomatic number.
- Based on independent path, design test cases such that each path is executed.

13.9.5 Advantages of basis path testing:

- ✓ It helps to reduce the redundant tests
- ✓ It focuses attention on program logic
- ✓ All program statements are executed and tested at least once.
- ✓ It guarantees complete branch coverage.

Example: Consider the following program segment.

```
1. int n=45;
2. if(n>18)
3. print “Eligible”
   Else
4. print “Not Eligible”
5. print n
6. End
```

a) Draw the flow graph for the program.

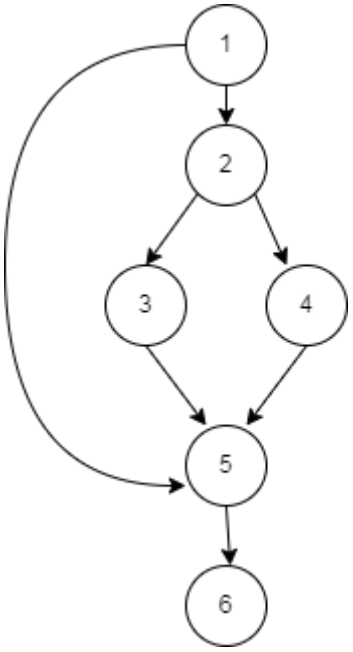
b) Calculate cyclomatic complexity

c) List all independent paths.

d) Design test cases

Solution:

a) Flow graph:



b) Cyclomatic Complexity:

Here, total number of nodes is 6 while the edges are 7. Therefore,

V(G) = e-n+2

= 7 – 6 + 2

= 3

c) Independent Paths:

From the above cyclomatic number, number of independent paths will be,

P1 = 1-2-3-5-6

P2 = 1-2-4-5-6

P3 = 1-5-6

d) Test Cases:

Test Case Id	Input number n	Expected output	Path Covered
1	45	45	P3
2	10	Not Eligible	P2
3	20	Eligible	P1

13.10 GRAPH MATRICES

Flow graph is an effective method of path testing. However, path testing with the use of flow graph may be time consuming activity. As the size of graph increases, manual path tracing becomes difficult and lead to errors. So, the idea is to develop a software tool, which will help in basis path testing.

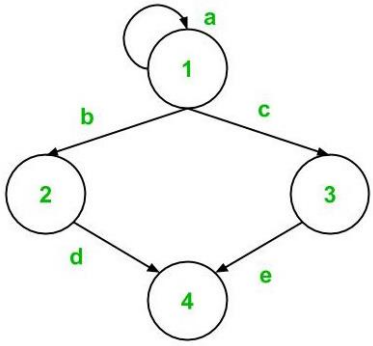
Graph matrix is the solution that can assist in developing a tool for automation of path tracing because the properties of graph matrices are test tool building.

Graph matrix:

A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in a flow graph. Each row and column identify a particular node and matrix entries represent a connection between the nodes.

Example:

Consider the following graph and represent it in the form of a graph matrix.



Solution:

The graph matrix for the given flow graph is as follows.

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

In the above graph matrix, the weight for each edge is denoted by its label. We can also create graph matrix by assigning 1 for weight.

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

Exercise:1 Fill in the blanks

1. White-box testing is also called _____.
2. A node with more than one arrow leaving is called _____.
3. Areas bounded by nodes and edges are called _____.
4. The number of independent paths can be given by _____.
5. White box testing is a type of _____ testing.

13.11 Let us sum up

In this chapter we have learnt about white-box testing. Along with these, we have seen different types of white-box testing techniques and their concepts. We have seen different statements of code coverage criteria. We learnt how basis path testing works. We have discussed how to create a test case using any one of the testing methods. Finally, we have seen how we can convert a flow graph into graph matrix.

