New Scheme Based On AICTE Flexible Curricula

Information Technology, VI-Semester

Open Elective IT 604(A) Intellectual Property Rights

Course Objectives:

1. To enable Students to understand Primary forms of IPR
2. To enable Students to understand what is infringement of copyright and its consequences
3. To introduce criteria and procedure for obtaining patents
4. To enable Students to understand the registration procedures related to IPR.
5. To expose Students to contemporary issues and enforcement policies in IPR.

UNIT I Introduction

Introduction and Justifications of IPR, Nature of IP, Major forms of IP- Copyright, Patent, Trade Marks Designs, Geographic indication, layout design of Semi conductors, Plant varieties, Concept & Meaning of Intellectual Property. Major international documents relating to the protection of IP - Berne Convention, Paris Convention, TRIPS. The World Intellectual Property Organization (WIPO).

UNIT II Copyright

Meaning and historical development of copyright , Subject matter , Ownership of copyright, Term of copyright, Rights of owner, Economic Rights, Moral Rights. Assignment and licence of rights, Infringement of copyright, Exceptions of infringement, Remedies, Civil, Criminal, Administrative, Registration Procedure.

UNIT III Patents

Meaning and historical development,. Criteria for obtaining patents, Non patentable inventions, Procedure for registration, Term of patent, Rights of patentee, Compulsory licence, Revocation, Infringement of patents, Exceptions to infringement, Remedies, Patent office and Appellate Board.

UNIT IV – Trade Marks, Designs & GI

Trade Marks: Functions of marks, Procedure for registration, Rights of holder, Assignment and licensing of marks, Infringement, Trade Marks Registry and Appellate Board.

Designs: Meaning and evolution of design protection, Registration, Term of protection, Rights of holder, unregistered designs.

Geographical Indication: Meaning and evolution of GI, Difference between GI and Trade Marks, Registration, Rights, Authorised user.

UNIT V Contemporary Issues & Enforcement of IPR

IPR & sustainable development, The Impact of Internet on IPR. IPR Issues in biotechnology, E-Commerce and IPR issues, Licensing and enforcing IPR, Case studies in IPR

References:

1. P. Narayanan, Intellectual Property Law, Eastern Law House
2. . Neeraj Pandey and Khushdeep[Dharni, Intellectual Property Rights, PHI, 2014
3. N.S Gopalakrishnan and T.G. Agitha, Principles of Intellectual Property, Eastern Book Co. Lucknow, 2009.
4. Anand Padmanabhan, Enforcement of Intellectual Property, Lexis Nexis Butterworths, Nagpur, 2012.
5. Managing Intellectual Property The Strategic Imperative, Vinod V. Sople, PHI.
6. Prabuddha Ganguli, " Intellectual Property Rights" Mcgraw Hill Education, 2016.

Course Outcome:

Upon completion of this course, students will be able to:

1. Understand Primary forms of IPR
2. Assess and critique some basic theoretical justification for major forms of IP Protection
3. Compare and contrast the different forms of IPR in terms of key differences and similarities.
4. Uderstand the registration procedures related to IPR.
5. Have exposure to contemporary issues and enforcement policies in IPR.

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Information Technology, VI- semester

Open Elective IT 604(B) Software Engineering

Course Objectives:

1. To introduce software development life cycle and various software process models
2. To introduce measures and metrics for software quality, reliability and software estimation techniques
3. To develop an understanding of software analysis and design phases
4. To introduce coding standards, guidelines and various software testing techniques
5. To introduce various activities for software maintenance and quality assurance

Unit I

Introduction, Software- problem and prospects Software development process: System Development Life Cycle, Waterfall Model, Spiral Model and other models, Unified process Agile development-Agile Process- Extreme Programming- Other agile Process models.

Unit II

Measures, Metrics and Indicators, Metrics in the Process and Project Domains, Software Measurement, Metrics of Software Quality, S/W reliability, Software estimation techniques, LOC and FP estimation. Empirical models like COCOMO, project tracking and scheduling, reverse engineering.

Unit III

Software requirements and specification: feasibility study, Informal/formal specifications, pre/post conditions, algebraic specification and requirement analysis models, Specification design tools. Software design and implementation: Software design objectives and techniques, User interface design, Modularity, Functional decomposition, DFD, Data Dictionary, Object oriented design, Design patterns implementation strategies like top- down, bottom-up.

Unit IV

Coding standard and guidelines, programming style, code sharing, code review, rapid prototyping, specialization, construction, class extensions, intelligent software agents, reuse performance improvement, debugging. Software Testing Strategies: Verification and Validation, Strategic Issues, test plan, white box, black-box testing, unit and integration testing, system testing test case design and acceptance testing, maintenance activities.

