**GAYA COLLEGE OF ENGINEERING, GAYA**

**SOFTWARE ENGINEERING**

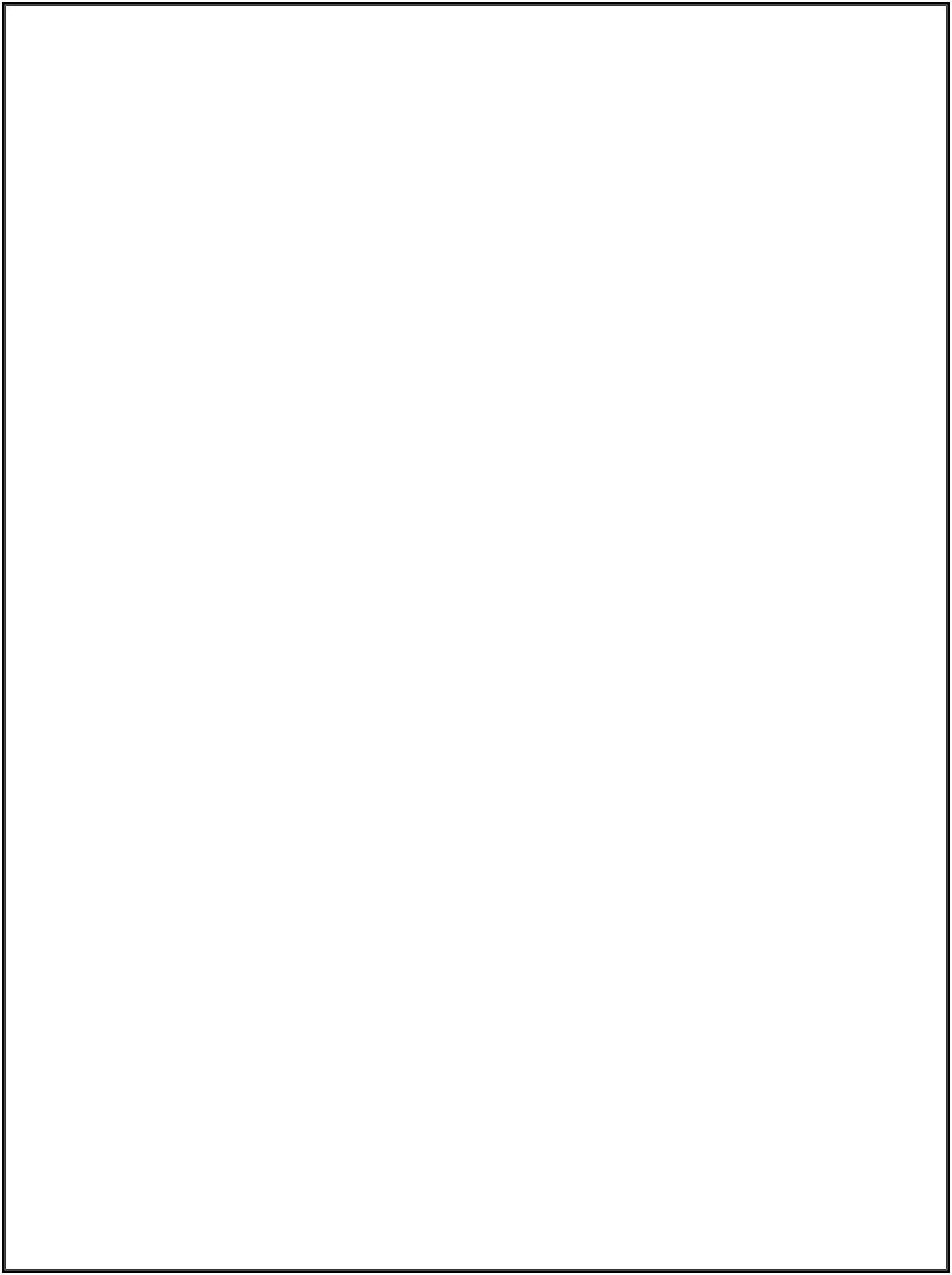
**(051 x 14)**

****

**PRABHAT KUMAR CHANDRA**

**ASSISTANT PROFESSOR**

**DEPARTMENT OF COMPUTER SC. & ENGINEERING**

****

**CONTENTS**

1. Scheme
2. Vision of the Department
3. Mission of the department
4. PEO’s and PO’s
5. Course objectives & course outcomes (CO’s)
6. Mapping of CO’s with PO’s
7. Course Syllabus
8. Time table
9. Student list
10. Course Handout
11. Lecture Plan
12. Assignment sheets
13. Tutorial Sheets
14. Sessional Question Papers
15. Old End Semester Exam (Final Exam) Question Papers
16. Question Bank
17. Power Point Presentations
18. Lecture Notes
19. Reference Materials
20. Results
21. Result Analysis
22. Quality Measurement Sheets
    1. Course End Survey
    2. Teaching Evaluation

**Aryabhatt Knowledge University, Patna**

**Scheme of Teaching & Examination**

**BE (Computer Science & Engineering) VI Semester**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Program** | **semester** | **Course code** | **Course title** | **L** | **T** | **P** | **credit** | **Total credit** |
| CSUG | 6 | 051x04 | OOAD | 3 | 0 | 0 | 3 |  |
| CSUG | 6 | 051x10 | Principal of Programming Language | 3 | 0 | 0 | 3 |  |
| CSUG | 6 | 051x11 | Formal Language and Automata Theory | 3 | 0 | 0 | 3 |  |
| CSUG | 6 | 241x06 | Industrial Economics & Accountancy | 3 | 1 | 0 | 4 |  |
| CSUG | 6 | 051x14 | Software Engineering | 3 | 0 | 3 | 5 |  |
| CSUG | 6 | 051x07 | Web Applications Design & Development | 3 | 0 | 3 | 5 |  |
| CSUG | 6 | 051x16 | Compiler Design | 3 | 0 | 3 | 5 |  |
|  |  |  |  |  |  |  |  | 28 |

1. **COURSE OVERVIEW**

Software Engineering comprises the core principles consistent in software construction and maintenance: fundamental software processes and life-cycles, mathematical foundations of software engineering, requirements analysis, software engineering methodologies and standard notations, principles of software architecture and re-use, software quality frameworks and validation, software development, and maintenance environments and tools. An introduction to object-oriented software development process and design.

II. **PREREQUISITES:**

OOAD

**III. MARKS DISTRIBUTION:**

|  |  |  |  |
| --- | --- | --- | --- |
| Mid semester | Teaching Assessment | ESE | Total |
| 20 | 10 | 70 | 100 |

**Department of Computer Sc. & Engineering**

**Vision**

Department of Computer science and engineering aims to generate computing professionals with global outlook and capable of providing leadership in development of theories products and services for inclusive and sustainable development of society.

**Mission**

* To impart quality education in theoretical and engineering aspects of computing.
* To train students in practices of computing hardware and software through laboratory activity.
* To cultivate students to incorporate for team spirit, efficient problem solving skills, better adaptability and good communication skills to become high professionals

**PEO’s of Computer Sc. & Engineering**

* Graduates will be capable of attaining higher position in their professional carrier, capable to do quality research by strengthening their mathematical, scientific and basic engineering fundamentals.
* Graduate will be capable to develop team-spirit, leadership abilities, collaborative learning, and ethical behaviour.
* Graduate will be capable of adopting the changing technologies, tools, and industrial environment.

**Program Outcomes**

* **Engineering Knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex Electrical and Electronics engineering problems.
* **Problem Analysis**: Students will be able to identify and analyze real life problems in order to provide meaningful solutions with the help of acquired knowledge in computer science.
* **Design/Development of solutions**: Students will be able to design cost effective software/hardware solution, which meets the specific requirement of the client.
* **Conduct Investigations of complex problem**: Students will be able to perform testing on the deployed software and analyse the results.
* **Modern Tool Usage:** Students will be able to create, select and apply appropriate techniques, resources and modern tools to solve existing scenario, within performance and cost constraint.
* **The Engineer and Society:** Students will be able to understand the computing needs of inter-disciplinary scientific and engineering disciplines and design, and develop algorithms and techniques for achieving these.
* **Environment and sustainability:** Students will be able to understand the software project management principles so that they can work cooperatively, responsibly, creatively, and respectfully in teams
* **Individual and team Work:** Students will be able to function effectively as an individual, and as a member or leader in diverse teams.
* **Communication:** Students will be able to communicate effectively in oral, written and graphical form. They should be able to comprehend and write effective reports, design effective presentations, and give and receive clear instructions.
* **Project management and Finance:** Students will be able to manage projects in multidisciplinary environments. They should be able to extend the state of art in some of the areas of interest and create new knowledge.
* **Life-long Learning:**Students will be able to acquire and understand new knowledge, use them to develop software products, and to understand the importance of lifelong learning.
* **Ethics:** Students will be able to understand professional and ethical responsibilities and analyse the impact of computing on individuals, organizations, and the society.

**Program Specific Outcomes:**

**PSO1:** Student will have ability to apply the concepts learn through courses like Algorithm, data structures, Formal methods and theoretical computer science to real life mathematical modelling problems and mathematical research.

**PSO2:** Student will have ability to apply the concepts learn through courses like operating system, Artificial intelligence, Database, Networking, Web Technology and programming languages to software developments.

**PSO3:** The ability to lead and work in a team with good communication, project management and documentation skills

**Course Objectives**:

* Be familiar with basic Software engineering methods and practices, and its applications.
* Master the implementation of software engineering layered technology and Process frame work.
* Be familiar with software measurement and software risks.
* Be familiar with role of project management including planning, scheduling, risk management.
* Master the implementation of different software architectural styles.

**Course Outcomes:**

|  |  |
| --- | --- |
| CO1 | Select and implement different software development process models. |
| CO2 | Extracting and analysing software requirements specifications for different projects |
| CO3 | Developing some basic level of software architecture/design |
| CO4 | Applying standard coding practices. Identification and implementation of software metrics. |
| CO5 | Defining the basic concepts and importance of Software project management concepts like cost estimation, scheduling and reviewing the progress |
| CO6 | Applying different testing and debugging techniques and analysing their effectiveness. |
| CO7 | Analysing software risks and risk management strategies |
| CO8 | Defining the concepts of software quality and reliability on the basis of International quality standards |

**CO-PO Mapping**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Course Outcomes** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** |
| CO1 | √ |  |  | √ |  |  |  |  |  |  |  |  |
| CO2 | √ |  |  |  |  | √ |  |  |  | √ |  |  |
| CO3 |  | √ | √ |  |  | √ |  | √ |  |  |  |  |
| CO4 |  | √ |  | √ |  |  |  |  | √ |  | √ |  |
| CO5 |  |  |  |  |  |  |  |  |  |  | √ |  |
| CO6 |  |  | √ | √ |  |  |  |  |  |  | √ |  |
| CO7 |  |  |  |  |  |  |  |  |  | √ |  |  |
| CO8 |  |  |  |  |  |  |  |  |  | √ |  |  |

**Syllabus**

1. **Introduction:** S/W Engineering Discipline-Evolution and Impact, Program vs S/W Product, Emergence of S/W Engineering.
2. **Software Life Cycle Models:** Waterfall, prototyping, Evolutionary, Spiral models and their comparisions.
3. **Software Project Management**: Project Manager Responsibilities, project planning, project size estimation Metrices, project estimation Techniques, COCOMO, staffing level estimation, scheduling, Organization & team structure staffing, Risk management, S/W Configuration Management.
4. **Requirement Analysis and Specification**: Requirement Gathering and Analysis, SRS, Formal System Development Techniques, Axiomatic and Algebraic Specification.
5. **Software Design:** Overview, Cohesion and coupling, S/W Design Approaches, Object-oriented vs Function oriented Design.
6. **Function-Oriented S/W Design**: SA/SD Methology, Structured Analysis, DFDs, Structured Design, Detailed Design, Design Preview.
7. **Object Modelling using UML:** Overview, UML, UML Diagrams, USE case Models, Class Diagram etc.
8. **Object Oriented Software Development**: Design Patterns, Object-Oriented analysis and Design Process, OOD Goodness Criteria.
9. **User Interface Design**: Characteristics, Basic Concepts, types, Components Baed GUI Development, User Interface Design Methodology.
10. **Coding and Testing**: Coding, code Review, Testing, unit Testing, Black Box Testing, White- Box Testiing, Debugging, Program analysis Tools, Intergration Testig, System Testing, General issues.
11. **Software Relibility and Quality Management**: S/W Reliability, stastical Testing, S/W Quality, S/W Quality Management system ISO 9000, Se CMM, Personal Software Process, Six sigma.
12. **Computer Aided Software Engineering**: CASE and its scope, Environment, Support, other characteristics.
13. **Software maintenance**: Characteristics , S/w Reverse Engineering, S/w maintenance Process Models, estimation of Maintenance Cost.
14. **Software Reuse**: Basic Issues, reuse Approach, Reuse at Organization level.

**Text Books**:

1. Fundamental of Software Engineering by Rajeev Mall, PHI.
2. Software Engineering by James F. Petere Wiley.
3. Software Engineering A. Practioner’s Approach by Pressman, MGH

Time Table:

**Student Lists:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | S.No | Name | Roll No. | | 1 | SAURAV KUMAR | 17/CSE/01 | | 2 | SAKCHI SINGH | 17/CSE/02 | | 3 | NIKHIL KUMAR | 17/CSE/04 | | 4 | SUDHIR KUMAR SAH | 17/CSE/06 | | 5 | MD SHAHZAD SHAMIM | 17/CSE/08 | | 6 | SATYAM PANDEY | 17/CSE/09 | | 7 | RAHUL KUMAR | 17/CSE/10 | | 8 | NANDAN KUMAR | 17/CSE/11 | | 9 | ABHISHEK KUMAR | 17/CSE/12 | | 10 | SHUBHAM KUMAR | 17/CSE/13 | | 11 | KSHITIJ KUMAR | 17/CSE/16 | | 12 | POOJA VERMA | 17/CSE/18 | | 13 | SHASHIKALA KUMARI | 17/CSE/20 | | 14 | VIVEK KUMAR | 17/CSE/22 | | 15 | RASHMI KUMARI | 17/CSE/23 | | 16 | SANTOSH KUMAR CHOUDHARY | 17/CSE/25 | | 17 | INDU KUMARI | 17/CSE/27 | | 18 | ANUBHAV KUMAR | 17/CSE/29 | | 19 | RANJEET KUMAR | 17/CSE/31 | | 20 | CHANDRA MANI | 17/CSE/32 | | 21 | SONU KUMAR | 17/CSE/33 | | 22 | SHALINI KUMARI | 17/CSE/35 | | 23 | ARADHANA KUMARI | 17/CSE/36 | | 24 | SAKSHI KUMARI | 17/CSE/38 | | 25 | ADARSH RAJ | 17/CSE/39 | | 26 | MANISH KUMAR | 17/CSE/42 | | 27 | PUJA KUMARI | 17/CSE/45 | | 28 | SARGAM KUMAR | 17/CSE/46 | | 29 | SUJEET KUMAR YADAV | 17/CSE/47 | | 30 | SANDEEP KUMAR SHARMA | 17/CSE/48 | | 31 | JEEVAN KUMAR | 17/CSE/49 | | 32 | RAKHI KUMARI SINGH | 17/CSE/50 | | 33 | PRINCE PRIYADARSHI | 17/CSE/51 | | 34 | APRAJITA PRIYADARSHINI | 17/CSE/54 | | 35 | AAKANSHA | 17/CSE/56 | | 36 | ARYAN RAJ | 17/CSE/57 | | 37 | GAYATRI BHARTI | 17/CSE/58 | | 38 | PRASHANT KASHYAP | 17/CSE/59 | | 39 | RASHMI PRIYA | 17/CSE/60 | | 40 | PANKAJ KUMAR | 17/CSE/61 | | 41 | SANKET KUMAR | 17/CSE/62 | | 42 | MOHIT KUMAR | 17/CSE/63 | | 43 | NEHAL KUMAR SINGH | 17/CSE/65 | | 44 | SUMAN KUMAR | 17/CSE/68 | | 45 | AKANKSHA KUMARI | 17/CSE/69 | | 46 | MD. INZEMAMUL HAQUE | 17/CSE/70 | | 47 | SAURABH KUMAR | 17/CSE/73 | | 48 | BARUN KUMAR | 17/CSE/74 | | 49 | ANUJ KUMAR | 18/CSE/75LE | | 50 | KUMARI NEHA VAISHALI | 18/CSE/76LE | | 51 | SAURABH KUMAR | 18/CSE/77LE | | 52 | CHANDANI KUMARI | 18/CSE/78LE | | 53 | KHUSHBU KUMARI | 18/CSE/79LE | | 54 | MONIKA KUMARI | 18/CSE/80LE | | 55 | ADARSH KUMAR | 18/CSE/81LE | | 56 | ARTI KUMARI | 18/CSE/82LE | | 57 | ANUPAM PATEL | 18/CSE/83LE | | 58 | SURUCHI KUMARI | 18/CSE/84LE | |

**Detailed Teaching Plan:**

|  |  |  |
| --- | --- | --- |
| **Lecture No.** | **Topics to be covered** | **Course Learning Outcomes** |
| **1-3** | The evolving role of software, Changing Nature of Software. Legacy software, Software Myths. | **Explain** the evolution of software |
| **4** | Software engineering-A layered technology, a process framework. | **Explain** process frame work |
| **5** | The Capability Maturity Model Integration (CMMI) | **Illustrate** CMMI |
| **6** | Process Patterns, Process Assessment, personal and team process models. | **Explain** process patterns |
| **7-9** | The Waterfall model, Incremental process models, Evolutionary Process Models, Specialized Process Models, The Unified Process | **Demonstrate** waterfall model, incremental evolutionary, specialized models |
| **8** | Functional and non-Functional Requirements | **Distinguish** between Functional and non functional requirements |
| **9-10** | User Requirements, System Requirements. Interface Specification, the software requirement document | **Discuss** user and system requirements, **Explain** software requirement document |
| **11-12** | Requirements elicitation and analysis, requirements validation, Requirements management | **Demonstrate** requirement management |
| **13-14** | Context Models, behavioural models, Data models, object models, structured method | **Demonstrate** Design Engineering |
| **15-16** | Design process and design quality, Design concepts | **Explain** design concepts |
| **17** | The design model, pattern based software design | **Explain** software design |
| **18-19** | Software architecture, Data design, Architectural Styles and patterns, Architectural design | **Demonstrate** Architectural Styles and patterns |
| **20-21** | Assessing alternative architectural designs, mapping data flow into software architecture. | **Illustrate** data flow in software architecture |
| **22** | Designing class –based components, conducting component-level design, object constraint language, designing conventional components | **Explain** component level design |
| **23** | Golden rules, User interface analysis and design, interface analysis | **Summarize** golden rules |
| **24** | Interface design steps, Design evaluation. | **Explain** interface design |
| **25-28** | A strategic approach to software testing, test strategies for conventional software, Black- Box and White-Box testing, Validation testing, system testing, the art of debugging. | **Demonstrate** testing techniques |
| **29-30** | Software Quality, Frame work for product metrics, Metrics for Analysis Model, Metrics for Design Model | **Explain** software quality |
| **31** | Metrics for source code, metrics for testing, metrics for maintenance | **Demonstrate** metrics for testing |
| **32** | Metrics for process and products: Software Measurement, Metrics for software quality | **Explain** metrics of software quality |
| **33-34** | Reactive vs. Proactive Risk strategies, software risks, Risk identification, Risk projection, Risk refinement, RMMM, RMMM plan. | **Demonstrate** RMMM |
| **35-36** | Quality concepts, software quality assurance, Software Reviews, Formal technical Reviews | **Explain** quality concepts |
| **37-38** | Statistical Software quality Assurance, Software reliability, The ISO 9000 quality standards | **Demonstrate** quality standards |

**Class Notes**

**Chapter 1**

**INTRODUCTION TO SOFTWARE ENGINEERING**

**Chapter 2**

**REQUIREMENTS ANALYSIS AND SPECIFICATION**

**Chapter 3**

**SOFTWARE DESIGN**

**Chapter 4**

**CODING**

**Chapter 5**

**TESTING**

**Chapter 6**

**SOFTWARE MAINTENANCE**

**Chapter 1**

**INTRODUCTION TO SOFTWARE ENGINEERING**

The term ***software engineering*** is composed of two words, software and engineering.