13.12 Check your progress: Possible Answers

Exercise: 1

1. Code coverage criteria
2. Basis path testing
3. Cyclomatic complexity
4. Comparison of black box and white box
5. Graph matrix

13.13 Further Reading

1. Software Testing: Principles and Practices by Naresh Chauhan (OXFORD)
2. Software Testing: Principles and Practices by Srinivasan Desikan, Gopalaswamy Ramesh(PEARSON)
3. Software Testing: A Craftsman's Approach by Paul C. Jorgensen (AUERBACH)

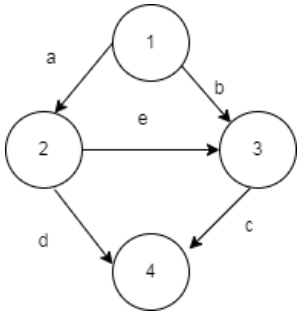
13.14 Activities

1. Consider the following program segment.

```
1. Int a=31;
2. If(a%2==0)
3.        Print “Number is even”;
          4. Else
5.        Printf “Number is odd”;
6. Print a;
7. End
```

a. Draw flow graph for the program.
b. Calculate cyclomatic complexity of the program.
c. List all independent paths.
d. Design test cases from all independent paths.

2. Consider the following graph, derive its graph matrix.



Unit 14: Levels of Testing 14

Unit Structure

14.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Know different levels of testing
- Learn different methods of testing
- Learn how to create test cases using levels of testing
- Understand how reports are used in testing

14.2 INTRODUCTION

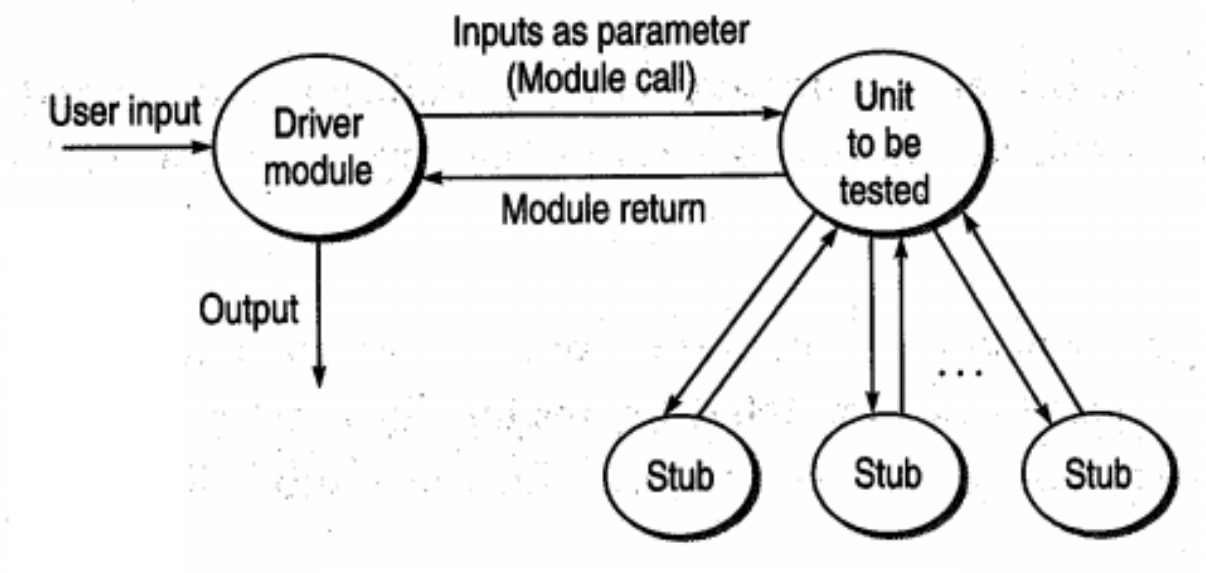
Software testing is a process of testing of software or product to ensure that the software or product is bug-free. Levels of software testing are process in which every component or small unit of software is tested. There are different levels of software testing; each has its features, advantages, and disadvantages. These different levels of software testing are designed to test the software performance and behaviours of the software at different stages. Testing levels prevent overlap and repetition between the development life cycle phases. There are different stages in the software development lifecycle like requirements, design, coding development, and execution. Levels of software testing are used to find those missing areas between these stages of the software development lifecycle.