Unit V

Software Maintenance: Software Supportability, Reengineering, Business Process Reengineering, Reverse Engineering, Restructuring, Forward Engineering, Economics of Reengineering, project scheduling and tracking plan, project management plan, SQA and quality planning, SCM activities

and plan, CMM, Software project management standards, Introduction to component based software engineering.

References:

1 P.S. Pressman, Software Engineering. A Practitioner's Approach, TMH.
2 Rajib Mall, Fundamental of Software Engineering, PHI.
3 Hans Van Vliet, Software Engineering, Wiley India Edition.
4 James S. Peters, Software Engineering, Wiley India Edition.
5 Pankaj Jalote, Software Engineering: A Precise Approach, Wiley India.
6 Kelkar, Software Project Management, PHI Learning

Course Outcomes:

Upon completion of this course, students will be able to-

1. Define various software application domains and remember different process model used in software development.
2. Understand various measures of software and Generate project schedule.
3. Describe functional and non-functional requirements of software and develop design models of software.
4. Investigate the reason for bugs and apply the software testing techniques in commercial environment.
5. Understand various activities to be performed for improving software quality and software maintenance.

RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

Information Technology, VI-Semester

Open Elective IT 604(C) Wireless Sensor Networks

Course Objectives:

1. To Understand the basic WSN technology and supporting protocols
2. Understand the medium access control protocols and address physical layer issues
3. Learn localization concepts for sensor networks
4. Learn energy efficiency and power control in sensor networks
5. Understand the security challenges in sensor networks.

Unit I

Overview of Wireless Sensor Networks: Network Characteristics, Network Applications, Network Design Objectives, Network Design Challenges, Technological Background : MEMS Technology , Wireless Communication Technology , Hardware and Software Platforms, Wireless Sensor Network Standards, Introduction, Network Architectures for Wireless Sensor Networks, Classifications of Wireless Sensor Networks, Protocol Stack for Wireless Sensor Networks.

Unit II

Fundamental MAC Protocols, MAC Design for Wireless Sensor Networks, MAC Protocols for Wireless Sensor Networks: Contention-Based Protocols, Contention-Free Protocols, Hybrid Protocols. Introduction, Fundamentals and Challenges, Taxonomy of Routing and Data Dissemination Protocols, Overview of Routing and Data Dissemination Protocols: Location-Aided Protocols, Layered and In-Network Processing-Based Protocols, Data-Centric Protocols, Multipath-Based Protocols, Mobility-Based Protocols, QoS Based Protocols, Heterogeneity-Based Protocols.

Unit III

Introduction, Query Processing in Wireless Sensor Networks, Data Aggregation in Wireless Sensor Networks, Node Localization: Concepts and Challenges of Node Localization Technologies, Ranging Techniques for Wireless Sensor Networks, Wireless Localization Algorithms, Wireless Sensor Node Localization.

Unit IV

Need for Energy Efficiency and Power Control in Wireless Sensor Networks, Passive Power Conservation Mechanisms: Physical-Layer Power Conservation Mechanisms, MAC Layer Power Conservation Mechanisms, Higher Layer Power Conservation Mechanisms, Active

Power Conservation Mechanisms: MAC Layer Mechanisms, Network Layer Mechanisms, Transport Layer Mechanisms.

Unit V

Fundamentals of Network Security, Challenges of Security in Wireless Sensor Networks, Security Attacks in Sensor Networks, Protocols and Mechanisms for Security, IEEE 802.15.4 and ZigBee Security .

References:

1. Wireless Sensor Networks A Networking Perspective, Jun Zheng & Abbas Jamalipour, a john Wiley & sons, Inc., publication .
2. Wireless sensor networks Technology, Protocols, and Applications , kazem sohraby, daniel minoli, taieb znati , a john Wiley & sons, Inc., publication .
3. Fundamentals of wireless sensor networks theory and practice, Waltenegus Dargie, Christian Poellabauer, A John Wiley and Sons, Ltd., Publication.

Course Outcomes:

Upon completion of this course, students will be able to-

1. Have knowledge of some existing applications of wireless sensor actuator networks
2. Learn the various hardware, software platforms that exist for sensor networks
3. Have knowledge of the various protocols for sensor networks
4. Analyze modeling and simulation of sensor networks
5. Understand what research problems sensor networks pose in disciplines such as signal processing, wireless communications and even control systems

CS403 Software Engineering
UNIT-I

SOFTWARE PRODUCT AND PROCESS CHARACTERISTICS:

Software: -

Software is nothing but collection of computer programs and related documents that are planned to provide desired features, functionalities and better performance.