**Software** is more than just a program code. A program is an executable code, which servessome computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

**Engineering** on the other hand, is all about developing products, using well-defined, scientificprinciples and methods.

So, we can define ***software engineering*** as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

IEEE defines software engineering as:

*The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.*

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: ***abstraction*** and ***decomposition*.** The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem. The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

**NEED OF SOFTWARE ENGINEERING**

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

* **Large software -** It is easier to build a wall than to a house or building, likewise, as thesize of software become large engineering has to step to give it a scientific process.
* **Scalability-** If the software process were not based on scientific and engineeringconcepts, it would be easier to re-create new software than to scale an existing one.
* **Cost-** As hardware industry has shown its skills and huge manufacturing has lower downthe price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
* **Dynamic Nature-** The always growing and adapting nature of software hugely dependsupon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
* **Quality Management-** Better process of software development provides better andquality software product.

**CHARACTERESTICS OF GOOD SOFTWARE**

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

* Operational
* Transitional
* Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

**Operational**

This tells us how well software works in operations. It can be measured on:

* Budget
* Usability
* Efficiency
* Correctness
* Functionality
* Dependability
* Security
* Safety

**Transitional**

This aspect is important when the software is moved from one platform to another:

* Portability
* Interoperability
* Reusability
* Adaptability

**Maintenance**

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

* Modularity
* Maintainability
* Flexibility
* Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

**SOFTWARE DEVELOPMENT LIFE CYCLE**

**LIFE CYCLE MODEL**

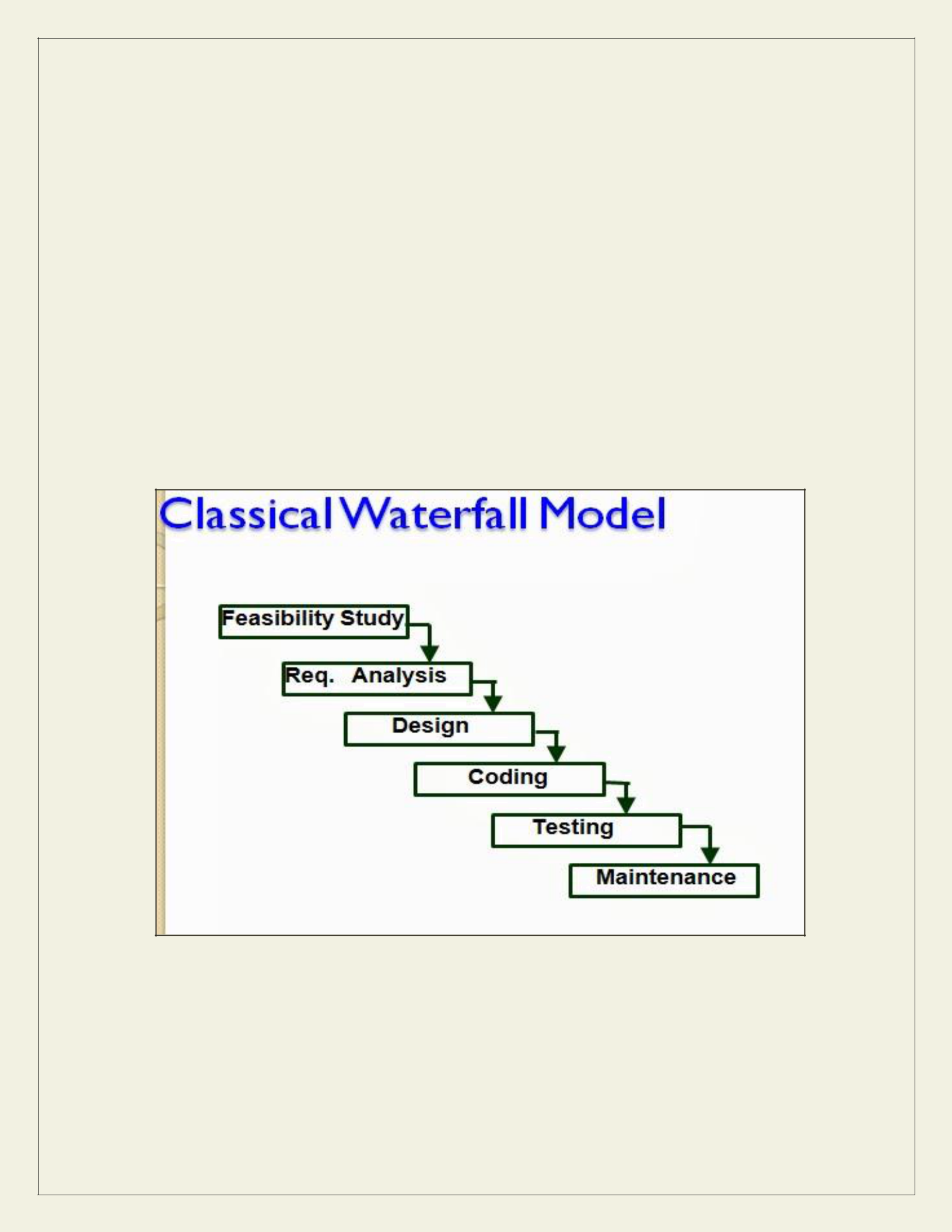
A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle. A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement. Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

**THE NEED FOR A SOFTWARE LIFE CYCLE MODEL**

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner. When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure. This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure. A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

***Different software life cycle models***

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

* + - Classical Waterfall Model
    - Iterative Waterfall Model
    - Prototyping Model
    - Evolutionary Model
    - Spiral Model

1. **CLASSICAL WATERFALL MODEL**

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.2.1

Fig 2.1 waterfall Model

* At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.
* After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
* Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

**Requirements analysis and specification: -** The aim of the requirements analysis andspecification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely

* Requirements gathering and analysis
* Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

**Design: -** The goal of the design phase is to transform the requirements specified in the SRSdocument into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. Two distinctly different approaches are available: the traditional design approach and the object-oriented design approach.

* **Traditional design approach -**Traditional design consists of two different activities; firsta structured analysis of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a structured design activity. During structured design, the results of structured analysis are transformed into the software design.
* **Object-oriented design approach** -In this technique, various objects that occur in theproblem domain and the solution domain are first identified, and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

**Coding and unit testing:-**The purpose of the coding phase (sometimes called theimplementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

**Integration and system testing: -**Integration of different modules is undertaken once they havebeen coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

* α – testing: It is the system testing performed by the development team.
* β –testing: It is the system testing performed by a friendly set of customers.

Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed,

specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

**Maintenance: -**Maintenance of a typical software product requires much more than the effortnecessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratios. Maintenance involves performing any one or more of the following three kinds of activities:

* Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.
* Improving the implementation of the system, and enhancing the functionalities of the system according to the customer’s requirements. This is called perfective maintenance.
* Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

***Shortcomings of the classical waterfall model***

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

1. **ITERATIVE WATERFALL MODEL**

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

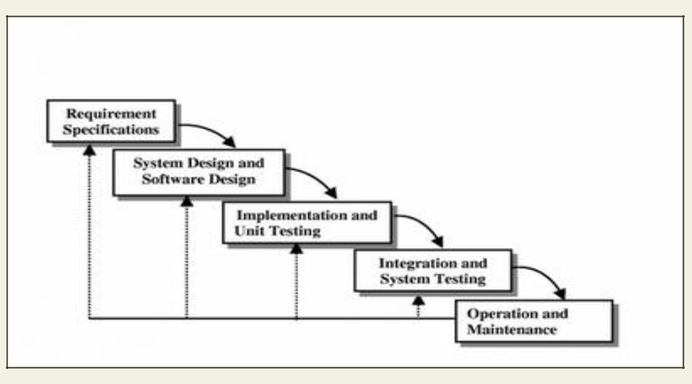


Fig 3.1: Iterartive waterfall Model

1. **PRTOTYPING MODEL Prototype**

A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts. The shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

***Need for a prototype in software development***

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer’s needs:

* + how the screens might look like
  + how the user interface would behave
  + how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

A prototype of the actual product is preferred in situations such as:

* User requirements are not complete
* Technical issues are not clear

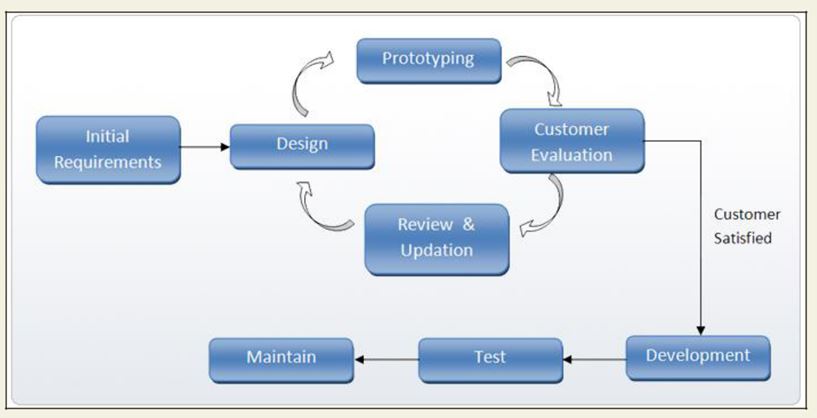


Fig 3.2: Prototype Model

**EVOLUTIONARY MODEL**

It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

* Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
* Also used in object oriented software development because the system can be easily

portioned into units in terms of objects.

Advantages:

* User gets a chance to experiment partially developed system
* Reduce the error because the core modules get tested thoroughly. Disadvantages:

It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.

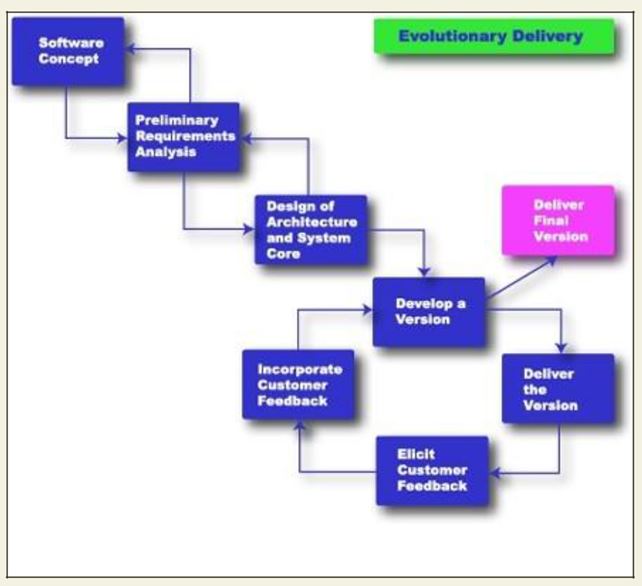


Fig: 3.3 Evolutionary Model

1. **SPIRAL MODEL**

The Spiral model of software development is shown in fig. 4.1. The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in fig. 4.1. The following activities are carried out during each phase of a spiral model.

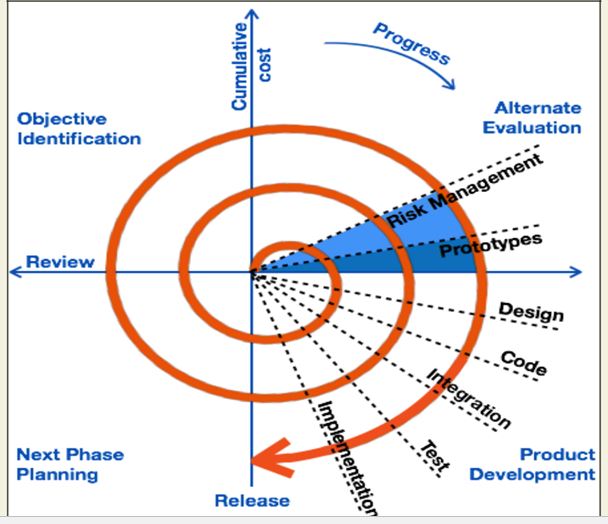


Fig: Spiral Model

**First quadrant (Objective Setting)**

* During the first quadrant, it is needed to identify the objectives of the phase.

Examine the risks associated with these objectives.

**Second Quadrant (Risk Assessment and Reduction)**

* + A detailed analysis is carried out for each identified project risk.
* Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

**Third Quadrant (Development and Validation)**

* Develop and validate the next level of the product after resolving the identified risks.

**Fourth Quadrant (Review and Planning)**

* Review the results achieved so far with the customer and plan the next iteration around the spiral.
* Progressively more complete version of the software gets built with each iteration around the spiral.

***Circumstances to use spiral model***

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

***Comparison of different life-cycle models***

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer’s trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer’s financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

**CHAPTER 2**

**REQUIREMENTS ANALYSIS AND SPECIFICATION**

Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as ***system analysts***.

The analyst starts *requirements gathering and analysis* activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

* What is the problem?
* Why is it important to solve the problem?
* What are the possible solutions to the problem?
* What exactly are the data input to the system and what exactly are the data output by the system?
* What are the likely complexities that might arise while solving the problem?
* If there are external software or hardware with which the developed software has to interface, then what exactly would the data interchange formats with the external system be?

After the analyst has understood the exact customer requirements, he proceeds to identify and resolve the various requirements problems. The most important requirements problems that the analyst has to identify and eliminate are the problems of anomalies, inconsistencies, and incompleteness. When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end-users and the customers.

Parts of a SRS document

• The important parts of SRS document are:

Functional requirements of the system

Non-functional requirements of the system, and

Goals of implementation

**Functional requirements:-**

The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions {*fi*}. The functional view of the

system is shown in fig. 5.1. Each function fi of the system can be considered as a transformation of a set of input data (ii) to the corresponding set of output data (*oi*). The user can get some meaningful piece of work done using a high-level function

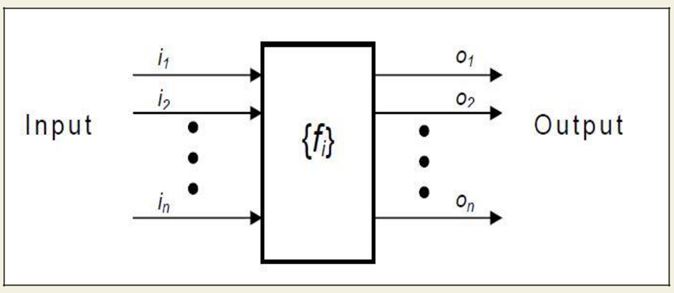
****

Fig. 5.1: View of a system performing a set of functions

**Nonfunctional requirements:-**

Nonfunctional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

**Goals of implementation:-**

The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

***Identifying functional requirements from a problem description***

The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each high-level requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user’s perspective.

**Example: -** Consider the case of the library system, where **–**

**F1:** Search Book function

**Input:** an author’s name

**Output:** details of the author’s books and the location of these books in the library

So the function Search Book (F1) takes the author's name and transforms it into book details.

Functional requirements actually describe a set of high-level requirements, where each high-level requirement takes some data from the user and provides some data to the user as an output. Also each high-level requirement might consist of several other functions.

***Documenting functional requirements***

For documenting the functional requirements, we need to specify the set of functionalities supported by the system. A function can be specified by identifying the state at which the data is to be input to the system, its input data domain, the output data domain, and the type of processing to be carried on the input data to obtain the output data. Let us first try to document the withdraw-cash function of an ATM (Automated Teller Machine) system. The withdraw-cash is a high-level requirement. It has several sub-requirements corresponding to the different user interactions. These different interaction sequences capture the different scenarios.

**Example: -** Withdraw Cash from ATM

R1: withdraw cash

Description: The withdraw cash function first determines the type of account that the user has and the account number from which the user wishes to withdraw cash. It checks the balance to determine whether the requested amount is available in the account. If enough balance is available, it outputs the required cash; otherwise it generates an error message.

select withdraw amount option

Input: “withdraw amount” option

Output: user prompted to enter the account type

R1.2: select account type

Input: user option

Output: prompt to enter amount

R1.3: get required amount

Input: amount to be withdrawn in integer values greater than 100 and less than 10,000 in multiples of 100.

Output: The requested cash and printed transaction statement.

Processing: the amount is debited from the user’s account if sufficient balance is available, otherwise an error message displayed

***Properties of a good SRS document***

The important properties of a good SRS document are the following:

* **Concise.** The SRS document should be concise and at the same time unambiguous,consistent, and complete. Verbose and irrelevant descriptions reduce readability and also increase error possibilities.
* **Structured.** It should be well-structured. A well-structured document is easy tounderstand and modify. In practice, the SRS document undergoes several revisions to cope up with the customer requirements. Often, the customer requirements evolve over a period of time. Therefore, in order to make the modifications to the SRS document easy, it is important to make the document well-structured.

**Black-box view.** It should only specify what the system should do and refrain fromstating how to do these. This means that the SRS document should specify the external behavior of the system and not discuss the implementation issues. The SRS document should view the system to be developed as black box, and should specify the externally visible behavior of the system. For this reason, the SRS document is also called the black-box specification of a system.

* **Conceptual integrity.** It should show conceptual integrity so that the reader can easilyunderstand it.
* **Response to undesired events.** It should characterize acceptable responses to undesiredevents. These are called system response to exceptional conditions.
* **Verifiable.** All requirements of the system as documented in the SRS document shouldbe verifiable. This means that it should be possible to determine whether or not requirements have been met in an implementation.

***Problems without a SRS document***

The important problems that an organization would face if it does not develop a SRS document are as follows:

* Without developing the SRS document, the system would not be implemented according to customer needs.
* Software developers would not know whether what they are developing is what exactly required by the customer.
* Without SRS document, it will be very much difficult for the maintenance engineers to understand the functionality of the system.
* It will be very much difficult for user document writers to write the users’ manuals properly without understanding the SRS document.

***Problems with an unstructured specification***

* It would be very much difficult to understand that document.
* It would be very much difficult to modify that document.
* Conceptual integrity in that document would not be shown.
* The SRS document might be unambiguous and inconsistent.