14.3 UNIT TESTING

Unit is the smallest building block of the software system. It is the first part of the system to be validated. Before testing entire software, units or modules are tested first. Unit test ensures that the software meets at least a baseline level of functionality. The main objective of unit testing is to isolate written code to test and determine if it works as intended. Unit tests can be performed manually or automated. A manual method may have an instinctual document made detailing each step in the process; however, automated testing is the more common method to unit tests

While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing. The types of interface modules which

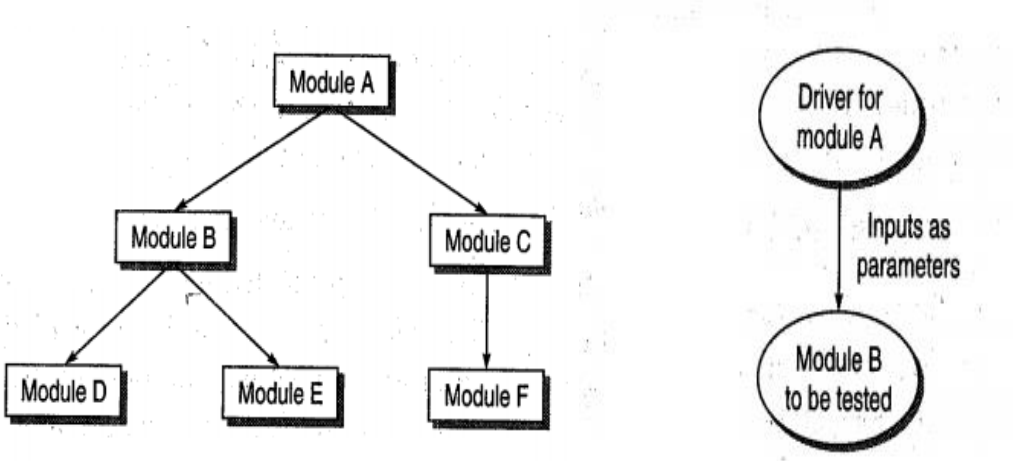
must be simulated, if required, to test a module can be either a driver or a stub.



14.3.1 Driver:

A driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution and report test results or line of code that calls a method. Suppose, the module is to be tested where some inputs are to be received from another module which is under development. In such situation, we need to simulate the inputs required in the module to be tested. This module where the required inputs for the module under test are simulated for the purpose of module is called driver module.

Example:



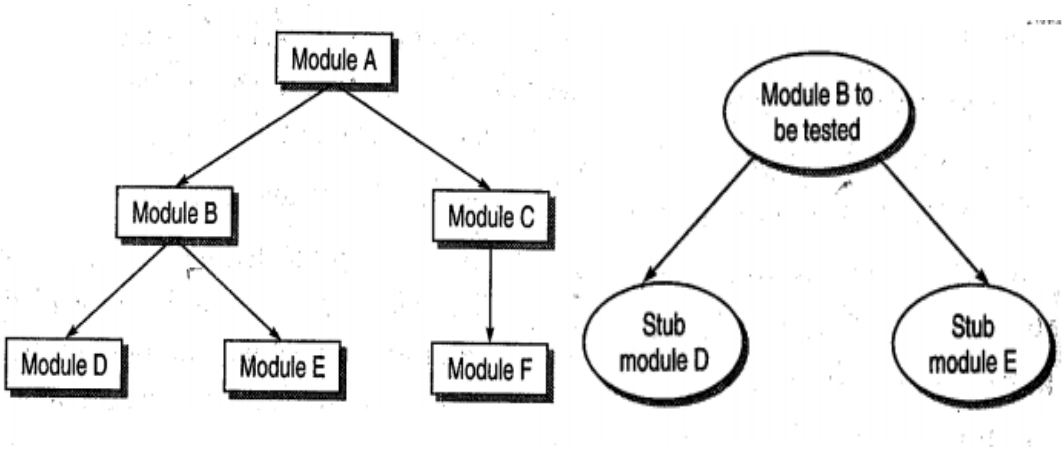
- Suppose module B is under test.
- In the hierarchy, module A is super module of module B.
- Suppose, module A is not ready and B has to be unit tested.

- In this case, module B needs input from module A.
- So, a driver module is needed which will simulate module A.

14.3.2 Stubs:

The module under testing may also call some other module, which is not ready at the time of testing. So, dummy modules are prepared for sub-ordinate modules. These dummy modules are called stubs. A stub can be defined as a piece of software that works similar to a unit and it is much simpler than the actual unit.

Example:



- In above example, module B under test needs to call module D and module E.
- They are not ready so stubs are designed for module D and E.