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

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

Characteristics of software: -

1. Software is developed or engineered; it is not manufactured in the classical sense:
   - Although some similarities exist between software development and hardware manufacturing, but few activities are fundamentally different.
   - In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems than software.
2. Software doesn't "wear out."
   - Hardware components suffer from the growing effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.
   - Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the "idealized curve".
   - When a hardware component wears out, it is replaced by a spare part.
   - There are no software spare parts.
   - Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.
   - However, the implication is clear—software doesn't wear out. But it does deteriorate.

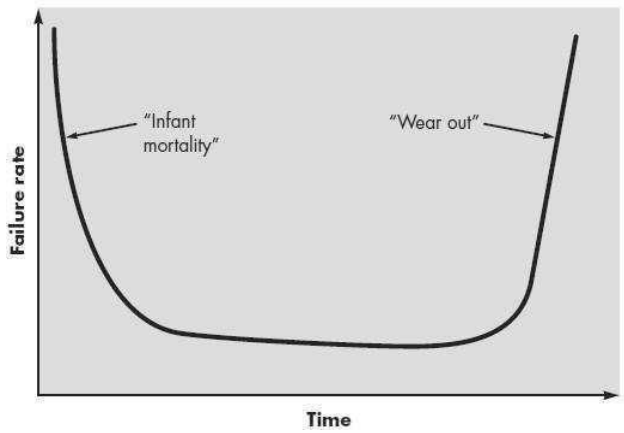


Figure 1.1 Hardware Failure Curve

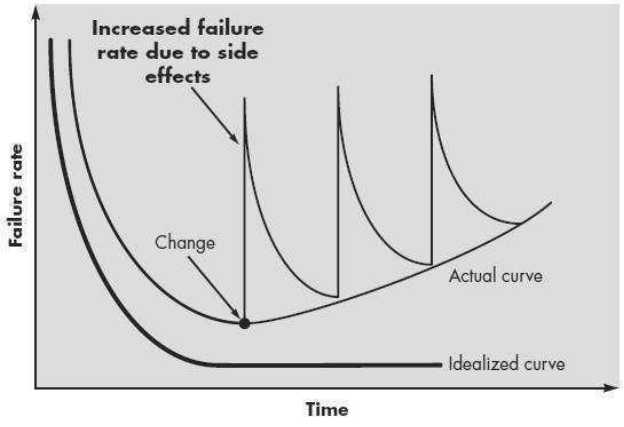


Figure 1.2 Software Failure Cure

3. Although the industry is moving toward component-based construction, most software continues to be custom built.

- A software component should be designed and implemented so that it can be reused programs.
- Modern reusable components encapsulate both data and the processing that is applied to the data, enabling the software engineer to create new application form reuable parts.
- In the hardware world, component reuse is a natural part of the engineering process

Good Software are-

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 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 Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

Software product classified in 2 classes:

2. **Generic software**: Developed to solution whose requirements are very common fairly stable and well understood by software engineer.
3. **Custom software**: Developed for a single customer according to their specification.

A Layered Technology:

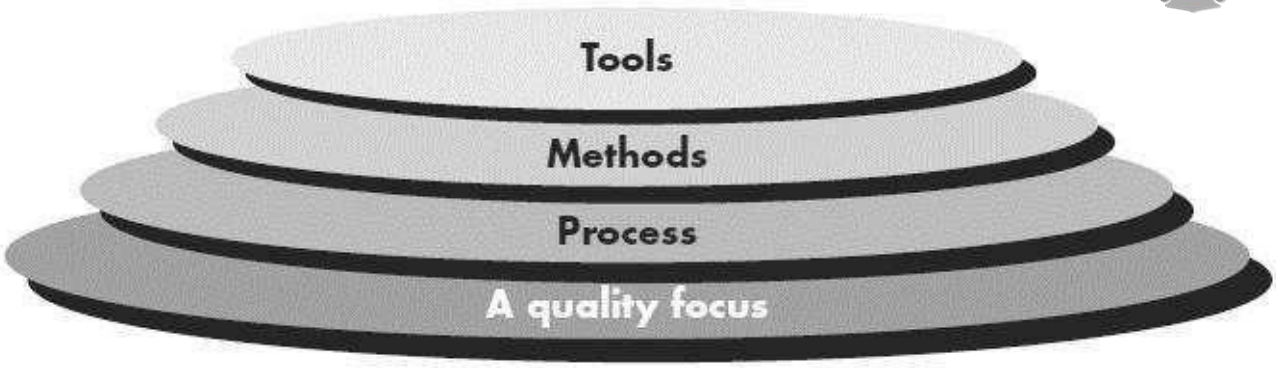


Figure 1.3 Layed Architecture

A quality Focus:

- Every organization is rest on its commitment to quality.
- Total quality management, Six Sigma, or similar continuous improvement culture and it is this culture ultimately leads to development of increasingly more effective approaches to software engineering.
- The foundation that supports software engineering is a quality focus.