**CHAPTER 3**

**SOFTWARE DESIGN**

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

For assessing user requirements, an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

Software Design Levels

Software design yields three levels of results:

* **Architectural Design -** The architectural design is the highest abstract version of thesystem. It identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of proposed solution domain.
* **High-level Design-** The high-level design breaks the ‘single entity-multiple component’concept of architectural design into less-abstracted view of sub-systems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other.
* **Detailed Design-** Detailed design deals with the implementation part of what is seen as asystem and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

**Modularization**

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally follows the rules of ‘divide and conquer’ problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

* Smaller components are easier to maintain
* Program can be divided based on functional aspects
* Desired level of abstraction ca n be brought in the program
* Components with high cohesion can be re-used again.
* Concurrent execution can be made possible
* Desired from security aspect

**Concurrency**

Back in time, all softwares were meant to be executed sequentially. By sequential execution we mean that the coded instruction will be executed one after another implying only one portion of program being activated at any given time. Say, a software has multiple modules, then only one of all the modules can be found active at any time of execution.

In software design, concurrency is implemented by splitting the software into multiple independent units of execution, like modules and executing them in parallel. In other words, concurrency provides capability to the software to execute more than one part of code in parallel to each other.

It is necessary for the programmers and designers to recognize those modules, which can be made parallel execution.

Example

The spell check feature in word processor is a module of software, which runs alongside the word processor itself.

**Coupling and Cohesion**

When a software program is modularized, its tasks are divided into several modules based on some characteristics. As we know, modules are set of instructions put together in order to achieve some tasks. They are though, considered as single entity but may refer to each other to work together. There are measures by which the quality of a design of modules and their interaction among them can be measured. These measures are called coupling and cohesion.

**Cohesion**

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

There are seven types of cohesion, namely –

* **Co-incidental cohesion -** It is unplanned and random cohesion, which might be the resultof breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.
* **Logical cohesion -** When logically categorized elements are put together into a module,it is called logical cohesion.
* **Temporal Cohesion -** When elements of module are organized such that they areprocessed at a similar point in time, it is called temporal cohesion.
* **Procedural cohesion -** When elements of module are grouped together, which areexecuted sequentially in order to perform a task, it is called procedural cohesion.
* **Communicational cohesion -** When elements of module are grouped together, which areexecuted sequentially and work on same data (information), it is called communicational cohesion.
* **Sequential cohesion -** When elements of module are grouped because the output of oneelement serves as input to another and so on, it is called sequential cohesion.
* **Functional cohesion -** It is considered to be the highest degree of cohesion, and it ishighly expected. Elements of module in functional cohesion are grouped because they all contribute to a single well-defined function. It can also be reused.

**Coupling**

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

* **Content coupling -** When a module can directly access or modify or refer to the contentof another module, it is called content level coupling.
* **Common coupling-** When multiple modules have read and write access to some globaldata, it is called common or global coupling.
* **Control coupling-** Two modules are called control-coupled if one of them decides thefunction of the other module or changes its flow of execution.
* **Stamp coupling-** When multiple modules share common data structure and work ondifferent part of it, it is called stamp coupling.
* **Data coupling-** Data coupling is when two modules interact with each other by means ofpassing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

**Design Verification**

The output of software design process is design documentation, pseudo codes, detailed logic diagrams, process diagrams, and detailed description of all functional or non-functional requirements.

The next phase, which is the implementation of software, depends on all outputs mentioned above.

It is then becomes necessary to verify the output before proceeding to the next phase. The early any mistake is detected, the better it is or it might not be detected until testing of the product. If the outputs of design phase are in formal notation form, then their associated tools for verification should be used otherwise a thorough design review can be used for verification and validation.

By structured verification approach, reviewers can detect defects that might be caused by overlooking some conditions. A good design review is important for good software design, accuracy and quality.

**SOFTWARE DESIGN STRATEGIES**

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Let us study them briefly:

**Structured Design**

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

**Cohesion** - grouping of all functionally related elements.

**Coupling** - communication between different modules.

A good structured design has ***high*** cohesion and ***low*** coupling arrangements.

**Function Oriented Design**

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

**Design Process**

* The whole system is seen as how data flows in the system by means of data flow diagram.
* DFD depicts how functions change the data and state of entire system.
* The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
* Each function is then described at large.

**Object Oriented Design**

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategy focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

* **Objects -** All entities involved in the solution design are known as objects. For example,person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.

**Classes -** A class is a generalized description of an object. An object is an instance of aclass. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

* **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on thedata) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
* **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where thelower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
* **Polymorphism -** OOD languages provide a mechanism where methods performingsimilar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

**Design Process**

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

* A solution design is created from requirement or previous used system and/or system sequence diagram.
* Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
* Class hierarchy and relation among them are defined.
* Application framework is defined.

**Software Design Approaches**

There are two generic approaches for software designing:

**Top down Design**

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their one set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-

system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

**Bottom-up Design**

The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

**SOFTWARE ANALYSIS & DESIGN TOOLS**

Software analysis and design includes all activities, which help the transformation of requirement specification into implementation. Requirement specifications specify all functional and non-functional expectations from the software. These requirement specifications come in the shape of human readable and understandable documents, to which a computer has nothing to do.

Software analysis and design is the intermediate stage, which helps human-readable requirements to be transformed into actual code.

Let us see few analysis and design tools used by software designers:

**Data Flow Diagram**

Data flow diagram is a graphical representation of data flow in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

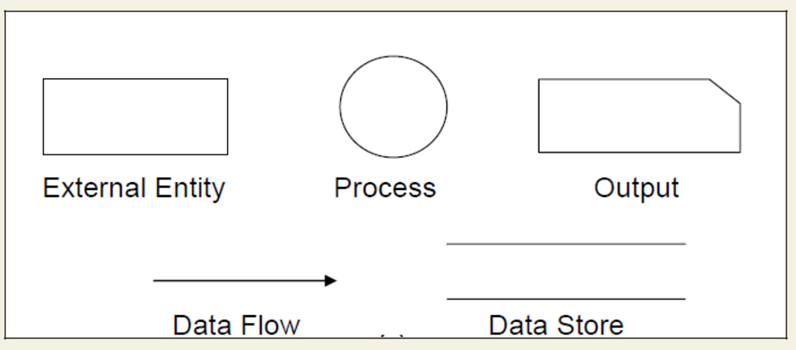
**Types of DFD**

Data Flow Diagrams are either Logical or Physical.

* **Logical DFD** - This type of DFD concentrates on the system process and flow of data inthe system. For example in a Banking software system, how data is moved between different entities.
* **Physical DFD** - This type of DFD shows how the data flow is actually implemented inthe system. It is more specific and close to the implementation.

**DFD Components:**

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

 Fig: DFD Components

* **Entities** - Entities are source and destination of information data. Entities are representedby rectangles with their respective names.
* **Process** - Activities and action taken on the data are represented by Circle or Round-edged rectangles.
* **Data Storage** - There are two variants of data storage - it can either be represented as arectangle with absence of both smaller sides or as an open-sided rectangle with only one side missing.
* **Data Flow** - Movement of data is shown by pointed arrows. Data movement is shownfrom the base of arrow as its source towards head of the arrow as destination.

**Importance of DFDs in a good software design**

The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism – it is simple to understand and use. Starting with a set of high-level functions that a system performs, a DFD model hierarchically represents various sub-functions. In fact, any hierarchical model is simple to understand. Human mind is such that it can easily understand any hierarchical model of a system – because in a hierarchical model, starting with a very simple and abstract model of a system, different details of the system are slowly introduced through different hierarchies. The data flow diagramming technique also follows a very simple set of intuitive concepts and rules. DFD is an elegant modeling technique that turns out to be useful not only to represent the results of structured analysis of a software problem, but also for several other applications such as showing the flow of documents or items in an organization.

**Data Dictionary**

A data dictionary lists all data items appearing in the DFD model of a system. The data items listed include all data flows and the contents of all data stores appearing on the DFDs in the DFD model of a system. A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items. For example, a data dictionary entry may represent that the data **grossPay** consists of the components regularPay and overtimePay.

**grossPay = regularPay + overtimePay**

For the smallest units of data items, the data dictionary lists their name and their type. Composite data items can be defined in terms of primitive data items using the following data definition operators:

**+**: denotes composition of two data items, e.g. **a+b** represents data a and **b**.

**[,,]**: represents selection, i.e. any one of the data items listed in the brackets can occur.

For example, **[a,b]** represents either **a** occurs or **b** occurs.

**()**: the contents inside the bracket represent optional data which may or may not appear.

e.g. **a+(b)** represents either **a** occurs or **a+b** occurs.

**{}**: represents iterative data definition, e.g. **{name}5** represents five **name** data. **{name}\*** represents zero or more instances of **name** data.

**=**: represents equivalence, e.g. **a=b+c** means that **a** represents **b** and **c**.

**/\* \*/**: Anything appearing within **/\*** and **\*/** is considered as a comment.

**Example 1:** Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

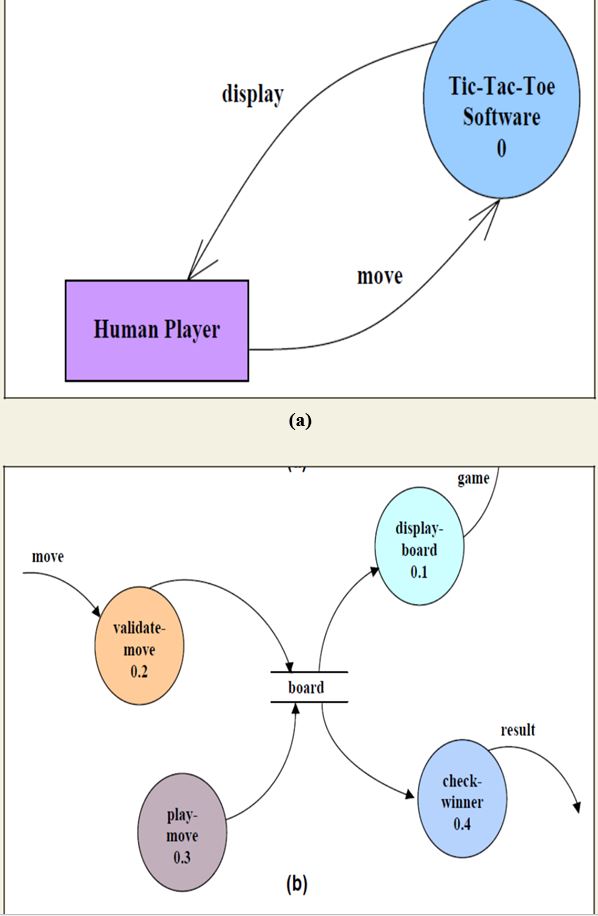


Fig (a) Level 0 (b) Level 1 DFD for Tic-Tac-Toe game

It may be recalled that the DFD model of a system typically consists of several DFDs: level 0, level 1, etc. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the model. Figure 10.2 represents the level 0 and level 1 DFDs for the tic-tac-toe game. The data dictionary for the model is given below.

**Data Dictionary for the DFD model in Example 1**

move: integer /\*number between 1 and 9 \*/

display: game+result

game: board

board: {integer}9

result: [“computer won”, “human won” “draw”]

**Importance of Data Dictionary**

A data dictionary plays a very important role in any software development process because of the following reasons:

* A data dictionary provides a standard terminology for all relevant data for use by the engineers working in a project. A consistent vocabulary for data items is very important, since in large projects different engineers of the project have a tendency to use different terms to refer to the same data, which unnecessary causes confusion.
* The data dictionary provides the analyst with a means to determine the definition of different data structures in terms of their component elements.

**Shortcomings of a DFD model**

DFD models suffer from several shortcomings. The important shortcomings of the DFD models are the following:

* ***DFDs leave ample scope to be imprecise*** - In the DFD model, the function performed bya bubble is judged from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find-book-position has only intuitive meaning and does not specify several things, e.g. what happens when some input information are missing or are incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
* ***Control aspects are not defined by a DFD***- For instance; the order in which inputs areconsumed and outputs are produced by a bubble is not specified. A DFD model does not specify the order in which the different bubbles are executed. Representation of such aspects is very important for modeling real-time systems.

The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.

* The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its sub-functions and we have to use subjective judgment to carry out decomposition.

**STRUCTURED DESIGN**

The aim of structured design is to transform the results of the structured analysis (i.e. a DFD representation) into a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart.

* Transform analysis
* Transaction analysis

Normally, one starts with the level 1 DFD, transforms it into module representation using either the transform or the transaction analysis and then proceeds towards the lower-level DFDs. At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD. These are discussed in the subsequent sub-sections.

**Structure Chart**

A structure chart represents the software architecture, i.e. the various modules making up the system, the dependency (which module calls which other modules), and the parameters that are passed among the different modules. Hence, the structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on the module structure of the software and the interactions among different modules, the procedural aspects (e.g. how a particular functionality is achieved) are not represented.

The basic building blocks which are used to design structure charts are the following:

* **Rectangular boxes:** Represents a module.
* **Module invocation arrows:** Control is passed from on one module to anothermodule in the direction of the connecting arrow.
* **Data flow arrows:** Arrows are annotated with data name; named data passesfrom one module to another module in the direction of the arrow.
* **Library modules:** Represented by a rectangle with double edges.
* **Selection:** Represented by a diamond symbol.
* **Repetition:** Represented by a loop around the control flow arrow.

**Structure Chart vs. Flow Chart**

We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:

* It is usually difficult to identify the different modules of the software from its flow chart representation.
* Data interchange among different modules is not represented in a flow chart.
* Sequential ordering of tasks inherent in a flow chart is suppressed in a structure chart.

**Transform Analysis**

Transform analysis identifies the primary functional components (modules) and the high level inputs and outputs for these components. The first step in transform analysis is to divide the DFD into 3 types of parts:

* Input
* Logical processing
* Output

The input portion of the DFD includes processes that transform input data from physical (e.g. character from terminal) to logical forms (e.g. internal tables, lists, etc.). Each input portion is called an afferent branch.

The output portion of a DFD transforms output data from logical to physical form. Each output portion is called an efferent branch. The remaining portion of a DFD is called the central transform.

In the next step of transform analysis, the structure chart is derived by drawing one functional component for the ***central transform***, and the ***afferent*** and ***efferent*** branches.

These are drawn below a root module, which would invoke these modules. Identifying the highest level input and output transforms requires experience and skill. One possible approach is to trace the inputs until a bubble is found whose output cannot be deduced from its inputs alone. Processes which validate input or add information to them are not central transforms. Processes which sort input or filter data from it are. The first level structure chart is produced by representing each input and output unit as boxes and each central transform as a single box. In the third step of transform analysis, the structure chart is refined by adding sub-functions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called factoring. Factoring includes adding read and write modules, error-handling modules, initialization and termination process, identifying customer modules, etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

**Example:** Structure chart for the RMS software

For this example, the context diagram was drawn earlier.

To draw the level 1 DFD (fig.11.1), from a cursory analysis of the problem description, we can see that there are four basic functions that the system needs to perform – accept the input numbers from the user, validate the numbers, calculate the root mean square of the input numbers and, then display the result.

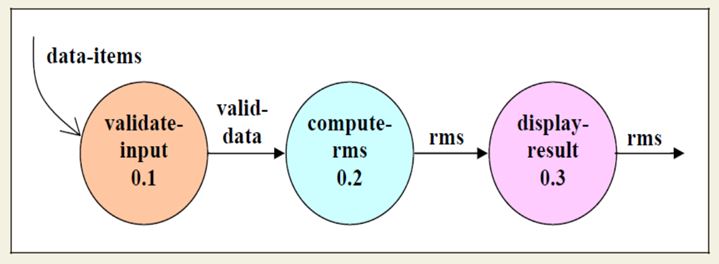


Fig. 11.1: Level 1 DFD

By observing the level 1 DFD, we identify the validate-input as the afferent branch and write-output as the efferent branch. The remaining portion (i.e. compute-rms) forms the central transform. By applying the step 2 and step 3 of transform analysis, we get the structure chart shown in fig.11.2.

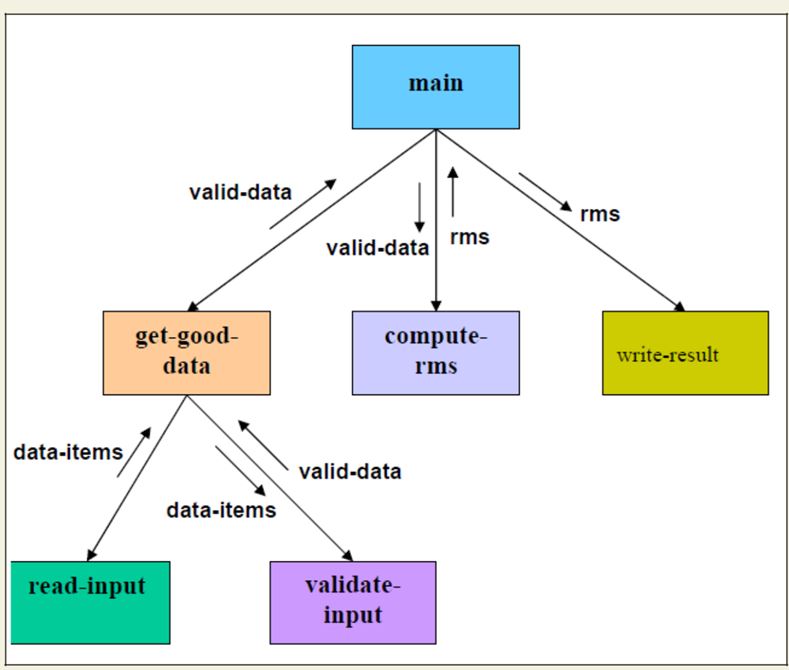


Fig. 11.2: Structure Chart

**Transaction Analysis**

A transaction allows the user to perform some meaningful piece of work. Transaction analysis is useful while designing transaction processing programs. In a transaction-driven system, one of several possible paths through the DFD is traversed depending upon the input data item. This is in contrast to a transform centered system which is characterized by similar processing steps for each data item. Each different way in which input data is handled is a transaction. A simple way to identify a transaction is to check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transaction may not require any input data. These transactions can be identified from the experience of solving a large number of examples.

For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction a module. Every transaction carries a tag, which identifies its type.

Transaction analysis uses this tag to divide the system into transaction modules and a transaction-center module.

The structure chart for the supermarket prize scheme software is shown in fig. 11.3.

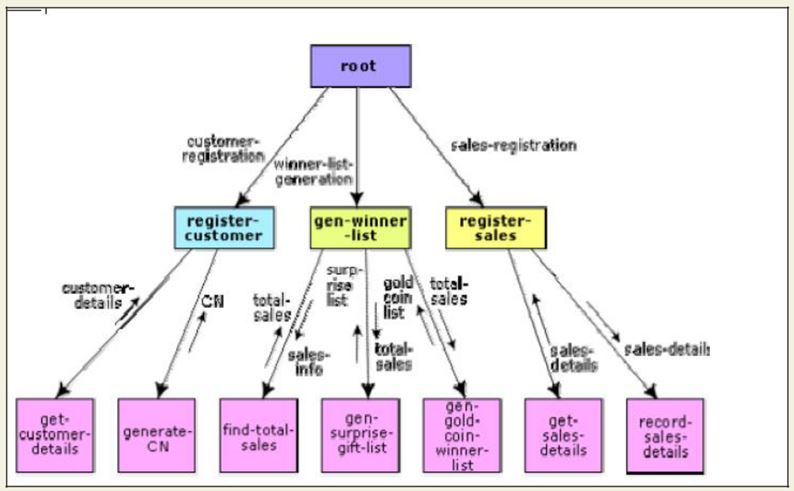


Fig. 11.3: Structure Chart for the supermarket prize scheme

**OBJECT MODELLING USING UML**

**Model**

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting the design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

**Need for a model**

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following**:**

* Analysis
* Specification
* Code generation
* Design
* Visualize and understand the problem and the working of a system
* Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model used for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.