Characteristics of stubs:

- It doesn't perform any action of its own.
- We may include a display instruction as a message in the body of stub.
- It is a place holder for the actual module to be called.
- Stub may simulate exceptions and abnormal conditions.

14.3.3 Advantages and disadvantages of unit testing

Advantages:

- ✓ The earlier a problem is identified; the fewer compound errors occur.
- ✓ Costs of fixing a problem early can quickly outweigh the cost of fixing it later.
- ✓ Debugging processes are made easier.
- ✓ Developers can quickly make changes to the code base.

- ✓ Developers can also re-use code, migrating it to new projects.

Disadvantages:

- × Tests will not uncover every bug.
- × Unit tests only test sets of data and its functionality—it will not catch errors in integration.
- × More lines of test code may need to be written to test one line of code—creating a potential time investment.
- × Unit testing may have a steep learning curve, for example, having to learn how to use specific automated software tools.

14.4 INTEGRATION TESTING

Integration is the activity of combining the module together when all the modules have been prepared. It is a type of testing where software modules are integrated logically and tested as a group. Integration testing is done after performing unit testing. A typical software project consists of multiple software modules, coded by different programmers. The purpose of this level of testing is to expose defects in the interaction between these software modules when they are integrated.

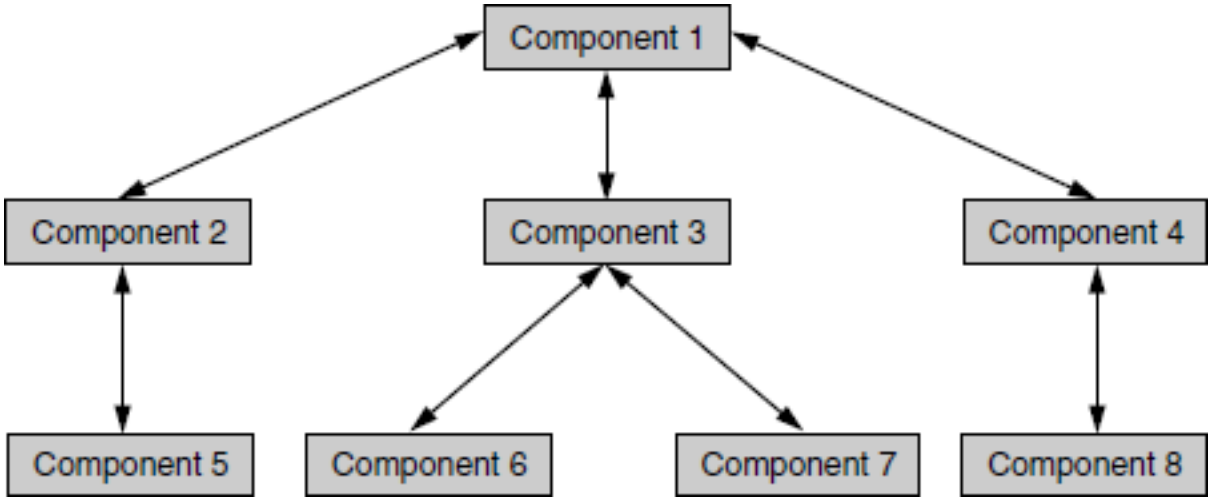
Methods of integration testing:

- Top-down Integration
- Bottom-up Integration

14.4.1 Top-down Integration:

Top-down Integration Testing is a method in which integration testing takes place from top to bottom following the control flow of software system. The higher-level modules are tested first and then lower-level modules are tested and integrated in order to check the software functionality. Stubs (Dummy modules) are used for testing if some modules are not ready.

Example:



In the above example, the integration starts with testing the interface between Component 1 and Component 2. To complete the integration testing, all interfaces covering all the arrows have to be tested together.

Step	Interfaces tested
1	1-2
2	1-3
3	1-4
4	1-2-5
5	1-3-6(7)
6	(1-2-5)-(1-3-6-(3-7))
7	1-4-8
8	(1-2-5)-(1-3-6-(3-7))-(1-4-8)

Advantages:

- ✓ Very few drivers are needed.
- ✓ Using this approach, major design flows found first.
- ✓ It is easy to localize fault.
- ✓ It is possible to obtain early prototype.