Process:

- The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.
- Process defines a framework that must be established for effective delivery of software engineering technology.
- The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

Methods:

- Software engineering methods provide the technical aspects for building software.
- Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.
- Software engineering methods rely on the set of modeling activities and other descriptive techniques.

Tools:

- Software engineering tools provide automated or semi automated support for the process and the method.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called CASE (Computer- aided softeware engineering), is established.

SOFTWARE PROCESS MODEL:

Software process can be defining as the structured set of activates that are required to develop the software system.

To solve actual problems in an industry setting, a software engineer or a team of engineers must incorporate a development strategy that encompasses the process, methods, and tools layers. This strategy is often referred to as a process model or a software engineering paradigm.

A process model for software engineering is chosen based on the nature of the project and application, the methods and tools to be used, and the controls and deliverables that are required.

Goal of Software Process Models: -

The goal of a software process model is to provide guidance for systematically coordinating and controlling the tasks that must be performed in order to achieve the end product and their project objectives. A process model defines the following:

- A set of tasks that need to be performed.

- The inputs to and output from each task.
- The preconditions and post-conditions for each task.
- The sequence and flow of these tasks.

Characteristics of Software Process: -

Software is often the single largest cost item in a computer-based application. Though software is a product, it is different from other physical products.

i. Software costs are concentrated in engineering (analysis and design) and not in production.
ii. Cost of software is not dependent on volume of production.
iii. Software does not wear out (in the physical sense).
iv. Software has no replacement (spare) parts.
v. Software maintenance is a difficult problem and is very different from hardware (physical product) maintenance.
vi. Most software is custom-built.
vii. Many legal issues are involved (e.g. inter-actual property rights, liability).

Software Product: -

A software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer's individual style of development, but a software product must be developed using the accepted software engineering principles.

Various Operational Characteristics of software are:

- **Correctness:** The software which we are making should meet all the specifications stated by the customer.
- **Usability/Learn-ability:** The amount of efforts or time required to learn how to use the software should be less. This makes the software user-friendly even for IT-illiterate people.
- **Integrity:** Just like medicines have side-effects, in the same way software may have aside-effect i.e. it may affect the working of another application. But quality software should not have side effects.
- **Reliability:** The software product should not have any defects. Not only this, it shouldn't fail while execution.
- **Efficiency:** This characteristic relates to the way software uses the available resources. The software should make effective use of the storage space and execute command as per desired timing requirements.
- **Security:** With the increase in security threats nowadays, this factor is gaining importance. The software shouldn't have ill effects on data / hardware. Proper measures should be taken to keep data secure from external threats.
- **Safety:** The software should not be hazardous to the environment/life.

Difference between software process and software product: -

Table 1 Difference between software process and software product

Software Process	Software Product
Processes are developed by individual user and it is used for personal use.	It is developed by multiple users and it is used by large number of people or customers.
Process may be small in size and possessing limited functionality.	It consists of multiple program codes; relate documents such as SRS, designing documents, user manuals, test cases.
Process is generally developed by process engineers.	Process is generally developed by process engineers. Therefore systematic approach of developing software product must be applied.
Software product relies on software process for its stability quality and control Only one person uses the process, hence lack of user interface	It is important than software product. Multiuser no lack of user interface.

Software Development Life Cycle/Process model/ Software Development Life Cycle: -

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product. it is a team of engineers must incorporate a development strategy that encompasses the process, method and tools layers. Each phase has various activities to develop the software product. It also specifies the order in which each phase must be executed.

A software life cycle model is either a descriptive or prescriptive characterization of how software is or should be developed. A descriptive model describes the history of how a particular software system was developed.

Definition: Software Development Life Cycle (SDLC) is a process used by software industry to design, develop and test high quality software. The SDLC aims to produce high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

SDLC is the acronym of Software Development Life Cycle. It is also called as Software development process. The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process.

LINEAR SEQUENTIAL MODEL:

A few of software development paradigms or process models are defined as follows:

Waterfall model or linear sequential model or classic life cycle model: -

Sometimes called the classic life cycle or the waterfall model, the linear sequential model suggests a systematic, sequential approach to software development that begins at the system level and progresses through analysis, design, coding, testing, and maintenance.