**Unified Modeling Language (UML)**

UML, as the name implies, is a modeling language. It may be used to visualize, specify, construct, and document the artifacts of a software system. It provides a set of notations (e.g. rectangles, lines, ellipses, etc.) to create a visual model of the system. Like any other language, UML has its own syntax (symbols and sentence formation rules) and semantics (meanings of symbols and sentences). Also, we should clearly understand that UML is not a system design or development methodology, but can be used to document object-oriented and analysis results obtained using some methodology.

**Origin of UML**

In the late 1980s and early 1990s, there was a proliferation of object-oriented design techniques and notations. Different software development houses were using different notations to document their object-oriented designs. These diverse notations used to give rise to a lot of confusion.

UML was developed to standardize the large number of object-oriented modeling notations that existed and were used extensively in the early 1990s. The principles ones in use were:

* Object Management Technology [Rumbaugh 1991]
* Booch’s methodology [Booch 1991]
* Object-Oriented Software Engineering [Jacobson 1992]
* Odell’s methodology [Odell 1992]
* Shaler and Mellor methodology [Shaler 1992]

It is needless to say that UML has borrowed many concepts from these modeling techniques. Especially, concepts from the first three methodologies have been heavily drawn upon. UML was adopted by Object Management Group (OMG) as a *de facto* standard in 1997. OMG is an association of industries which tries to facilitate early formation of standards.

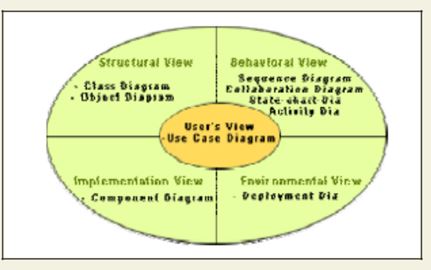
We shall see that UML contains an extensive set of notations and suggests construction of many types of diagrams. It has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and a strong industry backing have helped UML find widespread acceptance. UML is now being used in a large number of software development projects worldwide.

U**ML Diagram**

UML can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modeled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system**:**

* User’s view
* Structural view
* Behavioral view
* Implementation view
* Environmental view

****

**User’s view:** This view defines the functionalities (facilities) made available by the system toits users. The users’ view captures the external users’ view of the system in terms of the functionalities offered by the system. The users’ view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users’ view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users’ view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

**Structural view:** The structural view defines the kinds of objects (classes) important to theunderstanding of the working of a system and to its implementation. It also captures the

relationships among the classes (objects). The structural model is also called the static model, since the structure of a system does not change with time.

**Behavioral view:** The behavioral view captures how objects interact with each other to realizethe system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

**Implementation view:** This view captures the important components of the system and theirdependencies.

**Environmental view:** This view models how the different components are implemented ondifferent pieces of hardware.

**USE CASE DIAGRAM**

**Use Case Model**

The use case model for any system consists of a set of “use cases”. Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: “What the users can do using the system?” Thus for the Library Information System (LIS), the use cases could be**:**

* issue-book
* query-book
* return-book
* create-member
* add-book, etc

Use cases correspond to the high-level functional requirements. The use cases partition the system behavior into transactions, such that each transaction performs some useful action from the user’s point of view. To complete each transaction may involve either a single message or multiple message exchanges between the user and the system to complete. Purpose of use cases

The purpose of a use case is to define a piece of coherent behavior without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

**Representation of Use-cases:**

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e. use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (such as Library Information System) appears inside the rectangle.

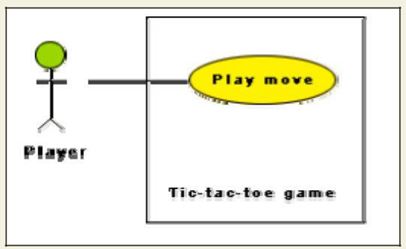
The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype <<external system>>.

**Example 1:**

Tic-Tac-Toe Computer Game

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3×3 square. A move consists of marking previously unmarked square. The player who first places three consecutive marks along a straight line on the square (i.e. along a row, column, or diagonal) wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

The use case model for the Tic-tac-toe problem is shown in fig. 13.1. This software has only one use case “play move”. Note that the use case “get-user-move” is not used here. The name “get-user-move” would be inappropriate because the use cases should be named from the user’s perspective.

****

**Text- Descriptions:**

Each ellipse on the use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer and other aspects of the use case. It should include all the behavior associated with the use case in terms of the mainline sequence, different variations to the normal behavior, the system responses associated with the use case, the exceptional conditions that may occur in the behavior, etc. The behavior description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is recommended. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternative scenarios.

**Contact persons:** This section lists the personnel of the client organization with whom the usecase was discussed, date and time of the meeting, etc.

**Actors:** In addition to identifying the actors, some information about actors using this use casewhich may help the implementation of the use case may be recorded.

**Pre-condition:** The preconditions would describe the state of the system before the use caseexecution starts.

**Post-condition:** This captures the state of the system after the use case has successfullycompleted.

**Non-functional requirements:** This could contain the important constraints for the design andimplementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

**Exceptions, error situations:** This contains only the domain-related errors such as lack ofuser’s access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

**Sample dialogs:** These serve as examples illustrating the use case.

**Specific user interface requirements:** These contain specific requirements for the userinterface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

**Document references:** This part contains references to specific domain-related documentswhich may be useful to understand the system operation,.

**Utility of use case diagrams**

From use case diagram, it is obvious that the utility of the use cases are represented by ellipses. They along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. Another possible use is in preparing the documentation (e.g. users’ manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

**Factoring of use cases**

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases are required under two situations. First, complex use cases need to be factored into simpler use cases. This would not only make the behavior associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single sized (A4) paper. Secondly, use cases need to be factored whenever there is common behavior across different use cases. Factoring would make it possible to define such behavior only once and reuse it whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it.

UML offers three mechanisms for factoring of use cases as follows:

1. **Generalization**

Use case generalization can be used when one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the parent use case. The notation is the same too (as shown in fig. 13.3). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

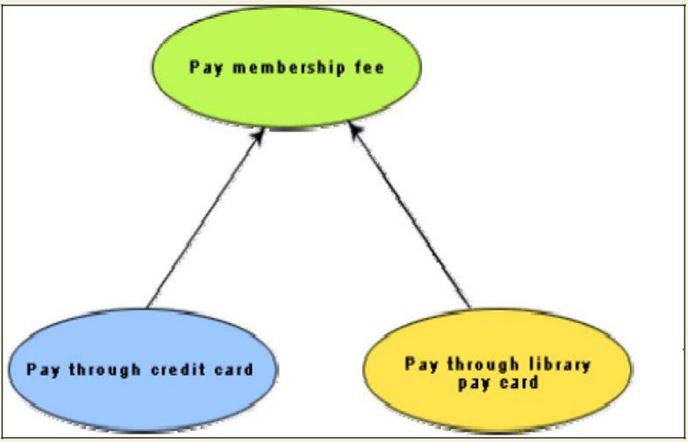


Fig.: Representation of use case generalization

**Includes**

The *includes* relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The *includes* relationship involves one use case including the behavior of another use case in its sequence of events and actions. The *includes* relationship occurs when a chunk of behavior that is similar across a number of use cases. The factoring of such behavior will help in not repeating the specification and implementation across different use cases. Thus, the *includes* relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use cases into more manageable parts. As shown in fig. 13.4 the *includes* relationship is represented using a predefined stereotype <<include>>.In the *includes* relationship, a base use case compulsorily and automatically.

includes the behavior of the common use cases. As shown in example fig. 13.5, issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and the independent text description should be provided for it.

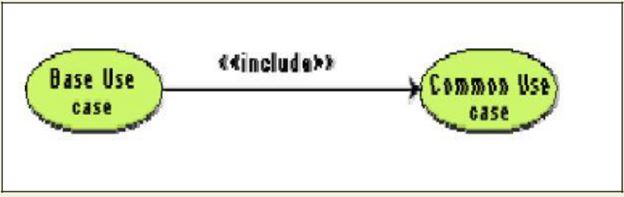


Fig. Representation of use case inclusion

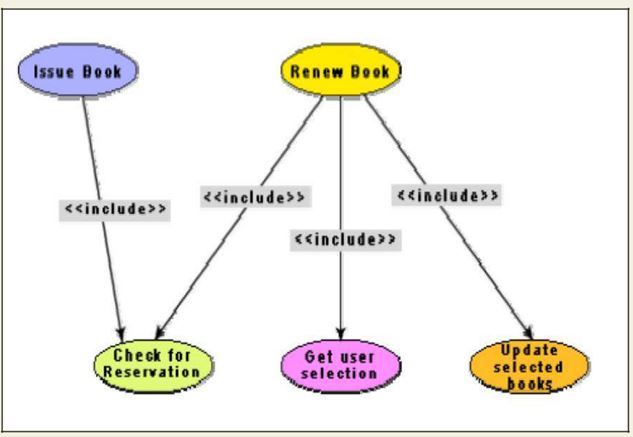


Fig. Example use case inclusion

**Extends**

The main idea behind the *extends* relationship among the use cases is that it allows you to show optional system behavior. An optional system behavior is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig. 13.6. The *extends* relationship is similar to generalization. But unlike generalization, the extending use case can add additional behavior only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The *extends* relationship is normally used to capture alternate paths or scenarios.

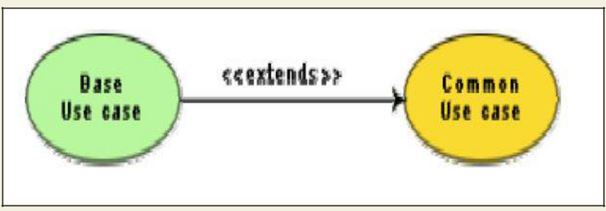


Fig. Example use case extension

**Organization of use cases**

When the use cases are factored, they are organized hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in fig. 13.7. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. In the top-level diagram, only those use cases with which external users of the system. The subsystem-level use cases specify the services offered by the subsystems. Any number of levels involving the subsystems may be utilized. In the lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

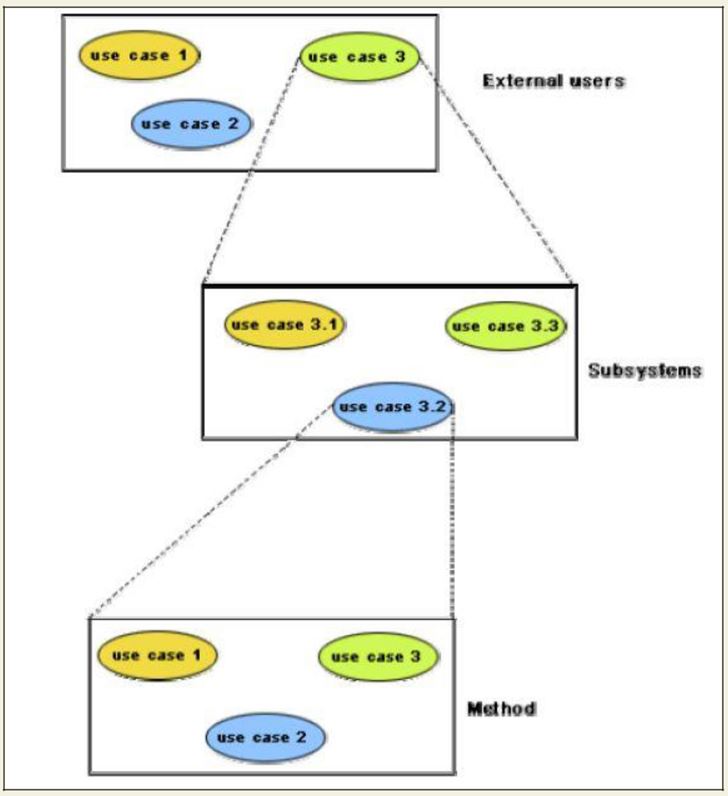


Fig: Hierarchical organization of use cases

**CLASS DIAGRAMS**

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

**Classes**

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns. Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

**Attributes**

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given.

**bookName : String**

**Operation**

Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An object’s data or state can be changed by invoking an operation of the object. A class may have any number of operations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be ‘in’, ‘out’ or ‘inout’. An operation may have a return type consisting of a single return type expression. An example for an operation is given.

**issueBook(in bookName):Boolean**

**Association**

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose Amit has borrowed the book Graph Theory. Here, borrowed is the connection between the objects Amit and Graph Theory book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship. Association between two classes is represented by drawing a straight line between the concerned classes.

Fig. 14.1 illustrates the graphical representation of the association relation. The name of the association is written alongside the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1.5. An asterisk is a wild card and means many (zero or more). The association of fig. 14.1 should be read as “Many books may be borrowed by a Library Member”. Observe that associations (and links) appear as verbs in the problem statement.

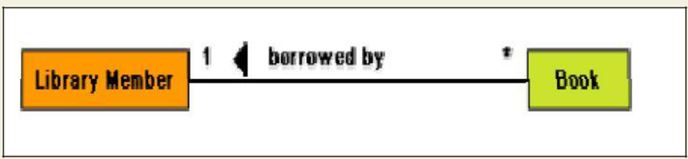
****

Fig: Association between two classes

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

**Aggregation**

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in fig.

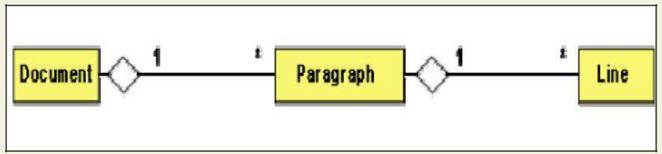


Fig. Representation of aggregation

Aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

**Composition**

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig.



Fig Representation of composition

**Association vs. Aggregation vs. Composition**

* Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation, ties the lifecycle of the part and the whole together.
* Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A cannot be a part of B).
* Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window**

whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.

* In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.
  + in general, the lifetime of parts and composite coincides
  + parts with non-fixed multiplicity may be created after composite itself
  + parts might be explicitly removed before the death of the composite

For example, when a **Frame** is created, it has to be attached to an enclosing **Window**.

Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

**Inheritance vs. Aggregation/Composition**

* Inheritance describes *‘is a’ / ‘is a kind of’* relationship between classes (base class - derived class) whereas aggregation describes *‘has a’* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation “cash payment *is a kind of* payment” is modeled using inheritance; “purchase order has a few items” ismodeled using aggregation.
* Inheritance is used to model a “generic-specific” relationship between classes whereas aggregation/composition is used to model a “whole-part” relationship between classes.
* Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of ‘payment’ in the system, we could substitute it with instances of ‘cash payment’, but the reverse cannot be done.
* Inheritance is defined statically. It cannot be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 14.4(a).But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 14.4(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a"C**ustomer\_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations thereof? The inheritance tree would be absolutely incomprehensible.

Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.

* The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.

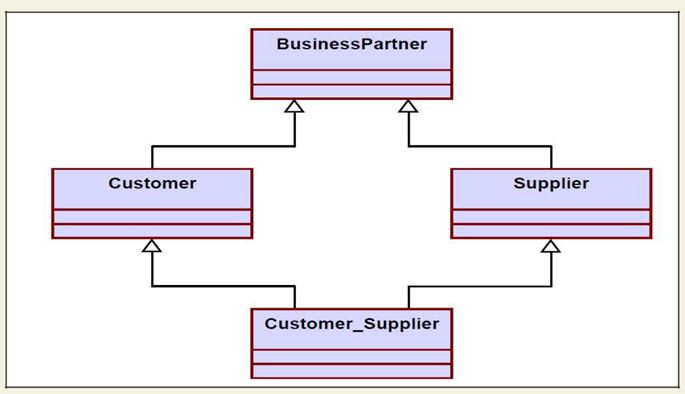


Fig.a) Representation of **BusinessPartner, Customer, Supplier** relationship using inheritance

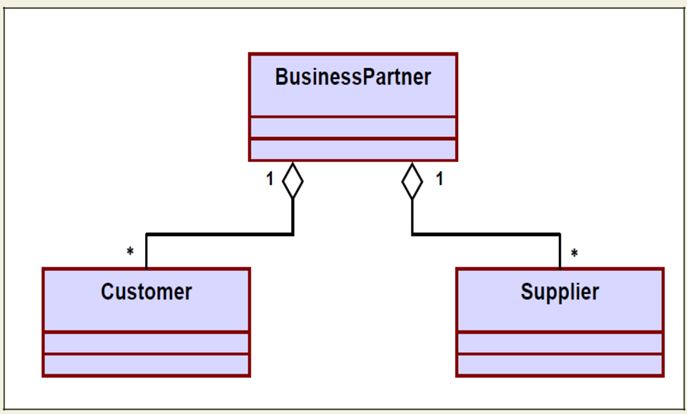


Fig. b) Representation of **BusinessPartner, Customer, Supplier** relationship using aggregation

**INTERACTION DIAGRAMS**

Interaction diagrams are models that describe how group of objects collaborate to realize some behavior. Typically, each interaction diagram realizes the behavior of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

There are two kinds of interaction diagrams: sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other. However, they are both useful. These two actually portray different perspectives of behavior of the system and different types of inferences can be drawn from them. The interaction diagrams can be considered as a major tool in the design methodology.

**Sequence Diagram**

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the use case and participates in the interaction (e.g. a method call), then the object should be shown at the appropriate place on the diagram where it is created. The vertical dashed line is called the object’s lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the life line of two objects. The messages are shown in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur. Each message is labeled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

* A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
* An iteration marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. 15.1. The development of the sequence diagram in the development methodology would help us in determining the responsibilities of the different classes; i.e. what methods should be supported by each class.

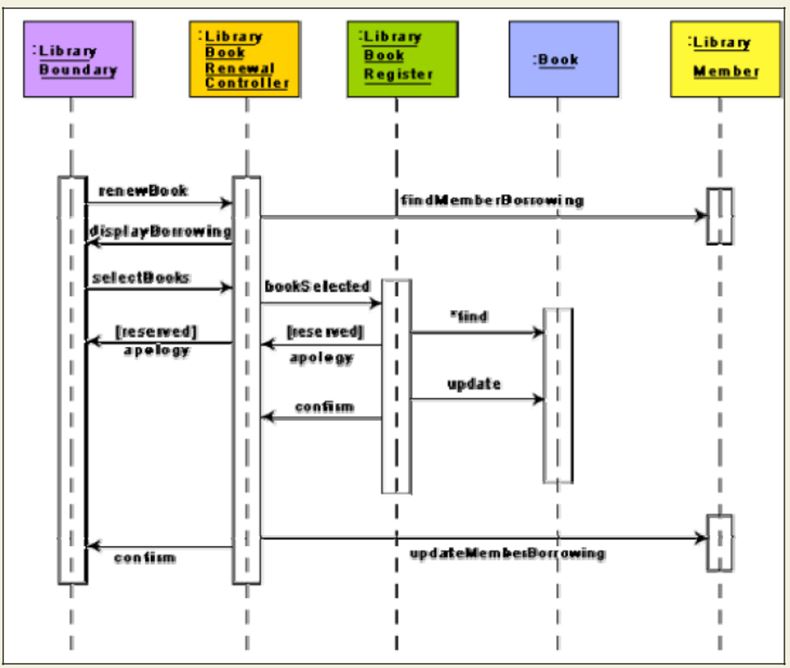


Fig. Sequence diagram for the renew book use case

**CHAPTER 4**

**CODING**

**Coding-** The objective of the coding phase is to transform the design of a system into code in ahigh level language and then to unit test this code. The programmers adhere to standard and well defined style of coding which they call their coding standard. The main advantages of adhering to a standard style of coding are as follows:

* A coding standard gives uniform appearances to the code written by different engineers
* It facilitates code of understanding.
* Promotes good programming practices.