Disadvantages:

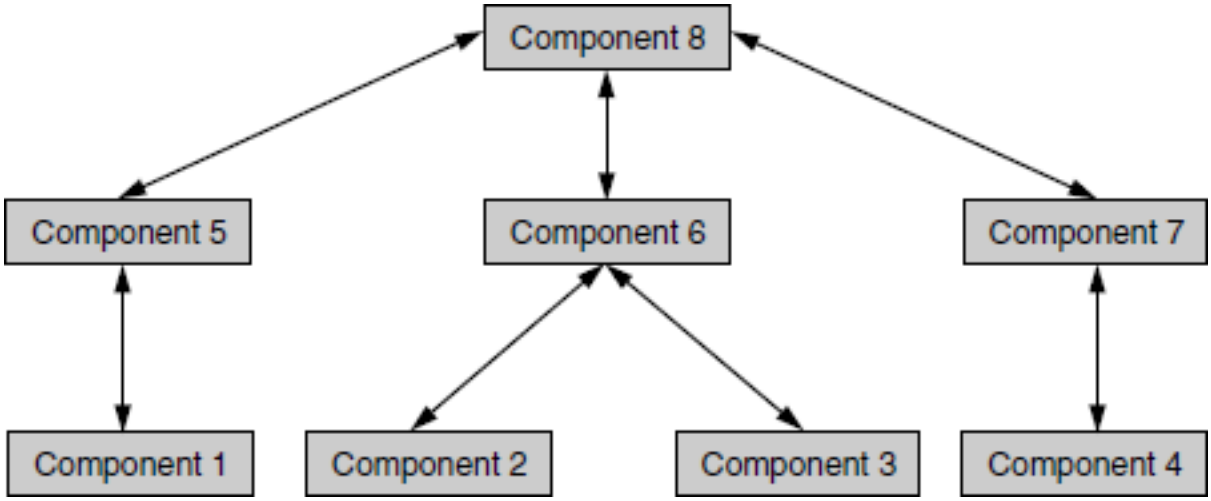
- × As it is top-down approach, more stubs are needed.

- Bottom-level modules may not be tested to the expected level or may not be tested to the requirements.

14.4.2 Bottom-up Integration:

Bottom-up Integration Testing is a strategy in which the lower-level modules are tested first. These tested modules are then further used to facilitate the testing of higher-level modules. The process continues until all modules at top level are tested. Once the lower-level modules are tested and integrated, then the next level of modules are formed.

Example:



The navigation in bottom-up integration starts from component 1 covering all sub-systems, till component 8 is reached.

Step	Interfaces tested
1	1-5
2	2-6, 3-6
3	2-6-(3-6)
4	4-7
5	1-5-8
6	2-6-(3-6)-8
7	4-7-8

8	(1-5-8)-(2-6-(3-6)-8)-(4-7-8)

Advantages:

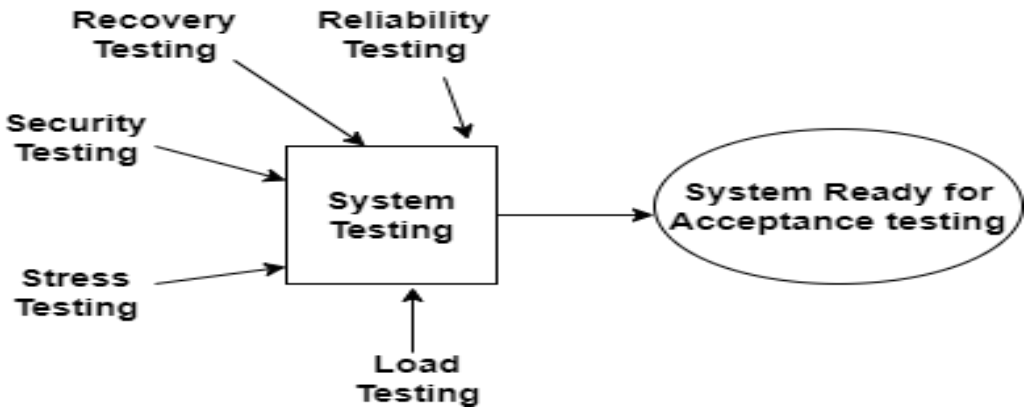
- ✓ Stubs are not required.
- ✓ Logical modules are tested thoroughly.
- ✓ Testing can be parallel with implementation
- ✓ It is easy to localize fault.