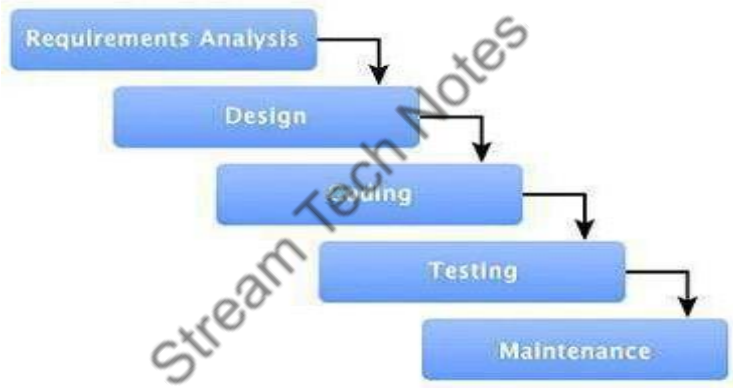


Figure 1.4 Waterfall model

Software requirements analysis: The requirements gathering process is focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

Design: Software design is actually a multi-step process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins. Like requirements, the design is documented and becomes part of the software configuration.

Coding : The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing: Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Maintenance: Software will undoubtedly undergo change after it is delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required

because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements. Software support/maintenance reapplies each of the preceding phases to an existing program rather than a new one.

Advantages of waterfall model: -

- This model is simple and easy to understand and use.
- Waterfall model works well for smaller projects where requirements are very well understood.
- Each phase proceeds sequentially.
- Documentation is produced at every stage of the software's development. This makes understanding the product designing procedure, simpler.
- After every major stage of software coding, testing is done to check the correct running of the code. help us to control schedules and budgets.

Disadvantages of waterfall model: -

- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.
- High amounts of risk and uncertainty.
- Customer can see working model of the project only at the end. after reviewing of the working model if the customer gets dissatisfied then it causes serious problem.
- You cannot go back a step if the design phase has gone wrong, things can get very complicated in the implementation phase.

PROTOTYPING MODEL:

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.

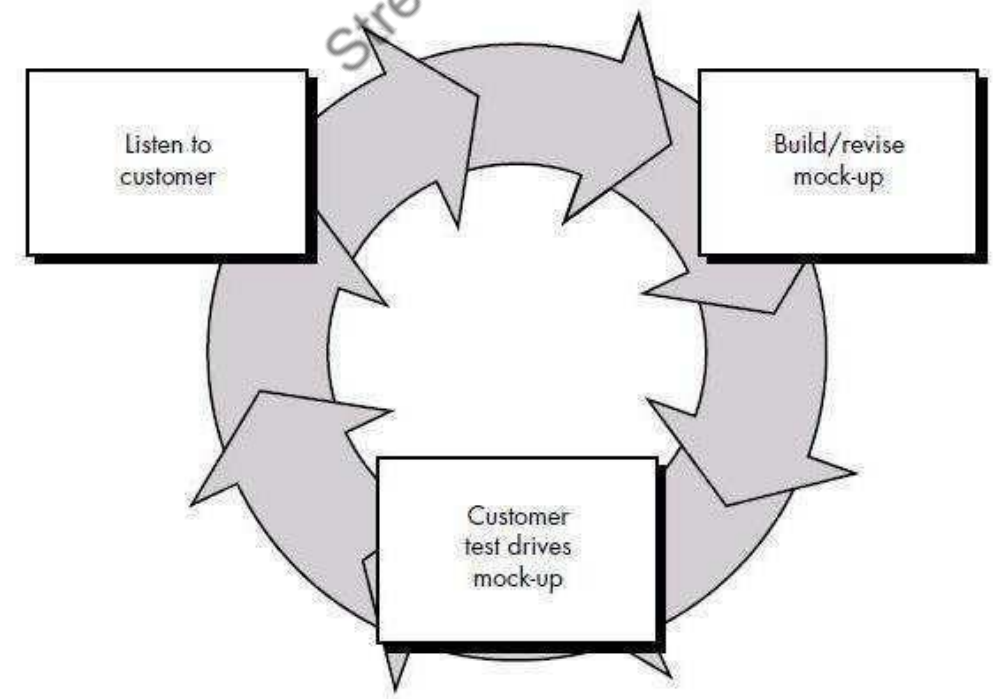


Figure 1.5 Prototype Model

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

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

RAPID APPLICATION MODEL:

Rapid application development (RAD) is a software development methodology that uses minimal planning in favor of rapid prototyping. A prototype is a working model that is functionally equivalent to a component of the product. In RAD model, the functional modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery.

Since there is no detailed pre-planning, it makes it easier to incorporate the changes within the development process. RAD projects follow iterative and incremental model and have small teams comprising of developers, domain experts, customer representatives and other IT resources working progressively on their component or prototype. The most important aspect for this model to be successful is to make sure that the prototypes developed are reusable.