For implementing our design into a code, we require a good high level language. A programming language should have the following features:

**Characteristics of a Programming Language**

* **Readability:** A good high-level language will allow programs to be written in some waysthat resemble a quite-English description of the underlying algorithms. If care is taken, the coding may be done in a way that is essentially self-documenting.
* **Portability:** High-level languages, being essentially machine independent, should be ableto develop portable software.
* **Generality:** Most high-level languages allow the writing of a wide variety of programs,thus relieving the programmer of the need to become expert in many diverse languages.
* **Brevity:** Language should have the ability to implement the algorithm with less amountof code. Programs expressed in high-level languages are often considerably shorter than their low-level equivalents.
* **Error checking:** Being human, a programmer is likely to make many mistakes in thedevelopment of a computer program. Many high-level languages enforce a great deal of error checking both at compile-time and at run-time.

**Cost:** The ultimate cost of a programming language is a function of many of itscharacteristics.

* **Familiar notation:** A language should have familiar notation, so it can be understood bymost of the programmers.
* **Quick translation:** It should admit quick translation.
* **Efficiency:** It should permit the generation of efficient object code.
* **Modularity:** It is desirable that programs can be developed in the language as acollection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.
* **Widely available:** Language should be widely available and it should be possible toprovide translators for all the major machines and for all the major operating systems.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

**Coding standards and guidelines**

Good software development organizations usually develop their own coding standards and guidelines depending on what best suits their organization and the type of products they develop.

The following are some representative coding standards.

1. **Rules for limiting the use of global:** These rules list what types of data can be declaredglobal and what cannot.
2. **Contents of the headers preceding codes for different modules:** The informationcontained in the headers of different modules should be standard for an organization. The exact format in which the header information is organized in the header can also be specified. The following are some standard header data:
   * Name of the module.
   * Date on which the module was created.
   * Author’s name.
   * Modification history.
   * Synopsis of the module.
   * Different functions supported, along with their input/output parameters.

Global variables accessed/modified by the module.

1. **Naming conventions for global variables, local variables, and constant identifiers:** Apossible naming convention can be that global variable names always start with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
2. **Error return conventions and exception handling mechanisms:** The way errorconditions are reported by different functions in a program are handled should be standard within an organization. For example, different functions while encountering an error condition should either return a 0 or 1 consistently.

The following are some representative coding guidelines recommended by many software development organizations.

1. **Do not use a coding style that is too clever or too difficult to understand:** Codeshould be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code. Clever coding can obscure meaning of the code and hamper understanding. It also makes maintenance difficult.
2. **Avoid obscure side effects:** The side effects of a function call include modification ofparameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code. For example, if a global variable is changed obscurely in a called module or some file I/O is performed which is difficult to infer from the function’s name and header information, it becomes difficult for anybody trying to understand the code.
3. **Do not use an identifier for multiple purposes:** Programmers often use the sameidentifier to denote several temporary entities. For example, some programmers use a temporary loop variable for computing and a storing the final result. The rationale that is usually given by these programmers for such multiple uses of variables is memory efficiency, e.g. three variables use up three memory locations, whereas the same variable used in three different ways uses just one memory location. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by use of variables for multiple purposes as follows:

Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.

* + Use of variables for multiple purposes usually makes future enhancements more difficult.

1. **The code should be well-documented:** As a rule of thumb, there must be at least onecomment line on the average for every three-source line.
2. **The length of any function should not exceed 10 source lines:** A function that is verylengthy is usually very difficult to understand as it probably carries out many different functions. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
3. **Do not use goto statements:** Use of goto statements makes a program unstructured andvery difficult to understand.

**Code Review**

Code review for a model is carried out after the module is successfully compiled and the all the syntax errors have been eliminated. Code reviews are extremely cost-effective strategies for reduction in coding errors and to produce high quality code. Normally, two types of reviews are carried out on the code of a module. These two types code review techniques are code inspection and code walk through.

**Code Walk Throughs**

Code walk through is an informal code analysis technique. In this technique, after a module has been coded, successfully compiled and all syntax errors eliminated. A few members of the development team are given the code few days before the walk through meeting to read and understand code. Each member selects some test cases and simulates execution of the code by hand (i.e. trace execution through each statement and function execution). The main objectives of the walk through are to discover the algorithmic and logical errors in the code. The members note down their findings to discuss these in a walk through meeting where the coder of the module is present. Even though a code walk through is an informal analysis technique, several guidelines have evolved over the years for making this naïve but useful analysis technique more effective. Of course, these guidelines are based on personal experience, common sense, and several subjective factors. Therefore, these guidelines should be considered as examples rather than accepted as rules to be applied dogmatically. Some of these guidelines are the following:

* The team performing code walk through should not be either too big or too small. Ideally, it should consist of between three to seven members.

Discussion should focus on discovery of errors and not on how to fix the discovered errors.

* In order to foster cooperation and to avoid the feeling among engineers that they are being evaluated in the code walk through meeting, managers should not attend the walk through meetings.

**Code Inspection**

In contrast to code walk through, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming. In other words, during code inspection the code is examined for the presence of certain kinds of errors, in contrast to the hand simulation of code execution done in code walk throughs. For instance, consider the classical error of writing a procedure that modifies a formal parameter while the calling routine calls that procedure with a constant actual parameter. It is more likely that such an error will be discovered by looking for these kinds of mistakes in the code, rather than by simply hand simulating execution of the procedure. In addition to the commonly made errors, adherence to coding standards is also checked during code inspection. Good software development companies collect statistics regarding different types of errors commonly committed by their engineers and identify the type of errors most frequently committed. Such a list of commonly committed errors can be used during code inspection to look out for possible errors.

Following is a list of some classical programming errors which can be checked during code inspection:

* Use of uninitialized variables.
* Jumps into loops.
* Nonterminating loops.
* Incompatible assignments.
* Array indices out of bounds.
* Improper storage allocation and deallocation.
* Mismatches between actual and formal parameter in procedure calls.
* Use of incorrect logical operators or incorrect precedence among operators.
* Improper modification of loop variables.
* Comparison of equally of floating point variables, etc.

**Clean Room Testing**

Clean room testing was pioneered by IBM. This type of testing relies heavily on walk throughs, inspection, and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler. The software development philosophy is based on avoiding software defects by using a rigorous inspection process. The objective of this software is zero-defect software. The name ‘clean room’ was derived from the analogy with semi-conductor fabrication units. In these units (clean rooms), defects are avoided by manufacturing in ultra-clean atmosphere. In this kind of development, inspections to check the consistency of the components with their specifications has replaced unit-testing.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing. The clean room approach to software development is based on five characteristics:

* **Formal specification:** The software to be developed is formally specified. A state-transition model which shows system responses to stimuli is used to express the specification.
* **Incremental development:** The software is partitioned into increments which aredeveloped and validated separately using the clean room process. These increments are specified, with customer input, at an early stage in the process.
* **Structured programming:** Only a limited number of control and data abstractionconstructs are used. The program development process is process of stepwise refinement of the specification.
* **Static verification:** The developed software is statically verified using rigorous softwareinspections. There is no unit or module testing process for code components
* **Statistical testing of the system:** The integrated software increment is tested statisticallyto determine its reliability. These statistical tests are based on the operational profile which is developed in parallel with the system specification. The main problem with this approach is that testing effort is increased as walk throughs, inspection, and verification are time-consuming.

**Software Documentation**

When various kinds of software products are developed then not only the executable files and the source code are developed but also various kinds of documents such as users’ manual, software requirements specification (SRS) documents, design documents, test documents, installation manual, etc are also developed as part of any software engineering process. All these documents are a vital part of good software development practice. Good documents are very useful and server the following purposes:

1. Good documents enhance understandability and maintainability of a software

product. They reduce the effort and time required for maintenance.

1. Use documents help the users in effectively using the system.
2. Good documents help in effectively handling the manpower turnover problem. Even when an engineer leaves the organization, and a new engineer comes in, he

can build up the required knowledge easily.

Production of good documents helps the manager in effectively tracking the progress of the project. The project manager knows that measurable progress is achieved if a piece of work is done and the required documents have been produced and reviewed.

Different types of software documents can broadly be classified into the following:

* Internal documentation
* External documentation

**Internal documentation** is the code comprehension features provided as part of the source codeitself. Internal documentation is provided through appropriate module headers and comments embedded in the source code. Internal documentation is also provided through the useful variable names, module and function headers, code indentation, code structuring, use of enumerated types and constant identifiers, use of user-defined data types, etc. Careful experiments suggest that out of all types of internal documentation meaningful variable names is most useful in understanding the code. This is of course in contrast to the common expectation that code commenting would be the most useful. The research finding is obviously true when comments are written without thought. For example, the following style of code commenting does not in any way help in understanding the code.

**a = 10; /\* a made 10 \*/**

But even when code is carefully commented, meaningful variable names still are more helpful in understanding a piece of code. Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines.

**External documentation** is provided through various types of supporting documents such asusers’ manual, software requirements specification document, design document, test documents, etc. A systematic software development style ensures that all these documents are produced in an orderly fashion.

**CHAPTER 5**

**TESTING**

**Program Testing**

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

Some commonly used terms associated with testing are:

* **Failure:** This is a manifestation of an error (or defect or bug). But, the mere presence ofan error may not necessarily lead to a failure.
* **Test case:** This is the triplet [I,S,O], where I is the data input to the system, S is the stateof the system at which the data is input, and O is the expected output of the system.
* **Test suite:** This is the set of all test cases with which a given software product is to betested.

**Aim of Testing**

The aim of the testing process is to identify all defects existing in a software product. However for most practical systems, even after satisfactorily carrying out the testing phase, it is not possible to guarantee that the software is error free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this practical limitation of the testing process, the importance of testing should not be underestimated. It must be remembered that testing does expose many defects existing in a software product. Thus testing provides a practical way of reducing defects in a system and increasing the users’ confidence in a developed system.

**Verification Vs Validation**

**Verification** is the process of determining whether the output of one phase of softwaredevelopment conforms to that of its previous phase, whereas **validation** is the process of determining whether a fully developed system conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is that the final product be error free.

**Design of Test Cases**

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an optional test suite that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of these randomly selected test cases do not contribute to the significance of the test suite, i.e. they do not detect any additional defects not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment which finds the greater of two integer values x and y. This code segment has a simple programming error.

**if (x>y)**

**max = x;**

**else**

**max = x;**

For the above code segment, the test suite, **{(x=3,y=2);(x=2,y=3)}** can detect the error, whereas a larger test suite **{(x=3,y=2);(x=4,y=3);(x=5,y=1)}** does not detect the error. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that the test suite should be carefully designed than picked randomly. Therefore, systematic approaches should be followed to design an optimal test suite. In an optimal test suite, each test case is designed to detect different errors.

**Functional Testing Vs. Structural Testing**

In the black-box testing approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is known as functional testing. On the other hand, in the white-box testing approach, designing test cases requires thorough knowledge about the internal structure of software, and therefore the white-box testing is called structural testing.

**BLACK-BOX TESTING**

**Testing in the large vs. testing in the small**

Software products are normally tested first at the individual component (or unit) level. This is referred to as testing in the small. After testing all the components individually, the components are slowly integrated and tested at each level of integration (integration testing). Finally, the fully integrated system is tested (called system testing). Integration and system testing are known as testing in the large.

**Unit Testing**

Unit testing is undertaken after a module has been coded and successfully reviewed. Unit testing (or module testing) is the testing of different units (or modules) of a system in isolation.

In order to test a single module, a complete environment is needed to provide all that is necessary for execution of the module. That is, besides the module under test itself, the following steps are needed in order to be able to test the module:

* The procedures belonging to other modules that the module under test calls.
* Nonlocal data structures that the module accesses.
* A procedure to call the functions of the module under test with appropriate parameters.

Modules are required to provide the necessary environment (which either call or are called by the module under test) is usually not available until they too have been unit tested, stubs and drivers are designed to provide the complete environment for a module. The role of stub and driver modules is pictorially shown in fig. 19.1. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly simplified behavior. For example, a stub procedure may produce the expected behavior using a simple table lookup mechanism. A driver module contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

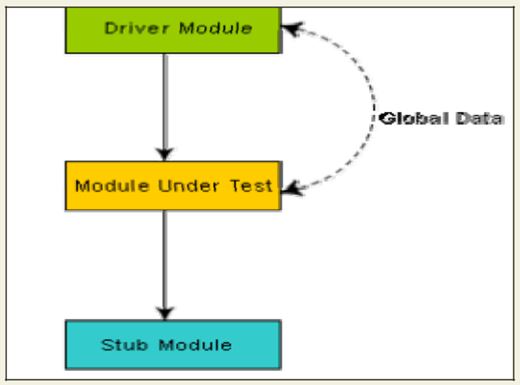


Fig. Unit testing with the help of driver and stub modules

**Black Box Testing**

In the black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. The following are the two main approaches to designing black box test cases.

* Equivalence class portioning
* Boundary value analysis

**Equivalence Class Partitioning**

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behavior of the program is similar for every input data belonging to the same equivalence class. The main idea behind defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data and output data. The following are some general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.

If the input data assumes values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

**Example 1:** For a software that computes the square root of an input integer which can assumevalues in the range of 0 to 5000, there are three equivalence classes: The set of negative integers, the set of integers in the range of 0 and 5000, and the integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes and a possible test set can be: {-5,500,6000}.

**Example 2:** Design the black-box test suite for the following program. The program computesthe intersection point of two straight lines and displays the result. It reads two integer pairs (m1, c1) and (m2, c2) defining the two straight lines of the form y=mx + c. The equivalence classes are the following:

* Parallel lines (m1=m2, c1≠c2)
* Intersecting lines (m1≠m2)
* Coincident lines (m1=m2, c1=c2)

Now, selecting one representative value from each equivalence class, the test suit (2, 2) (2, 5), (5, 5) (7, 7), (10, 10) (10, 10) are obtained.

**Boundary Value Analysis**

A type of programming error frequently occurs at the boundaries of different equivalence classes of inputs. The reason behind such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely <= for <. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

**Example:** For a function that computes the square root of integer values in the range of 0 and

5000, the test cases must include the following values: {0, -1,5000,5001}.

**WHITE-BOX TESTING**

One white-box testing strategy is said to be *stronger than* another strategy, if all types of errors detected by the first testing strategy is also detected by the second testing strategy, and the second testing strategy additionally detects some more types of errors. When two testing strategies detect errors that are different at least with respect to some types of errors, then they are called *complementary*. The concepts of stronger and complementary testing are schematically illustrated in fig.

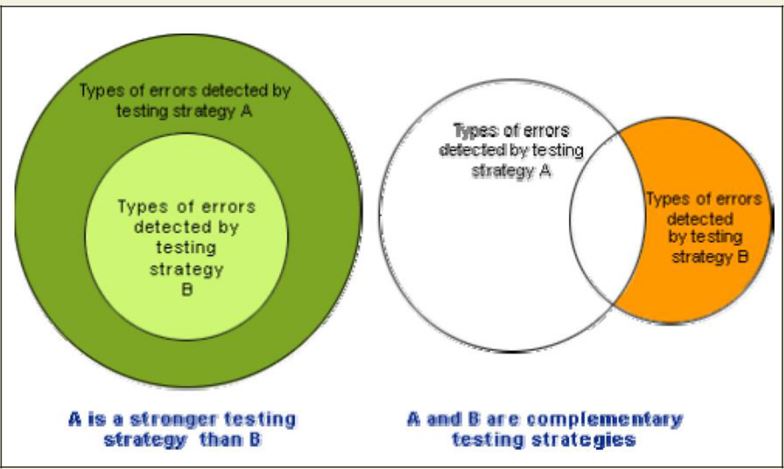


Fig. Stronger and complementary testing strategies

**Statement Coverage**

The statement coverage strategy aims to design test cases so that every statement in a program is executed at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, it is very hard to determine if an error exists in that statement. Unless a statement is executed, it is very difficult to observe whether it causes failure due to some illegal memory access, wrong result computation, etc. However, executing some statement once and observing that it behaves properly for that input value is no guarantee that it will behave correctly for all input values. In the following, designing of test cases using the statement coverage strategy have been shown.

**Example:** Consider the Euclid’s GCD computation algorithm:

int compute\_gcd(x, y)

int x, y;

{

1 while (x! = y)

{

2 if (x>y) then

3 x= x – y;

4 else y= y – x;

1. }

6 return x;

}

By choosing the test set {(x=3, y=3), (x=4, y=3), (x=3, y=4)}, we can exercise the program such that all statements are executed at least once.

**Branch Coverage**

In the branch coverage-based testing strategy, test cases are designed to make each branch condition to assume true and false values in turn. Branch testing is also known as edge testing as in this testing scheme, each edge of a program’s control flow graph is traversed at least once.

It is obvious that branch testing guarantees statement coverage and thus is a stronger testing strategy compared to the statement coverage-based testing. For Euclid’s GCD computation algorithm, the test cases for branch coverage can be {(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)}.

**Condition Coverage**

In this structural testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, in the conditional expression ((c1.and.c2).or.c3), the components c1, c2 and c3 are each made to assume both true and false values. Branch testing is probably the simplest condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. Thus, condition testing is a stronger testing strategy than branch testing and branch testing is stronger testing strategy than the statement coverage-based testing. For a composite conditional expression of n components, for condition coverage, 2ⁿ test cases are required. Thus, for condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, a condition coverage-based testing technique is practical only if n (the number of conditions) is small.

**Path Coverage**

The path coverage-based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

**Control Flow Graph (CFG)**

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph (as shown in fig. 20.2). An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node.

The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig. 20.2 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the loop. Using these basic ideas, the CFG of Euclid’s GCD computation algorithm can be drawn.

**Path**

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

**Linearly independent path**

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because; any path having a new node automatically implies that it has a new edge. Thus, a path that is sub-path of another path is not considered to be a linearly independent path.

**Control Flow Graph**

In order to understand the path coverage-based testing strategy, it is very much necessary to understand the control flow graph (CFG) of a program. Control flow graph (CFG) of a program has been discussed earlier.

**Linearly Independent Path**

The path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths. Linearly independent paths have been discussed earlier.

**Cyclomatic Complexity**

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe’s cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe’s cyclomatic complexity is very simple to compute. Thus, the McCabe’s cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe’s metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

There are three different ways to compute the cyclomatic complexity. The answers computed by the three methods are guaranteed to agree.

**Method 1:**

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

**V(G)=E–N+2**

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in fig. 20.3, E=7 and N=6. Therefore, the cyclomatic complexity = 7-6+2 = 3.

**Method 2:**

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

**V(G) = Total number of bounded areas + 1**

In the program’s control flow graph G, any region enclosed by nodes and edges can be called as a bounded area. This is an easy way to determine the McCabe’s cyclomatic complexity.