Disadvantages:

- × Drivers are necessary to call the high-level modules.
- × If the software system contains more and more small but complex modules, it may take more time for the completion of the software testing process.
- × The process of testing does not finish till all the modules of both the top and the bottom levels are included and tested.

14.5 SYSTEM TESTING

System Testing is a level of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. It is process of demonstrating a program and a system that fulfills the requirement and objectives. The software is interfaced with other software/hardware systems. It tests the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. System testing done by a professional testing agent on the completed software product before it is introduced to the market.



Types of system testing:

14.5.1 Reliability testing:

Since software is an important part of any field, the importance of reliability testing has also increased. It is a method to detect those failures earlier which are likely to appear during actual operation. The purpose of Reliability testing is to assure that the software product is bug free and reliable enough for its expected purpose.
Software reliability testing includes Feature Testing, Load Testing and Regression Testing.

a) Feature Test: The purpose of feature test is to ensure that each functionality of all the modules in the software are tested.
b) Load Test: The purpose of load test is to check the software under maximum load and breakdown.
c) Regression Test: The purpose of regression test is to recheck the software functionality whenever there is a change in it.

14.5.2 Recovery Testing:

Recovery is the ability of a system to restart the operation after the integrity of application has been lost. The purpose of recovery testing is to show that the recovery functions work properly. Recovery testing is the activity of testing how well the software is able to recover from hardware failure and other similar problems.
The time taken to recover depends upon:

- The number of restart points
- A volume of the applications

- Training and skills of people conducting recovery activities and tools available for recovery.

When there are a number of failures then instead of taking care of all failures, the recovery testing should be done in a structured fashion which means recovery testing should be carried out for one segment and then another. It is done by professional testers. Before recovery testing, adequate backup data is kept in secure locations. This is done to ensure that the operation can be continued even after a disaster.

14.5.3 Security Testing:

Security testing is used to assure the customers that their data will be protected. Security may include controlling access to data, encrypting data in communication. The main goal of Security Testing is to identify the threats in the system and measure its potential vulnerabilities. It also helps in detecting all possible security risks in the system and helps developers to fix the problems through coding.

The basic elements of security testing are,

a) Integrity: Integrity ensures that the information has not been altered by anyone other than the intended sender.

b) Confidentiality: Confidentiality protects against the disclosure of information to users other than intended recipient.

c) Non-repudiation: non-repudiation is a measure to prevent the later denial that an action happened.

d) Availability: Availability assures that the information will be ready for use when expected.

e) Authentication: Authentication is a measure designed to establish the validity of a message.

f) Authorization: It is a process of determining that the requester is allowed to receive a service.

14.5.4 Stress testing:

Stress Testing is a type of software testing that verifies stability & reliability of software application. The goal of Stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and

ensuring that software doesn’t crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions.

The goal of stress testing is to analyze the behavior of the system after a failure. For stress testing to be successful, a system should display an appropriate error message while it is under extreme conditions. To conduct Stress Testing, sometimes, massive data sets may be used which may get lost during Stress Testing. Testers should not lose this security-related data while doing stress testing. The main purpose of stress testing is to make sure that the system recovers after failure which is called as recovery.

14.5.5 Load Testing:

When a system is tested with a load that causes it to allocate its resources in maximum amount is called load testing. Load testing is testing where we check an application's performance by applying some load, which is either less than or equal to the desired load. Load testing will help to detect the maximum operating capacity of an application and any blockages or bottlenecks. It governs how the software application performs while being accessed by several users at the same time.

The load testing is mainly used to test the Client/Server's performance and applications that are web-based. In other words, we can say the load testing is used to find whether the organization used for comparing the application is necessary or not, and the performance of the application is maintained when it is at the maximum of its user load.

14.5.6 Advantages of system testing:

- ✓ Verifies the system against the business, functional and technical requirements of the end users.
- ✓ It helps in getting maximum bugs before acceptance testing.
- ✓ System testing increases the confidence level of the team in the product before the product goes for acceptance testing.
- ✓ It is the first testing level in which the whole system is under test from end to end. So, it helps in finding important defects which, unit and integration testing could not detect.