Rapid application development (RAD) is an incremental software development process model that emphasizes an extremely short development cycle. The 'AD model is a "high-speed" adaptation of the linear sequential model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within very short time periods (e.g., 60 to 90 days). Used primarily for information systems applications, the RAD approach encompasses the following phases:

Business modeling: The information flow among business functions is modeled in a way that answers the following questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?

Data modeling: The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics (called *attributes*) of each object are identified and the relationships between these objects defined.

Process modeling: The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation: RAD assumes the use of fourth generation techniques. Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components (when possible) or create reusable components (when necessary). In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover: Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all

interfaces must be fully exercised.

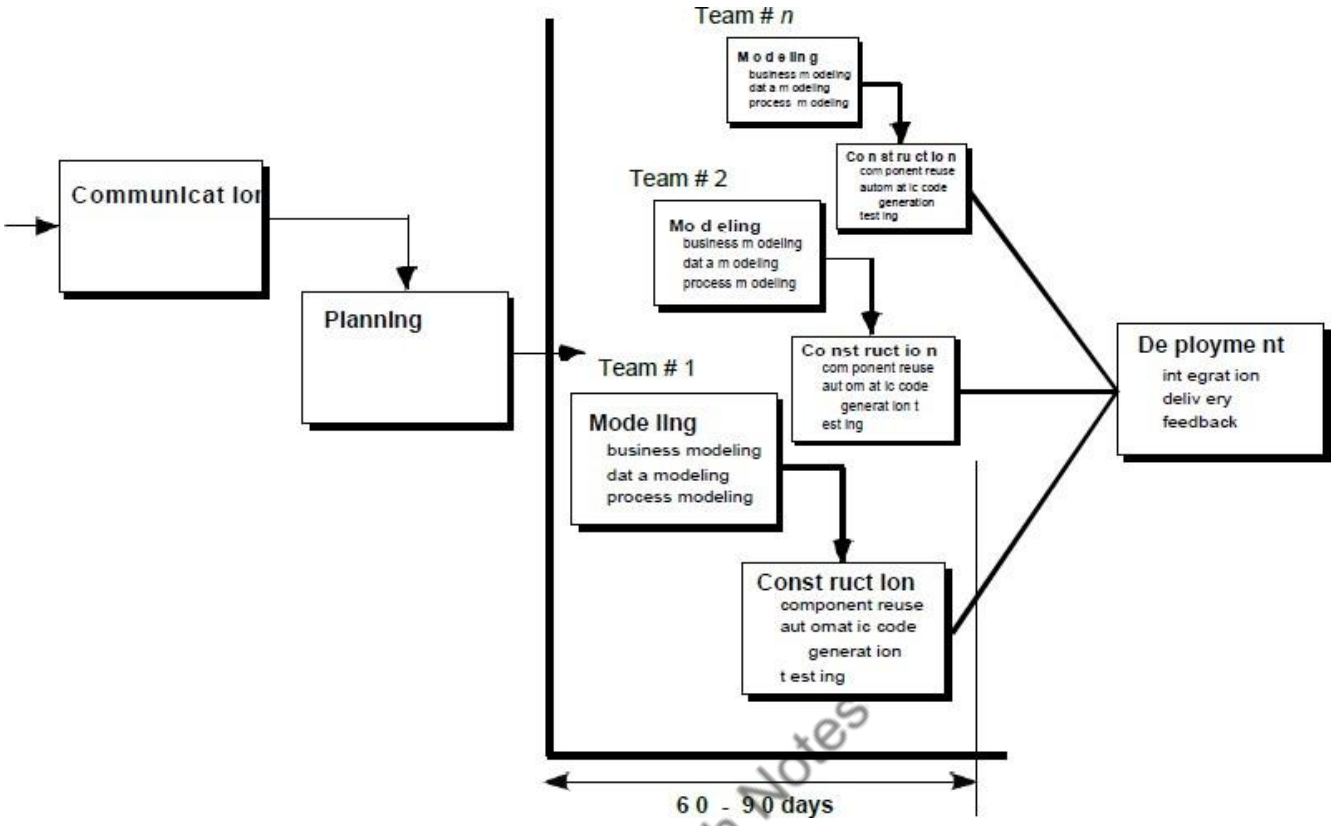


Figure 1.6.1:Rapid Application Model

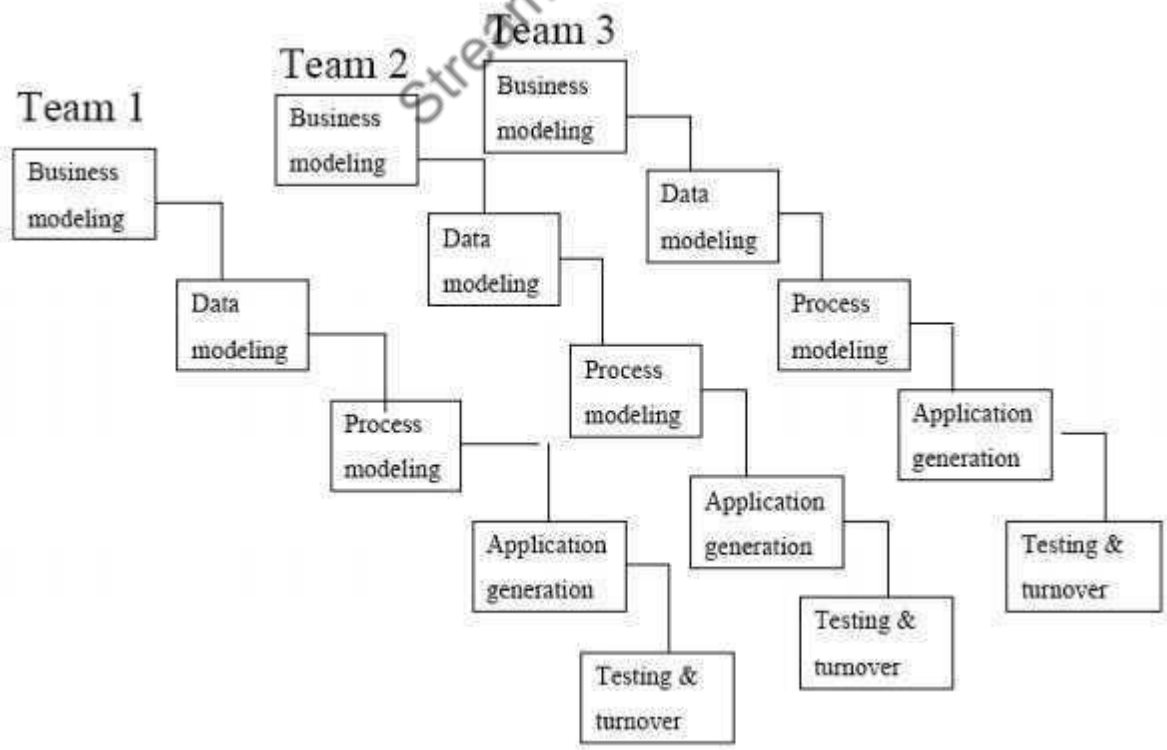


Figure 1.6.2:Rapid Application Model Teamwise

Advantages of the RAD model:

- Reduced development time.
- Increases re usability of components
- Quick initial reviews occur
- Encourages customer feedback
- Integration from very beginning solves a lot of integration issues.

Disadvantages of RAD model:

- Depends on strong team and individual performances for identifying business requirements.