But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO’s can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe’s cyclomatic complexity cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe’s metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the CFG example shown in fig. 20.3, from a visual examination of the CFG the number of bounded areas is 2. Therefore the cyclomatic complexity, computing with this method is also 2+1 = 3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

**Method 3:**

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe’s metric is equal to N+1.

**Data Flow Based Testing:**

Data flow-based testing method selects test paths of a program according to the locations of the definitions and uses of different variables in a program.

For a statement numbered S, let

**DEF(S) = {X/statement S contains a definition of X}, and USES(S) = {X/statement S contains a use of X}**

For the statement **S:a=b+c;,** DEF(S) = {**a**}. USES(S) = {**b,c**}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.

The *definition-use chain* (or DU chain) of a variable X is of form [X, S, S1], where S and S1 are statement numbers, such that X Є DEF(S) and X Є USES(S1), and the definition of X in the statement S is live at statement S1. One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements.

**Mutation Testing**

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant. The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc. A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated.

Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.

**DEBUGGING, INTEGRATION AND SYSTEM TESTING**

**Need for Debugging**

Once errors are identified in a program code, it is necessary to first identify the precise program statements responsible for the errors and then to fix them. Identifying errors in a program code and then fix them up are known as debugging.

**Debugging Approaches**

The following are some of the approaches popularly adopted by programmers for debugging.

**Brute Force Method:**

This is the most common method of debugging but is the least efficient method. In this approach, the program is loaded with print statements to print the intermediate values with the hope that some of the printed values will help to identify the statement in error. This approach becomes more systematic with the use of a symbolic debugger (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.

**Backtracking:**

This is also a fairly common approach. In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered. Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large thus limiting the use of this approach.

**Cause Elimination Method:**

In this approach, a list of causes which could possibly have contributed to the error symptom is developed and tests are conducted to eliminate each. A related technique of identification of the error from the error symptom is the software fault tree analysis.

**Program Slicing:**

This technique is similar to back tracking. Here the search space is reduced by defining slices. A slice of a program for a particular variable at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

**Debugging Guidelines:**

Debugging is often carried out by programmers based on their ingenuity. The following are some general guidelines for effective debugging:

* Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the system design and implementation may require an inordinate amount of effort to be put into debugging even simple problems.
* Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
* One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

**Program Analysis Tools**

A program analysis tool means an automated tool that takes the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, etc. We can classify these into two broad categories of program analysis tools:

* Static Analysis tools
* Dynamic Analysis tools
* Static program analysis tools

***Static Analysis Tool*** is also a program analysis tool. It assesses and computes variouscharacteristics of a software product without executing it. Typically, static analysis tools analyze some structural representation of a program to arrive at certain analytical conclusions, e.g. that some structural properties hold. The structural properties that are usually analyzed are:

* Whether the coding standards have been adhered to?
* Certain programming errors such as uninitialized variables and mismatch between actual and formal parameters, variables that are declared but never used are also checked.

Code walk throughs and code inspections might be considered as static analysis methods. But, the term static program analysis is used to denote automated analysis tools. So, a compiler can be considered to be a static program analysis tool.

***Dynamic program analysis tools -*** Dynamic program analysis techniques require the program tobe executed and its actual behavior recorded. A dynamic analyzer usually instruments the code (i.e. adds additional statements in the source code to collect program execution traces). The instrumented code when executed allows us to record the behavior of the software for different test cases. After the software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool caries out a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete test suite for the program. For example, the post execution dynamic analysis report might provide data on extent statement, branch and path coverage achieved.

Normally the dynamic analysis results are reported in the form of a histogram or a pie chart to describe the structural coverage achieved for different modules of the program. The output of a dynamic analysis tool can be stored and printed easily and provides evidence that thorough testing has been done. The dynamic analysis results the extent of testing performed in white-box mode. If the testing coverage is not satisfactory more test cases can be designed and added to the test suite. Further, dynamic analysis results can help to eliminate redundant test cases from the test suite.

**INTEGRATION TESTING**

The primary objective of integration testing is to test the module interfaces, i.e. there are no errors in the parameter passing, when one module invokes another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the steps and the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested. An important factor that guides the integration plan is the module dependency graph. The structure chart (or module dependency graph) denotes the order in which different modules call each other. By examining the structure chart the integration plan can be developed. Integration test approaches

There are four types of integration testing approaches. Any one (or a mixture) of the following approaches can be used to develop the integration test plan. Those approaches are the following:

* Big bang approach
* Bottom- up approach
* Top-down approach
* Mixed-approach

**Big-Bang Integration Testing**

It is the simplest integration testing approach, where all the modules making up a system are integrated in a single step. In simple words, all the modules of the system are simply put together and tested. However, this technique is practicable only for very small systems. The main problem with this approach is that once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. Therefore, debugging errors reported during big bang integration testing are very expensive to fix.

**Bottom-Up Integration Testing**

In bottom-up testing, each subsystem is tested separately and then the full system is tested. A subsystem might consist of many modules which communicate among each other through well-defined interfaces. The primary purpose of testing each subsystem is to test the interfaces among various modules making up the subsystem. Both control and data interfaces are tested. The test cases must be carefully chosen to exercise the interfaces in all possible manners Large software systems normally require several levels of subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems. A principal advantage of bottom-up integration testing is that several disjoint subsystems can be tested simultaneously. In a pure bottom-up testing no stubs are required, only test-drivers are required. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems. The extreme case corresponds to the big-bang approach.

**Top- Down Integration Testing:**

Top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level ‘skeleton’ has been tested, the immediately subroutines of the ‘skeleton’ are combined with it and tested. Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test. A pure top-down integration does not require any driver routines. A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, many times it may become difficult to exercise the top-level routines in the desired manner since the lower-level routines perform several low-level functions such as I/O.

**Mixed Integration Testing**

A mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Similarly, bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. In the mixed testing approaches, testing can start as and when modules become available. Therefore, this is one of the most commonly used integration testing approaches.

**Phased Vs. Incremental Testing**

The different integration testing strategies are either phased or incremental. A comparison of these two strategies is as follows:

1. In incremental integration testing, only one new module is added to the partial system each time.
2. In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system in the incremental testing approach since it is known that the error is caused by addition of a single module. In fact, big bang testing is a degenerate case of the phased integration testing approach. System testing

System tests are designed to validate a fully developed system to assure that it meets its requirements. There are essentially three main kinds of system testing:

* **Alpha Testing.** Alpha testing refers to the system testing carried out by the test teamwithin the developing organization.
* **Beta testing.** Beta testing is the system testing performed by a select group of friendlycustomers.

**Acceptance Testing.** Acceptance testing is the system testing performed by the customerto determine whether he should accept the delivery of the system.

In each of the above types of tests, various kinds of test cases are designed by referring to the SRS document. Broadly, these tests can be classified into functionality and performance tests. The functionality test tests the functionality of the software to check whether it satisfies the functional requirements as documented in the SRS document. The performance test tests the conformance of the system with the nonfunctional requirements of the system.

**Performance Testing**

Performance testing is carried out to check whether the system needs the non-functional requirements identified in the SRS document. There are several types of performance testing. Among of them nine types are discussed below. The types of performance testing to be carried out on a system depend on the different non-functional requirements of the system documented in the SRS document. All performance tests can be considered as black-box tests.

* Stress testing
* Volume testing
* Configuration testing
* Compatibility testing
* Regression testing
* Recovery testing
* Maintenance testing
* Documentation testing
* Usability testing

**Stress Testing** -Stress testing is also known as ***endurance testing***. Stress testingevaluates system performance when it is stressed for short periods of time. Stress tests are black box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software. Input data volume, input data rate, processing time, utilization of memory, etc. are tested beyond the designed capacity. For example, suppose an operating system is supposed to support 15 multi programmed jobs, the system is stressed by attempting to run 15 or more jobs simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.

Stress testing is especially important for systems that usually operate below the maximum capacity but are severely stressed at some peak demand hours. For example, if the non-functional requirement specification states that the response time should not be more than 20 secs per transaction when 60 concurrent users are working, then during the stress testing the response time is checked with 60 users working simultaneously.

**Volume Testing-**It is especially important to check whether the data structures (arrays,queues, stacks, etc.) have been designed to successfully extraordinary situations. For example, a compiler might be tested to check whether the symbol table overflows when a very large program is compiled.

**Configuration Testing** - This is used to analyze system behavior in various hardwareand software configurations specified in the requirements. Sometimes systems are built in variable configurations for different users. For instance, we might define a minimal system to serve a single user, and other extension configurations to serve additional users. The system is configured in each of the required configurations and it is checked if the system behaves correctly in all required configurations.

**Compatibility Testing -**This type of testing is required when the system interfaces withother types of systems. Compatibility aims to check whether the interface functions perform as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

**Regression Testing -**This type of testing is required when the system being tested is anupgradation of an already existing system to fix some bugs or enhance functionality, performance, etc. Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix. However, if only a few statements are changed, then the entire test suite need not be run - only those test cases that test the functions that are likely to be affected by the change need to be run.

**Recovery Testing -**Recovery testing tests the response of the system to the presence offaults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as applicable and discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

**Maintenance Testing-** This testing addresses the diagnostic programs, and otherprocedures that are required to be developed to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

**Documentation Testing-** It is checked that the required user manual, maintenancemanuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance.

**Usability Testing-** Usability testing concerns checking the user interface to see if itmeets all user requirements concerning the user interface. During usability testing, the display screens, report formats, and other aspects relating to the user interface requirements are tested.

**Error Seeding**

Sometimes the customer might specify the maximum number of allowable errors that may be present in the delivered system. These are often expressed in terms of maximum number of allowable errors per line of source code. Error seed can be used to estimate the number of residual errors in a system. Error seeding, as the name implies, seeds the code with some known errors. In other words, some artificial errors are introduced into the program artificially. The number of these seeded errors detected in the course of the standard testing procedure is determined. These values in conjunction with the number of unseeded errors detected can be used to predict:

* The number of errors remaining in the product.
* The effectiveness of the testing strategy.

Let N be the total number of defects in the system and let n of these defects be found by testing.

Let S be the total number of seeded defects, and let s of these defects be found during testing.

**n/N = s/S**

**or**

**N = S** × **n/s**

Defects still remaining after testing = **N–n = n**×**(S** **–** **s)/s**

Error seeding works satisfactorily only if the kind of seeded errors matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that remain can be estimated to a first approximation by analyzing historical data of similar projects. Due to the shortcoming that the types of seeded errors should match closely with the types of errors actually existing in the code, error seeding is useful only to a moderate extent.

**Regression Testing**

Regression testing does not belong to either unit test, integration test, or system testing. Instead, it is a separate dimension to these three forms of testing. The functionality of regression testing has been discussed earlier.

**CHAPTER 6**

**SOFTWARE MAINTENANCE**

**Necessity of Software Maintenance**

Software maintenance is becoming an important activity of a large number of software organizations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product per se, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features. When the hardware platform is changed, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface. For instance, a software product may need to be maintained when the operating system changes. Thus, every software product continues to evolve after its development through maintenance efforts. Therefore it can be stated that software maintenance is needed to correct errors, enhance features, port the software to new platforms, etc.

**Types of software maintenance**

There are basically three types of software maintenance. These are:

* **Corrective:** Corrective maintenance of a software product is necessary to rectify the bugsobserved while the system is in use.
* **Adaptive:** A software product might need maintenance when the customers need theproduct to run on new platforms, on new operating systems, or when they need the product to interface with new hardware or software.
* **Perfective:** A software product needs maintenance to support the new features that userswant it to support, to change different functionalities of the system according to customer demands, or to enhance the performance of the system.

**Problems associated with software maintenance**

Software maintenance work typically is much more expensive than what it should be and takes more time than required. In software organizations, maintenance work is mostly carried out using ad hoc techniques. The primary reason being that software maintenance is one of the most neglected areas of software engineering. Even though software maintenance is fast becoming an important area of work for many companies as the software products of yester years age, still software maintenance is mostly being carried out as fire-fighting operations, rather than through systematic and planned activities.

Software maintenance has a very poor image in industry. Therefore, an organization often cannot employ bright engineers to carry out maintenance work. Even though maintenance suffers from a poor image, the work involved is often more challenging than development work. During maintenance it is necessary to thoroughly understand someone else’s work and then carry out the required modifications and extensions.

Another problem associated with maintenance work is that the majority of software products needing maintenance are legacy products.

**Software Reverse Engineering**

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing of its functionalities. A process model for reverse engineering has been shown in fig. 24.1. A program can be reformatted using any of the several available prettyprinter programs which layout the program neatly. Many legacy software products with complex control structure and unthoughtful variable names are difficult to comprehend. Assigning meaningful variable names is important because meaningful variable names are the most helpful thing in code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible. Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

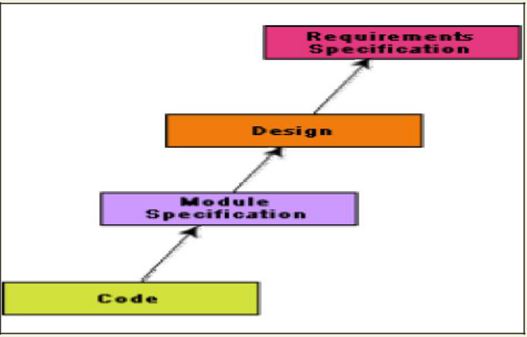
****

Fig. A process model for reverse engineering

**Legacy software products**

It is prudent to define a legacy system as any software system that is hard to maintain. The typical problems associated with legacy systems are poor documentation, unstructured (spaghetti code with ugly control structure), and lack of personnel knowledgeable in the product. Many of the legacy systems were developed long time back. But, it is possible that a recently developed system having poor design and documentation can be considered to be a legacy system.

The activities involved in a software maintenance project are not unique and depend on several factors such as:

• the extent of modification to the product required

* The resources available to the maintenance team
* the conditions of the existing product (e.g., how structured it is, how well documented it is, etc.)
* the expected project risks, etc.

When the changes needed to a software product are minor and straightforward, the code can be directly modified and the changes appropriately reflected in all the documents. But more elaborate activities are required when the required changes are not so trivial. Usually, for complex maintenance projects for legacy systems, the software process can be represented by a reverse engineering cycle followed by a forward engineering cycle with an emphasis on as much reuse as possible from the existing code and other documents.

**SOFTWARE RELIABILITY AND QUALITY MANAGEMENT**

**Repeatable vs. non-repeatable software development organization**

A repeatable software development organization is one in which the software development process is person-independent. In a non-repeatable software development organization, a software development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. Thus, in a non-repeatable software development organization, the chances of successful completion of a software project is to a great extent depends on the team members.

**Software Reliability**

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing 60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently the corresponding instruction is executed.

Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the “correctly” implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

**Reasons for software reliability being difficult to measure**

The reasons why software reliability is difficult to measure can be summarized as follows:

* The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
* The perceived reliability of a software product is highly observer-dependent.
* The reliability of a product keeps changing as errors are detected and fixed.
* Hardware reliability vs. software reliability differs.

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase). The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 26.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics “bath tub” shape. On the other hand, for software the failure rate is at it’s highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.

**SOFTWARE QUALITY**

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.

The modern view of a quality associates with a software product several quality factors such as the following:

* **Portability:** A software product is said to be portable, if it can be easily made to work indifferent operating system environments, in different machines, with other software products, etc.
* **Usability:** A software product has good usability, if different categories of users (i.e. bothexpert and novice users) can easily invoke the functions of the product.
* **Reusability:** A software product has good reusability, if different modules of the productcan easily be reused to develop new products.
* **Correctness:** A software product is correct, if different requirements as specified in theSRS document have been correctly implemented.
* **Maintainability:** A software product is maintainable, if errors can be easily corrected asand when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

**Software Quality Management System**

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality.

A quality system consists of the following:

**Managerial Structure and Individual Responsibilities-** A quality system is actually theresponsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously. **Quality System Activities-** The quality system activities encompass the following:

* auditing of projects
* review of the quality system
* development of standards, procedures, and guidelines, etc.
* production of reports for the top management summarizing the effectiveness of the quality system in the organization.

**Evolution of Quality Management System**

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. 28.1. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization’s processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements. TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance.

**ISO 9000 certification**

ISO (International Standards Organization) is a consortium of 63 countries established to formulate and foster standardization. ISO published its 9000 series of standards in 1987. ISO certification serves as a reference for contract between independent parties. The ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organization applies to all activities related to its product or service. The ISO standard mainly addresses operational aspects and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 specifies a set of guidelines for repeatable and high quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.

**Types of ISO 9000 quality standards**

ISO 9000 is a series of three standards: ISO 9001, ISO 9002, and ISO 9003. The ISO 9000 series of standards is based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically. The types of industries to which the different ISO standards apply are as follows.

ISO 9001 applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organizations.

ISO 9002 applies to those organizations which do not design products but are only involved in production. Examples of these category industries include steel and car manufacturing industries that buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organizations.

ISO 9003 applies to organizations that are involved only in installation and testing of the products.

**Software products vs. other products**

There are mainly two differences between software products and any other type of products.

* Software is intangible in nature and therefore difficult to control. It is very difficult to control and manage anything that is not seen. In contrast, any other industries such as car manufacturing industries where one can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it is easy to accurately determine how much work has been completed and to estimate how much more time will it take.
* During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product.

**Need for obtaining ISO 9000 certification**

There is a mad scramble among software development organizations for obtaining ISO certification due to the benefits it offers. Some benefits that can be acquired to organizations by obtaining ISO certification are as follows:

* Confidence of customers in an organization increases when organization qualifies for ISO certification. This is especially true in the international market. In fact, many organizations awarding international software development contracts insist that the development organization have ISO 9000 certification. For this reason, it is vital for software organizations involved in software export to obtain ISO 9000 certification.
* ISO 9000 requires a well-documented software production process to be in place. A well-documented software production process contributes to repeatable and higher quality of the developed software.
* ISO 9000 makes the development process focused, efficient, and cost-effective.
* ISO 9000 certification points out the weak points of an organization and recommends remedial action.
* ISO 9000 sets the basic framework for the development of an optimal process and Total Quality Management (TQM).

**Summary of ISO 9001 certification**

A summary of the main requirements of ISO 9001 as they relate of software development is as follows. Section numbers in brackets correspond to those in the standard itself:

**Management Responsibility (4.1)**

* The management must have an effective quality policy.
* The responsibility and authority of all those whose work affects quality must be defined and documented.
* A management representative, independent of the development process, must be responsible for the quality system. This requirement probably has been put down so that the person responsible for the quality system can work in an unbiased manner.
* The effectiveness of the quality system must be periodically reviewed by audits.

**Quality System (4.2)**

A quality system must be maintained and documented.

**Contract Reviews (4.3)**

Before entering into a contract, an organization must review the contract to ensure that it is understood, and that the organization has the necessary capability for carrying out its obligations.

**Design Control (4.4)**

* The design process must be properly controlled, this includes controlling coding also. This requirement means that a good configuration control system must be in place.
* Design inputs must be verified as adequate.
* Design must be verified.
* Design output must be of required quality.
* Design changes must be controlled.

**Document Control (4.5)**

* There must be proper procedures for document approval, issue and removal.
* Document changes must be controlled. Thus, use of some configuration management

tools is necessary.