- ✓ It is a black box testing hence testers does not need programming knowledge to perform it.

14.5.7 Disadvantages of System testing:

- × System test strategy is very crucial to the success of the testing.
- × System testing starts late when all the components are ready. Hence, the cost of fixing bugs is higher.
- × The localization of the bugs is difficult as the entire system is participating in the testing.

14.6 PERFORMANCE TESTING

Performance Testing is a software testing process used for testing the speed, response time, stability, reliability, scalability and resource usage of a software application under particular workload. The main purpose of performance testing is to identify and eliminate the performance bottlenecks in the software application. Performance Testing is done to provide stakeholders with information about their application regarding speed, stability, and scalability. Performance Testing uncovers what needs to be improved before the product goes to market. Since it is non-functional testing which doesn't mean that we always use performance testing, we only go for performance testing when the application is functionally stable.

The performance testing cannot be done manually because:

- We need a lot of resources, and it became a costlier approach.
- And the accuracy cannot maintain when we track response time manually.

Following factors are tested during performance testing:

1) **Throughput:** The capability of the system or the product in handling multiple transactions is determined by a factor called throughput. Throughput represents the number of transactions that can be handled by the server
2) **Response Time:** Response time can be defined as the delay between the point of request and the first response from the product. response time represents the delay between the request and response.
3) **Latency:** Latency is a delay caused by the application, operating system, and by the environment that are calculated separately.

4) **Tuning:** Tuning is a procedure by which the product performance is enhanced by setting different values to the parameters (variables) of the product, operating system, and other components. Tuning improves the product performance without having to touch the source code of the product.
5) **Benchmarking:** The type of performance testing wherein competitive products is compared is called benchmarking.
6) **Capacity Planning:** The exercise to find out what resources and configurations are needed is called capacity planning.

14.7 ACCEPTANCE TESTING

Acceptance testing is a formal testing conducted to determine whether a software system satisfies its acceptance criteria and allows buyers to determine whether to accept the system or not. It must take place at the end of the development process. It consists of tests to determine whether the developed system meets the predetermined functionality, performance, quality. Acceptance testing is carried out by end-users. It is designed for the following reasons:

- During the development of a project if there are changes in requirements and it may not be communicated effectively to the development team.
- Developers develop functions by examining the requirement document on their own understanding and may not understand the actual requirements of the client.
- There are maybe some minor errors which can be identified only when the system is used by the end user in the actual scenario so, to find out these minor errors, acceptance testing is essential.

Acceptance test might be supported by the testers. A well-defined acceptance plan will help development teams to understand users' need. The acceptance test plan must be created or reviewed by the customer.

14.7.1 Methods of Acceptance testing:

There are two types of acceptance testing:

- **Alpha testing:**

Alpha is the test period during which the product is complete and usable in a test environment but not necessarily bug free. The main objective of alpha testing is to refine the software product by finding and fixing the bugs that were not discovered through previous tests. This testing is referred to as an alpha testing only because it is done early on, near the end of the development of the software.

Alpha testing is typically done for the two purposes:

- It gives confidence that the software is in a ready state but not released and reviewed by the customer.
- Any other major defects or performance issues should be discovered in this stage.

Since alpha testing is performed at the development site, testers and users together perform this testing.

Advantages of alpha testing:

- ✓ Better insight about the software's reliability at its early stages
- ✓ Free up your team for other projects
- ✓ Reduce delivery time to market
- ✓ Early feedback helps to improve software quality

Disadvantages of alpha testing:

- × Alpha testing does not involve in-depth testing of the software.
- × The difference between the tester's tests the data for testing the software and the customer's data from their perspective may result in the discrepancy in the software functioning.
- × The lab environment is used to simulate the real environment. But still, the lab cannot furnish all the requirement of the real environment such as multiple conditions, factors, and circumstances.

- **Beta Testing:**

  Once the alpha phase is complete, development enters the beta phase. Beta is the test period during which the product should be complete and usable. The

software is released to group of people so that further testing can be ensuring the product has few or no bugs. Beta testing is the last phase of the testing, which is carried out at the client's or customer's site.