**Purchasing (4.6)**

Purchasing material, including bought-in software must be checked for conforming to requirements.

**Purchaser Supplied Product (4.7)**

Material supplied by a purchaser, for example, client-provided software must be properly managed and checked.

**Product Identification (4.8)**

The product must be identifiable at all stages of the process. In software terms this means configuration management.

**Process Control (4.9)**

* The development must be properly managed.
* Quality requirement must be identified in a quality plan.

**Inspection and Testing (4.10)**

In software terms this requires effective testing i.e., unit testing, integration testing and system testing. Test records must be maintained.

**Inspection, Measuring and Test Equipment (4.11)**

If integration, measuring, and test equipments are used, they must be properly maintained and calibrated.

**Inspection and Test Status (4.12)**

The status of an item must be identified. In software terms this implies configuration management and release control.

**Control of Nonconforming Product (4.13)**

In software terms, this means keeping untested or faulty software out of the released product, or other places whether it might cause damage.

**Corrective Action (4.14)**

This requirement is both about correcting errors when found, and also investigating why the errors occurred and improving the process to prevent occurrences. If an error occurs despite the quality system, the system needs improvement.

**Handling, (4.15)**

This clause deals with the storage, packing, and delivery of the software product.

**Quality records (4.16)**

Recording the steps taken to control the quality of the process is essential in order to be able to confirm that they have actually taken place.

**Quality Audits (4.17)**

Audits of the quality system must be carried out to ensure that it is effective.

**Training (4.18)**

Training needs must be identified and met.

**Salient features of ISO 9001 certification**

The salient features of ISO 9001 are as follows:

* All documents concerned with the development of a software product should be properly managed, authorized, and controlled. This requires a configuration management system to be in place.
* Proper plans should be prepared and then progress against these plans should be monitored.
* Important documents should be independently checked and reviewed for effectiveness and correctness.
* The product should be tested against specification.
* Several organizational aspects should be addressed e.g., management reporting of the quality team.

**Shortcomings of ISO 9000 certification**

Even though ISO 9000 aims at setting up an effective quality system in an organization, it suffers from several shortcomings. Some of these shortcomings of the ISO 9000 certification process are the following:

* ISO 9000 requires a software production process to be adhered to but does not guarantee the process to be of high quality. It also does not give any guideline for defining an appropriate process.
* ISO 9000 certification process is not fool-proof and no international accreditation agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
* Organizations getting ISO 9000 certification often tend to downplay domain expertise. These organizations start to believe that since a good process is in place, any engineer is as effective as any other engineer in doing any particular activity relating to software development. However, many areas of software development are so specialized that special expertise and experience in these areas (domain expertise) is required. In manufacturing industry there is a clear link between process quality and product quality. Once a process is calibrated, it can be run again and again producing quality goods. In contrast, software development is a creative process and individual skills and experience are important.
* ISO 9000 does not automatically lead to continuous process improvement, i.e. does not automatically lead to TQM.

**SEI CAPABILITY MATURITY MODEL**

SEI Capability Maturity Model (SEI CMM) helped organizations to improve the quality of the software they develop and therefore adoption of SEI CMM model has significant business benefits.

SEI CMM can be used two ways: capability evaluation and software process assessment. Capability evaluation and software process assessment differ in motivation, objective, and the final use of the result. Capability evaluation provides a way to assess the software process capability of an organization. The results of capability evaluation indicates the likely contractor performance if the contractor is awarded a work. Therefore, the results of software process capability assessment can be used to select a contractor. On the other hand, software process assessment is used by an organization with the objective to improve its process capability. Thus, this type of assessment is for purely internal use.

SEI CMM classifies software development industries into the following five maturity levels. The different levels of SEI CMM have been designed so that it is easy for an organization to slowly build its quality system starting from scratch.

**Level 1: Initial -** A software development organization at this level is characterized by ad hocactivities. Very few or no processes are defined and followed. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic. Therefore, it is also called chaotic level. The success of projects depends on individual efforts and heroics. When engineers leave, the successors have great difficulty in understanding the process followed and the work completed. Since formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

**Level 2: Repeatable -** At this level, the basic project management practices such as tracking costand schedule are established. Size and cost estimation techniques like function point analysis, COCOMO, etc. are used. The necessary process discipline is in place to repeat earlier success on projects with similar applications. Please remember that opportunity to repeat a process exists only when a company produces a family of products.

**Level 3: Defined -** At this level the processes for both management and development activitiesare defined and documented. There is a common organization-wide understanding of activities, roles, and responsibilities. The processes though defined, the process and product qualities are not measured. ISO 9000 aims at achieving this level.

**Level 4: Managed -** At this level, the focus is on software metrics. Two types of metrics arecollected. Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc. Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc. Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality. The process metrics are used to check if a project performed satisfactorily. Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

**Level 5: Optimizing -** At this stage, process and product metrics are collected. Process andproduct measurement data are analyzed for continuous process improvement. For example, if from an analysis of the process measurement results, it was found that the code reviews were not very effective and a large number of errors were detected only during the unit testing, then the process may be fine-tuned to make the review more effective. Also, the lessons learned from specific projects are incorporated in to the process. Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies. Such an organization identifies the best software engineering practices and innovations which may be tools, methods, or processes. These best practices are transferred throughout the organization.

**Key process areas (KPA) of a software organization**

Except for SEI CMM level 1, each maturity level is characterized by several Key Process Areas (KPAs) that includes the areas an organization should focus to improve its software process to the next level. The focus of each level and the corresponding key process areas are shown in the fig.

A **pplicability of SEI CMM to organizations**

Highly systematic and measured approach to software development suits large organizations dealing with negotiated software, safety-critical software, etc. For those large organizations, SEI CMM model is perfectly applicable. But small organizations typically handle applications such as Internet, e-commerce, and are without an established product range, revenue base, and experience on past projects, etc. For such organizations, a CMM-based appraisal is probably excessive. These organizations need to operate more efficiently at the lower levels of maturity. For example, they need to practice effective project management, reviews, configuration management, etc.

**Personal Software Process**

Personal Software Process (PSP) is a scaled down version of the industrial software process. PSP is suitable for individual use. It is important to note that SEI CMM does not tell software developers how to analyze, design, code, test, or document software products, but assumes that engineers use effective personal practices. PSP recognizes that the process for individual use is different from that necessary for a team.

The quality and productivity of an engineer is to a great extent dependent on his process. PSP is a framework that helps engineers to measure and improve the way they work. It helps in developing personal skills and methods by estimating and planning, by showing how to track performance against plans, and provides a defined process which can be tuned by individuals.

**Time measurement-** PSP advocates that engineers should rack the way they spend time.Because, boring activities seem longer than actual and interesting activities seem short. Therefore, the actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. For example, he may stop the clock when attending a telephone call, taking a coffee break etc. An engineer should measure the time he spends for designing, writing code, testing, etc.

**PSP Planning-** Individuals must plan their project. They must estimate the maximum, minimum,and the average LOC required for the product. They should use their productivity in minutes/LOC to calculate the maximum, minimum, and the average development time. They must record the plan data in a project plan summary.

The PSP is schematically shown in fig. 29.2. While carrying out the different phases, they must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve their process, etc.

**SOFTWARE PROJECT PLANNING**

**Project Planning and Project Estimation Techniques**

**Responsibilities of a software project manager**

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

**Skills necessary for software project management**

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

**Project Planning**

Once a project is found to be feasible, software project managers undertake project planning.

Project planning is undertaken and completed even before any development activity starts.

Project planning consists of the following essential activities:

* Estimating the following attributes of the project:
  + **Project size**: What will be problem complexity in terms of the effort and timerequired to develop the product?
  + **Cost**: How much is it going to cost to develop the project?
* **Duration**: How long is it going to take to complete development?
* **Effort**: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

* Scheduling manpower and other resources.
* Staff organization and staffing plans.
* Risk identification, analysis, and abatement planning
* Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

**Software Project Management Plan (SPMP)**

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. This list can be used as a possible organization of the SPMP document.

Organization of the Software Project Management Plan (SPMP) Document

1. **Introduction**
   1. Objectives
   2. Major Functions
   3. Performance Issues
   4. Management and Technical Constraints
2. **Project Estimates**
   1. Historical Data Used
   2. Estimation Techniques Used
   3. Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   1. Work Breakdown Structure
   2. Task Network Representation
   3. Gantt Chart Representation
   4. PERT Chart Representation

 **Project Resources**

* 1. People
  2. Hardware and Software
  3. Special Resources

1. **Staff Organization**
   1. Team Structure
   2. Management Reporting
2. **Risk Management Plan**
   1. Risk Analysis
   2. Risk Identification
   3. Risk Estimation
   4. Risk Abatement Procedures
3. **Project Tracking and Control Plan**
4. **Miscellaneous Plans**
   1. Process Tailoring
   2. Quality Assurance Plan
   3. Configuration Management Plan
   4. Validation and Verification
   5. System Testing Plan
   6. Delivery, Installation, and Maintenance Plan

**METRICS FOR SOFTWARE PROJECT SIZE ESTIMATION**

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

**Lines of Code (LOC)**

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

**Function point (FP)**

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed. The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data.

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

**UFP = (Number of inputs)\*4 + (Number of outputs)\*5 + (Number of inquiries)\*4 + (Number of files)\*10 + (Number of interfaces)\*10**

**Number of inputs:** Each data item input by the user is counted. Data inputs should bedistinguished from user inquiries. Inquiries are user commands such as print-account-balance.

Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input. For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

**Number of outputs:** The outputs considered refer to reports printed, screen outputs, errormessages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

**Number of inquiries:** Number of inquiries is the number of distinct interactive queries whichcan be made by the users. These inquiries are the user commands which require specific action by the system.

**Number of files:** Each logical file is counted. A logical file means groups of logically relateddata. Thus, logical files can be data structures or physical files.

**Number of interfaces:** Here the interfaces considered are the interfaces used to exchangeinformation with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01\*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP\*TCF.

**Shortcomings of function point (FP) metric**

LOC as a measure of problem size has several shortcomings:

* LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.
* A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.
* LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
* LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.
* LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.
* It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager’s perspective.

**Feature Point Metric**

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed. Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

**Project Estimation Techniques**

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

* Empirical estimation techniques
* Heuristic techniques
* Analytical estimation techniques

**Empirical Estimation Techniques**

Empirical estimation techniques are based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: ***Expert judgment technique*** and

***Delphi cost estimation.***

***Expert Judgment Technique***

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part.

A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

***Delphi Cost Estimation***

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the software requirements specification (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

**HEURISTIC TECHNIQUES**

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

**d**

Estimated Parameter = **c1** **\* e** **1**

In the above expression, e is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c1 and d1 are

constants. The values of the constants c1 and d1 are usually determined using data collected from

past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **d** |  |  | **d** |
| Estimated Resource = **c \*e** |  |  | **+ c \*e** | | **+ ...** |
| **1** | **1** | **1** | **2** | **2** | **2** |

Where e1, e2, … are the basic (independent) characteristics of the software already estimated, and

c1, c2, d1, d2, … are constants. Multivariable estimation models are expected to give more

accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants c1, c2, d1, d2, … .

Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

**Analytical Estimation Techniques**

Analytical estimation techniques derive the required results starting with basic assumptions regarding the project. Thus, unlike empirical and heuristic techniques, analytical techniques do have scientific basis. Halstead’s software science is an example of an analytical technique. Halstead’s software science can be used to derive some interesting results starting with a few

simple assumptions. Halstead’s software science is especially useful for estimating software maintenance efforts. In fact, it outperforms both empirical and heuristic techniques when used for predicting software maintenance efforts.

**Halstead’s Software Science – An Analytical Technique**

Halstead’s software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for overall program length, potential minimum value, actual volume, effort, and development time.

For a given program, let:

* η1 be the number of unique operators used in the program,
* η2 be the number of unique operands used in the program,
* N1 be the total number of operators used in the program,
* N2 be the total number of operands used in the program.

**Length and Vocabulary**

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length N = N1 +N2. Halstead’s definition of the length of the

program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program.

Thus, *program vocabulary* η = η1 + η2.

**Program Volume**

The length of a program (i.e. the total number of operators and operands used in the code) depends on the choice of the operators and operands used. In other words, for the same programming problem, the length would depend on the programming style. This type of dependency would produce different measures of length for essentially the same problem when different programming languages are used. Thus, while expressing program size, the programming language used must be taken into consideration:

V = Nlog2η

Here the program volume V is the minimum number of bits needed to encode the program. In fact, to represent η different identifiers uniquely, at least log2η bits (where η is the program

vocabulary) will be needed. In this scheme, Nlog2η bits will be needed to store a program of

length N. Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

**Potential Minimum Volume**

The potential minimum volume V\* is defined as the volume of most succinct program in which a problem can be coded. The minimum volume is obtained when the program can be expressed using a single source code instruction. say a function call like foo( ) ;. In other words, the volume is bound from below due to the fact that a program would have at least two operators and no less than the requisite number of operands.

Thus, if an algorithm operates on input and output data d1, d2, … dn, the most succinct program would be *f*(d1, d2, … dn); for which η1 = 2, η2 = n. Therefore, V\* = (2 + η2)log2(2 + η2).

The program level L is given by L = V\*/V. The concept of program level L is introduced in an attempt to measure the level of abstraction provided by the programming language. Using this definition, languages can be ranked into levels that also appear intuitively correct.

The above result implies that the higher the level of a language, the less effort it takes to develop a program using that language. This result agrees with the intuitive notion that it takes more effort to develop a program in assembly language than to develop a program in a high-level language to solve a problem.

**Effort and Time**

The effort required to develop a program can be obtained by dividing the program volume with the level of the programming language used to develop the code. Thus, effort E = V/L, where E is the number of mental discriminations required to implement the program and also the effort required to read and understand the program. Thus, the programming effort E = V²/V\* (since L = V\*/V) varies as the square of the volume. Experience shows that E is well correlated to the effort needed for maintenance of an existing program.

The programmer’s time T = E/S, where S the speed of mental discriminations. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.

**Length Estimation**

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity. His method is summarized below.

Halstead assumed that it is quite unlikely that a program has several identical parts – in formal language terminology identical substrings – of length greater than η (η being the program vocabulary). In fact, once a piece of code occurs identically at several places, it is made into a procedure or a function. Thus, it can be assumed that any program of length N consists of N/ η unique strings of length η. Now, it is standard combinatorial result that for any given alphabet of

r

size K, there are exactly K different strings of length r.

Thus.

* **η+1**

**N/η ≤ η Or, N ≤ η**

Since operators and operands usually alternate in a program, the upper bound can be further

η1 η2

refined into N ≤ η η1 η2 . Also, N must include not only the ordered set of n elements, but it

should also include all possible subsets of that ordered sets, i.e. the power set of N strings (This particular reasoning of Halstead is not very convincing!!!).

Therefore,

|  |  |
| --- | --- |
| **N** | **η1η2** |

1. **=ηη1 η2**

Or, taking logarithm on both sides,

|  |  |  |
| --- | --- | --- |
|  | **η1** | **η2** |
| **N = log2η +log 2(η1** | | **η2 )** |
| So we get, |  |  |
| **η1** | **η2** |  |
| **N = log 2(η1** | **η2 )** |  |

(approximately, by ignoring **log2η**)

Or,

**η1** **η2**

**N = log2η1** **+ log2η2**

* **η1log2η1 + η2log2η2**

Experimental evidence gathered from the analysis of larger number of programs suggests that the computed and actual lengths match very closely. However, the results may be inaccurate when small programs when considered individually.

In conclusion, Halstead’s theory tries to provide a formal definition and quantification of such qualitative attributes as program complexity, ease of understanding, and the level of abstraction based on some low-level parameters such as the number of operands, and operators appearing in the program. Halstead’s software science provides gross estimation of properties of a large collection of software, but extends to individual cases rather inaccurately.

**Example:**

Let us consider the following C program:

**main( )**

**{**

**int a, b, c, avg;**

**scanf(“%d %d %d”, &a, &b, &c);**

**avg = (a+b+c)/3;**

**printf(“avg = %d”, avg);**

**}**

The unique operators are:

**main,(),{},int,scanf,&,“,”,“;”,=,+,/, printf**

The unique operands are:

**a, b, c, &a, &b, &c, a+b+c, avg, 3,**

**“%d %d %d”, “avg = %d”**

Therefore,

**η1= 12, η2= 11**

Estimated Length **= (12\*log12 + 11\*log11)**

* **(12\*3.58 + 11\*3.45)**
* **(43+38) = 81**

Volume **= Length\*log(23)**

* **81\*4.52**
* **366**

**COCOMO MODEL**

**Organic, Semidetached and Embedded software projects**

Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. In order to classify a product into the identified categories, Boehm not only considered the characteristics of the product but also those of the development team and development environment. Roughly speaking, these three product classes correspond to application, utility and system programs, respectively. Normally, data processing programs are considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs. System programs interact directly with the hardware and typically involve meeting timing constraints and concurrent processing.

Boehm’s [1981] definition of organic, semidetached, and embedded systems are elaborated below.

**Organic:** A development project can be considered of organic type, if the project deals withdeveloping a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.

**Semidetached:** A development project can be considered of semidetached type, if thedevelopment consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

**Embedded:** A development project is considered to be of embedded type, if the software beingdeveloped is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.

**COCOMO**

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

B **asic COCOMO Model**

The basic COCOMO model gives an approximate estimate of the project parameters. The basic

COCOMO estimation model is given by the following expressions:

**a**

**Effort = a1 х (KLOC) 2 PM**

**b**

**Tdev = b1 x (Effort) 2 Months**

Where

* KLOC is the estimated size of the software product expressed in Kilo Lines of

Code,

* a1, a2, b1, b2 are constants for each category of software products,
* Tdev is the estimated time to develop the software, expressed in months,
* Effort is the total effort required to develop the software product, expressed in person months (PMs).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in fig. 33.1). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve (as shown in fig.

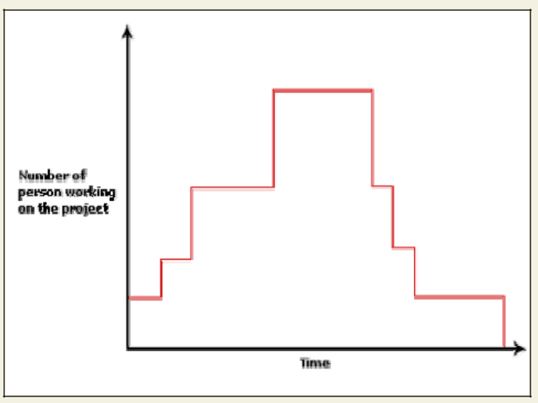


Fig. Person-month curve

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a1, a2, b1, b2 for different categories of

products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

**Estimation of development effort**

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

**1.05**

Organic : **Effort = 2.4(*KLOC*)** **PM**

**1.12**

Semi-detached : **Effort = 3.0(*KLOC*)** **PM**

**1.20**

Embedded : **Effort = 3.6(*KLOC*)** **PM**

**Estimation of development time**

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

**0.38**

Organic : **Tdev = 2.5(*Effort*)** **Months**

**0.35**

Semi-detached : **Tdev = 2.5(*Effort*)** **Months**

**0.32**

Embedded : **Tdev = 2.5(*Effort*)** **Months**

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig. 33.2 shows a plot of estimated effort versus product size. From fig. 33.2, we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

**INTERMEDIATE COCOMO MODEL**

The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, a host of other project parameters besides the product size affect the effort required to develop the product as well as the development time. Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account. The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers can be classified as being attributes of the following items:

**Product:** The characteristics of the product that are considered include the inherent complexityof the product, reliability requirements of the product, etc.