Beta testing reduces the risk of failure and provides the quality of the product through customer validation. It is the final testing before shipping the product to the customers. Beta testing obtains direct feedback from the customers. It helps in testing to test the product in the customer's environment.

Advantages of Beta testing:

- ✓ Beta testing focuses on the customer's satisfaction.
- ✓ It helps to reduce the risk of product failure via user validations.
- ✓ Beta testing helps to get direct feedback from users.
- ✓ It helps to detect the defect and issues in the system, which is overlooked and undetected by the team of software testers.
- ✓ Beta testing helps the user to install, test, and send feedback regarding the developed software.

Disadvantages of Beta testing:

- × A software engineer has no control over the process of the testing, as the users in the real-world environment perform it.
- × This testing can be a time-consuming process and can delay the final release of the product.
- × Beta testing does not test the functionality of the software in depth as software still in development.
- × It is a waste of time and money to work on the feedback of the users who do not use the software themselves properly.

14.7.3 Difference between Alpha and Beta testing:

Alpha testing	**Beta testing**
Alpha testing performed by Testers and customers	Beta testing is performed by Clients or End Users
Alpha Testing performed at developer's site	Beta testing is performed at a client location or end user of the product

Alpha testing involves both the white box and black box techniques	Beta Testing typically uses Black Box Testing
Alpha testing is performed before beta testing	Beta testing is performed after alpha testing.
Alpha testing is the last chance to modify the system by customer	Beta testing is the phase in which the product is ready and usable
Alpha testing is more time consuming	Beta testing takes less time to test the system
Reliability and Security Testing are not performed in Alpha Testing	Reliability and Security Testing are performed in Beta Testing

14.8 TEST REPORTING

Test Report is a document which contains a summary of all test activities and final test results of a testing project. Test report is an assessment of how well the testing is performed. Based on the test report, stakeholders can evaluate the quality of the tested product and make a decision on the software release. Testing requires constant communication between the test team and other teams. There are three types of reports or communication that are required,

- Test incident reports
- Test cycle report
- Test summary reports

14.8.1 Test Incident reports:

Test incident report is a document/report generated after the culmination of software testing process, wherein the various incidents and defects are reported and logged by the team members to maintain transparency among the team members and to take important steps to resolve these issues. A test incident report is a communication that happens through the testing cycle as and when defects are encountered. A test incident report is an entry made in the defect repository. Each defect has a unique ID and this is used to identify the incident.

14.8.2 Test Cycle reports:

A test cycle report, at the end of each cycle, gives information like,

- A summary of the activities carried out during that cycle;
- Defects that were uncovered during that cycle,
- Progress from the previous cycle to the current cycle in terms of defects fixed;
- Outstanding defects that are yet to be fixed in this cycle; and
- Any variations observed in effort or schedule

14.8.3 Test Summary reports:

A report that summarizes the results of a test cycle is the test summary report. Also known as a Test Closure Report.

There are two types of test summary reports.

- Phase-wise test summary, which is produced at the end of every phase
- Final test summary reports which have all the details of all testing done by all phases and teams, also called as “release test report”.

A summary report should present,

- A summary of the activities carried out during the test cycle or phase
- Variance of the activities carried out from the activities planned.
- Summary of results should include
- Comprehensive assessment and recommendation for release should include “Fit for release Assessment”.

Exercise:1 Fill in the blanks

1. _____ testing requires minimum 2 modules to test.
2. _____ testing assures that the software product is bug free and reliable enough for its expected purpose.
3. _____ is the last level of testing.
4. _____ testing is the next level of acceptance testing.
5. _____ can be defined as the delay between the point of request and the first response from the product.

14.9 Let us sum up

In this chapter, we have seen different levels of testing and how it is useful in entire testing process. We have also discussed how testing is performed step by step. In last, we have seen types of reports created during testing process.

14.10 Check your progress: Possible Answers

Exercise 1:

1. Role of drivers and stub
2. Integration testing
3. Categories of system testing
4. Factors affecting performance testing
5. Acceptance testing

14.11 Further Reading

1. Software Testing: Principles and Practices by Srinivasan Desikan, Gopalaswamy Ramesh(PEARSON)
2. Software Testing: A Craftsman's Approach by Paul C. Jorgensen (AUERBACH)