**Computer:** Characteristics of the computer that are considered include the execution speedrequired, storage space required etc.

**Personnel:** The attributes of development personnel that are considered include the experiencelevel of personnel, programming capability, analysis capability, etc.

**Development Environment:** Development environment attributes capture the developmentfacilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

**Complete COCOMO model**

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the

development team might have no previous experience of similar development, and so on. The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

* Database part
* Graphical User Interface (GUI) part
* Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

**COMPUTER AIDED SOFTWARE ENGINEERING**

**CASE tool and its scope**

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form

of automated support for software engineering. In a more restrictive sense, a CASE tool means

any tool used to automate some activity associated with software development. Many CASE

tools are available. Some of these CASE tools assist in phase related tasks such as specification,

structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project

management and configuration management.

Reasons for using CASE tools

The primary reasons for using a CASE tool are:

* To increase productivity
* To help produce better quality software at lower cost

**CASE environment**

Although individual CASE tools are useful, the true power of a tool set can be realized only when these set of tools are integrated into a common framework or environment. CASE tools are characterized by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software development artifacts. This central repository is usually a data dictionary containing the definition of all composite and elementary

data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus a CASE environment facilities the automation of the step-by-step methodologies for software development. A schematic representation of a CASE environment is shown in fig.

**CASE environment vs programming environment**

A CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast to a CASE environment, a programming environment is an integrated collection of tools to support only the coding phase of software development.

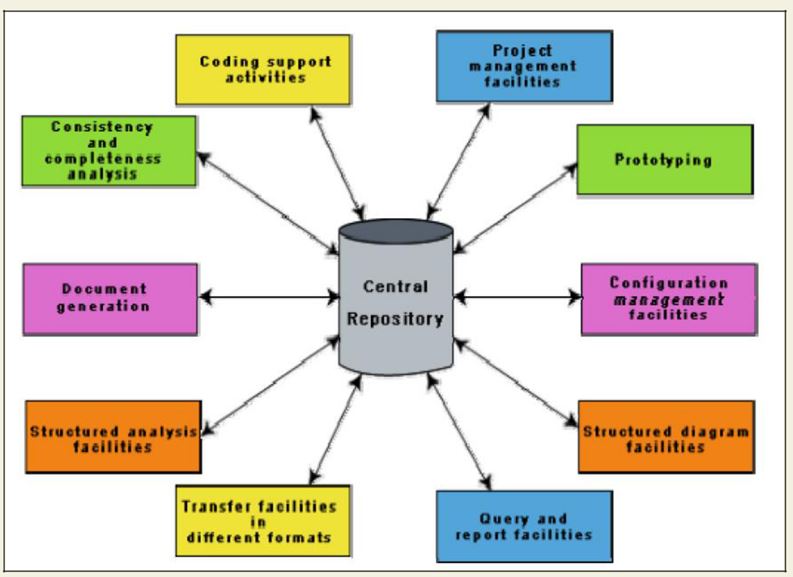


Fig. A CASE Environment

**Benefits of CASE**

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

* A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%.

Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.

* CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundancy in the stored data is reduced and therefore chances of inconsistent documentation is reduced to a great extent.
* CASE tools take out most of the drudgery in a software engineer’s work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.
* CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.
* Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.

**Requirements of a prototyping CASE tool**

Prototyping is useful to understand the requirements of complex software products, to demonstrate a concept, to market new ideas, and so on. The important features of a prototyping CASE tool are as follows:

* Define user interaction
* Define the system control flow
* Store and retrieve data required by the system
* Incorporate some processing logic

**Features of a good prototyping CASE tool**

There are several stand-alone prototyping tools. But a tool that integrates with the data dictionary can make use of the entries in the data dictionary, help in populating the data dictionary and ensure the consistency between the design data and the prototype. A good prototyping tool should support the following features:

* Since one of the main uses of a prototyping CASE tool is graphical user interface (GUI) development, prototyping CASE tool should support the user to create a GUI using a graphics editor. The user should be allowed to define all data entry forms, menus and controls.
* It should integrate with the data dictionary of a CASE environment.
* If possible, it should be able to integrate with external user defined modules written in C or some popular high level programming languages.

The user should be able to define the sequence of states through which a created prototype can run. The user should also be allowed to control the running of the prototype.

* The run time system of prototype should support mock runs of the actual system and management of the input and output data.

**Structured analysis and design with CASE tools**

Several diagramming techniques are used for structured analysis and structured design. The following supports might be available from CASE tools.

* A CASE tool should support one or more of the structured analysis and design techniques.
* It should support effortlessly drawing analysis and design diagrams.
* It should support drawing for fairly complex diagrams, preferably through a hierarchy of levels.
* The CASE tool should provide easy navigation through the different levels and through the design and analysis.
* The tool must support completeness and consistency checking across the design and analysis and through all levels of analysis hierarchy. Whenever it is possible, the system should disallow any inconsistent operation, but it may be very difficult to implement such a feature. Whenever there arises heavy computational load while consistency checking, it should be possible to temporarily disable consistency checking.

**Code generation and CASE tools**

As far as code generation is concerned, the general expectation of a CASE tool is quite low. A reasonable requirement is traceability from source file to design data. More pragmatic supports expected from a CASE tool during code generation phase are the following:

* The CASE tool should support generation of module skeletons or templates in one or more popular languages. It should be possible to include copyright message, brief description of the module, author name and the date of creation in some selectable format.
* The tool should generate records, structures, class definition automatically from the contents of the data dictionary in one or more popular languages.
* It should generate database tables for relational database management systems.
* The tool should generate code for user interface from prototype definition for X window and MS window based applications.

**Test case generation CASE tool**

The CASE tool for test case generation should have the following features:

* It should support both design and requirement testing.
* It should generate test set reports in ASCII format which can be directly imported into the test plan document.

H **ardware and environmental requirements**

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, it can be emphasized on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and this can be chosen for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients who run different modules access data dictionary through this server. The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows sharing information between users and projects. The data dictionary provides consistent view of all project entities, e.g. a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow back up/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

**Documentation Support**

The deliverable documents should be organized graphically and should be able to incorporate text and diagrams from the central repository. This helps in producing up-to-date documentation. The CASE tool should integrate with one or more of the commercially available desktop publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

**Project Management Support**

The CASE tool should support collecting, storing, and analyzing information on the software project’s progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

E **xternal Interface**

The CASE tool should allow exchange of information for reusability of design. The information which is to be exported by the CASE tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information. It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

**Reverse Engineering**

The CASE tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code. If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

**Data Dictionary Interface**

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

**Second-generation CASE tool**

An important feature of the second-generation CASE tool is the direct support of any adapted methodology. This would necessitate the function of a CASE administrator organization who can tailor the CASE tool to a particular methodology. In addition, the second-generation CASE tools have following features:

* **Intelligent diagramming support-** The fact that diagramming techniques are useful forsystem analysis and design is well established. The future CASE tools would provide help to aesthetically and automatically lay out the diagrams.
* **Integration with implementation environment-** The CASE tools should provideintegration between design and implementation.
* **Data dictionary standards-** The user should be allowed to integrate many developmenttools into one environment. It is highly unlikely that any one vendor will be able to deliver a total solution. Moreover, a preferred tool would require tuning up for a particular system. Thus the user would act as a system integrator. This is possibly only if some standard on data dictionary emerges.
* **Customization support-** The user should be allowed to define new types of objects andconnections. This facility may be used to build some special methodologies. Ideally it should be possible to specify the rules of a methodology to a rule engine for carrying out the necessary consistency checks.

A**rchitecture of a CASE environment**

The architecture of a typical modern CASE environment is shown diagrammatically in fig. 39.2. The important components of a modern CASE environment are user interface, tool set, object management system (OMS), and a repository. Characteristics of a tool set have been discussed earlier.

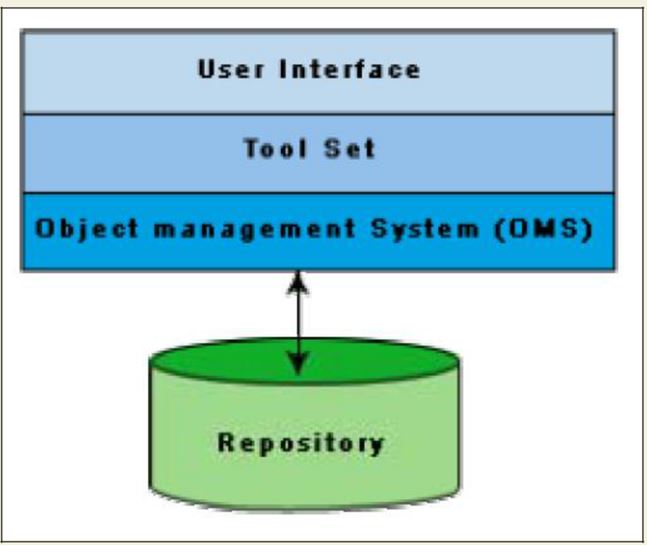


Fig. Architecture of a Modern CASE Environment

**User Interface**

The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

**Object Management System (OMS) and Repository**

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities such into the underlying storage management system (repository). The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of entities but large number of instances. By contrast, CASE tools create a large number of entity and relation types with

perhaps a few instances of each. Thus the object management system takes care of appropriately mapping into the underlying storage management system.

**SOFTWARE REUSE**

**Advantages of software reuse**

Software products are expensive. Software project managers are worried about the high cost of software development and are desperately look for ways to cut development cost. A possible way to reduce development cost is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to higher quality of the developed products since the reusable components are ensured to have high quality. Artifacts that can be reused

It is important to know about the kinds of the artifacts associated with software development that can be reused. Almost all artifacts associated with software development, including project plan and test plan can be reused. However, the prominent items that can be effectively reused are:

* Requirements specification
* Design
* Code
* Test cases
* Knowledge

**Pros and cons of knowledge reuse**

Knowledge is the most abstract development artifact that can be reused. Out of all the reuse artifacts i.e. requirements specification, design, code, test cases, reuse of knowledge occurs automatically without any conscious effort in this direction. However, two major difficulties with unplanned reuse of knowledge are that a developer experienced in one type of software product might be included in a team developing a different type of software. Also, it is difficult to remember the details of the potentially reusable development knowledge. A planned reuse of knowledge can increase the effectiveness of reuse. For this, the reusable knowledge should be systematically extracted and documented. But, it is usually very difficult to extract and document reusable knowledge.

**Easiness of reuse of mathematical functions**

The routines of mathematical libraries are being reused very successfully by almost every programmer. No one in his right mind would think of writing a routine to compute sine or cosine. Reuse of commonly used mathematical functions is easy. Several interesting aspects emerge. Cosine means the same to all. Everyone has clear ideas about what kind of argument should cosine take, the type of processing to be carried out and the results returned. Secondly,

mathematical libraries have a small interface. For example, cosine requires only one parameter.

Also, the data formats of the parameters are standardized.

**Basic issues in any reuse program**

The following are some of the basic issues that must be clearly understood for starting any reuse program.

* Component creation
* Component indexing and storing
* Component search
* Component understanding
* Component adaptation
* Repository maintenance

**Component creation-** For component creation, the reusable components have to be firstidentified. Selection of the right kind of components having potential for reuse is important. Domain analysis is a promising technique which can be used to create reusable components.

**Component indexing and storing-** Indexing requires classification of the reusablecomponents so that they can be easily searched when looking for a component for reuse. The components need to be stored in a Relational Database Management System (RDBMS) or an Object-Oriented Database System (ODBMS) for efficient access when the number of components becomes large.

**Component searching-** The programmers need to search for right components matchingtheir requirements in a database of components. To be able to search components efficiently, the programmers require a proper method to describe the components that they are looking for.

**Component understanding-** The programmers need a precise and sufficiently completeunderstanding of what the component does to be able to decide whether they can reuse the component. To facilitate understanding, the components should be well documented and should do something simple.

**Component adaptation-** Often, the components may need adaptation before they can bereused, since a selected component may not exactly fit the problem at hand. However, tinkering with the code is also not a satisfactory solution because this is very likely to be a source of bugs.

**Repository maintenance-** A component repository once is created requires continuousmaintenance. New components, as and when created have to be entered into the repository.

The faulty components have to be tracked. Further, when new applications emerge, the older applications become obsolete. In this case, the obsolete components might have to be removed from the repository.

**Domain Analysis**

The aim of domain analysis is to identify the reusable components for a problem domain.

**Reuse domain-** A reuse domain is a technically related set of application areas. A body ofinformation is considered to be a problem domain for reuse, if a deep and comprehensive relationship exists among the information items as categorized by patterns of similarity among the development components of the software product. A reuse domain is shared understanding of some community, characterized by concepts, techniques, and terminologies that show some coherence. Examples of domains are accounting software domain, banking software domain, business software domain, manufacturing automation software domain, telecommunication software domain, etc.

Just to become familiar with the vocabulary of a domain requires months of interaction with the experts. Often, one needs to be familiar with a network of related domains for successfully carrying out domain analysis. Domain analysis identifies the objects, operations, and the relationships among them. For example, consider the airline reservation system, the reusable objects can be seats, flights, airports, crew, meal orders, etc. The reusable operations can be scheduling a flight, reserving a seat, assigning crew to flights, etc. The domain analysis generalizes the application domain. A domain model transcends specific applications. The common characteristics or the similarities between systems are generalized.

During domain analysis, a specific community of software developers gets together to discuss community-wide-solutions. Analysis of the application domain is required to identify the reusable components. The actual construction of reusable components for a domain is called domain engineering.

**Evolution of a reuse domain-** The ultimate result of domain analysis is development ofproblem-oriented languages. The problem-oriented languages are also known as application generators. These application generators, once developed form application development standards. The domains slowly develop. As a domain develops, it is distinguishable the various stages it undergoes:

**Stage 1:** There is no clear and consistent set of notations. Obviously, no reusable components areavailable. All software is written from scratch.

**Stage 2:** Here, only experience from similar projects is used in a development effort. This meansthat there is only knowledge reuse.

**Stage 3:** At this stage, the domain is ripe for reuse. The set of concepts are stabilized and thenotations standardized. Standard solutions to standard problems are available. There is both knowledge and component reuse.

**Stage 4:** The domain has been fully explored. The software development for the domain can belargely automated. Programs are not written in the traditional sense any more. Programs are written using a domain specific language, which is also known as an application generator.

**REUSE APPROACH**

**Components Classification**

Components need to be properly classified in order to develop an effective indexing and storage scheme. Hardware reuse has been very successful. Hardware components are classified using a multilevel hierarchy. At the lowest level, the components are described in several forms: natural language description, logic schema, timing information, etc. The higher the level at which a component is described, the more is the ambiguity. This has motivated the Prieto-Diaz’s classification scheme.

**Prieto-Diaz’s classification scheme:** Each component is best described using a number ofdifferent characteristics or facets. For example, objects can be classified using the following:

**Searching-** The domain repository may contain thousands of reuse items. A popular searchtechnique that has proved to be very effective is one that provides a web interface to the repository. Using such a web interface, one would search an item using an approximate automated search using key words, and then from these results do a browsing using the links provided to look up related items. The approximate automated search locates products that appear to fulfill some of the specified requirements. The items located through the approximate search serve as a starting point for browsing the repository. These serve as the starting point for browsing the repository. The developer may follow links to other products until a sufficiently good match is found. Browsing is done using the keyword-to-keyword, keyword-to-product, and product-to-product links. These links help to locate additional products and compare their detailed attributes. Finding a satisfactorily item from the repository may require several locations of approximate search followed by browsing. With each iteration, the developer would get a better understanding of the available products and their differences. However, we must remember that the items to be searched may be components, designs, models, requirements, and even knowledge.

**Repository maintenance -** Repository maintenance involves entering new items, retiring thoseitems which are no more necessary, and modifying the search attributes of items to improve the effectiveness of search. The software industry is always trying to implement something that has not been quite done before. As patterns requirements emerge, new reusable components are identified, which may ultimately become more or less the standards. However, as technology advances, some components which are still reusable, do not fully address the current requirements. On the other hand, restricting reuse to highly mature components, sacrifices one of that creates potential reuse opportunity. Making a product available before it has been thoroughly assessed can be counter productive. Negative experiences tend to dissolve the trust in the entire reuse framework.

**Application generator -**The problem- oriented languages are known as application generators.Application generators translate specifications into application programs. The specification is usually written using 4GL. The specification might also in a visual form. Application generator can be applied successfully to data processing application, user interface, and compiler development.

**Advantages of application generators**

Application generators have significant advantages over simple parameterized programs. The biggest of these is that the application generators can express the variant information in an appropriate language rather than being restricted to function parameters, named constants, or tables. The other advantages include fewer errors, easier to maintain, substantially reduced development effort, and the fact that one need not bother about the implementation details.

**Shortcomings of application generators**

Application generators are handicapped when it is necessary to support some new concepts or features. Application generators are less successful with the development of applications with close interaction with hardware such as real-time systems.

**Re-use at organization level**

Achieving organization-level reuse requires adoption of the following steps:

* Assessing a product’s potential for reuse
* Refining products for greater reusability
* Entering the product in the reuse repository

**Assessing a product’s potential for reuse.** Assessment of components reuse potentialcan be obtained from an analysis of a questionnaire circulated among the developers. The questionnaire can be devised to access a component’s reusability. The programmers working in similar application domain can be used to answer the questionnaire about the product’s reusability. Depending on the answers given by the programmers, either the component be taken up for reuse as it is, it is modified and refined before it is entered into the reuse repository, or it is ignored. A sample questionnaire to assess a component’s reusability is the following.

* Is the component’s functionality required for implementation of systems in the future?
* How common is the component’s function within its domain?
* Would there be a duplication of functions within the domain if the component is taken up?
* I s the component hardware dependent?
* Is the design of the component optimized enough?
* If the component is non-reusable, then can it be decomposed to yield some reusable components?

Can we parameterize a non-reusable component so that it becomes reusable?

**Refining products for greater reusability.** For a product to be reusable, it must berelatively easy to adapt it to different contexts. Machine dependency must be abstracted out or localized using data encapsulation techniques. The following refinements may be carried out:

* **Name generalization:** The names should be general, rather than being directlyrelated to a specific application.
* **Operation generalization:** Operations should be added to make the componentmore general. Also, operations that are too specific to an application can be removed.
* **Exception generalization:** This involves checking each component to see whichexceptions it might generate. For a general component, several types of exceptions might have to be handled.
* **Handling portability problems:** Programs typically make some assumptionregarding the representation of information in the underlying machine. These assumptions are in general not true for all machines. The programs also often need to call some operating system functionality and these calls may not be same on all machines. Also, programs use some function libraries, which may not be available on all host machines. A portability solution to overcome these problems is shown in fig. 41.1. The portability solution suggests that rather than call the operating system and I/O procedures directly, abstract versions of these should be called by the application program. Also, all platform-related calls should be routed through the portability interface. One problem with this solution is the significant overhead incurred, which makes it inapplicable to many real-time systems and applications requiring very fast response.