- Only system that can be modularized can be built using RAD
- Requires highly skilled developers/designers.
- High dependency on modeling skills
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.

When to use RAD model:

- RAD should be used when there is a need to create a system that can be modularized in 2-3 months of time.
- It should be used if there's high availability of designers for modeling and the budget is high enough to afford their cost along with the cost of automated code generating tools.
- RAD SDLC model should be chosen only if resources with high business knowledge are available and there is a need to produce the system in a short span of time (2-3 months).

EVOLUTIONARY PROCESS MODEL:

Evolutionary Software Process Model Evolutionary software models are iterative. They are characterized in manner that enables the software engineers to develop increasingly more complete version of software. In programming "iteration" means sequential access to objects. It is typically a cycle. Software engineers can follow this process model that has been clearly designed to put up a product that regularly complete over time.

Iterative Model design:

Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

Following is the pictorial representation of Iterative and Incremental model:

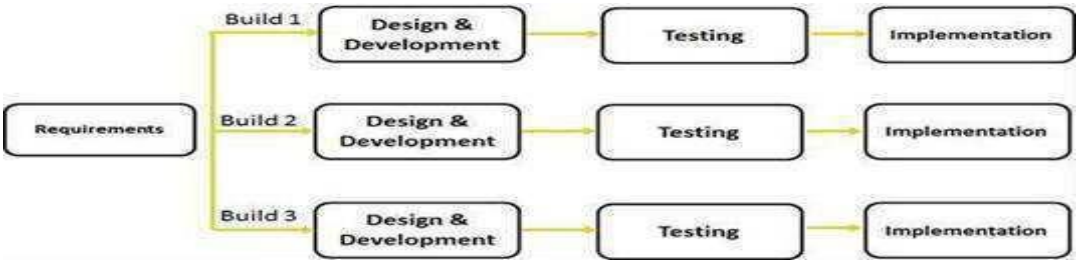


Figure 1.7 Iterative Model

Iterative Model Application:

Like other SDLC models, Iterative and incremental development has some specific applications in the software industry. This model is most often used in the following scenarios:

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.

- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill set are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

Evolutionary Process Model is of 2 types

- Incremental model and
- Spiral model

INCREMENTAL MODEL:

- The incremental model combines the elements of waterfall model and they are applied in an iterative fashion.
- The first increment in this model is generally a core product.
- Each increment builds the product and submits it to the customer for any suggested modifications.
- The next increment implements on the customer's suggestions and add additional requirements in the previous increment.
- This process is repeated until the product is finished.

For example, the word-processing software is developed using the incremental model.

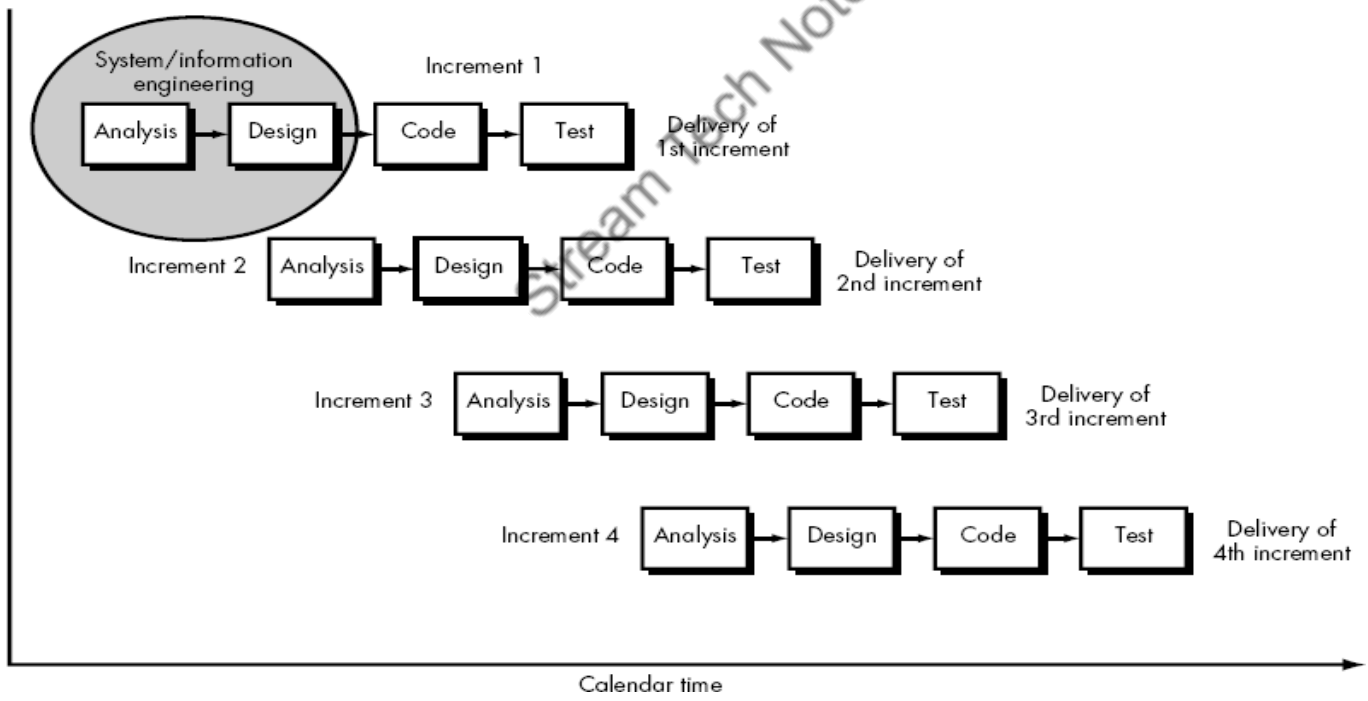


Figure 1.8 Incremental Model

Advantages of Incremental model: -

- Generates working software quickly and early during the software life cycle.
- This model is more flexible – less costly to change scope and requirements.
- It is easier to test and debug during a smaller iteration.
- In this model customer can respond to each built.
- Lowers initial delivery cost.
- Easier to manage risk because risky pieces are identified and handled during it'd iteration.
- There is low risk for overall project failure.

- Customer does not have to wait until the entire system is delivered.

Disadvantages of Incremental model: -

- Needs good planning and design at the management a technical level.
- Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
- Total cost is higher than waterfall.
- Time foundation create problem to complete the project.

When to use the Incremental model:

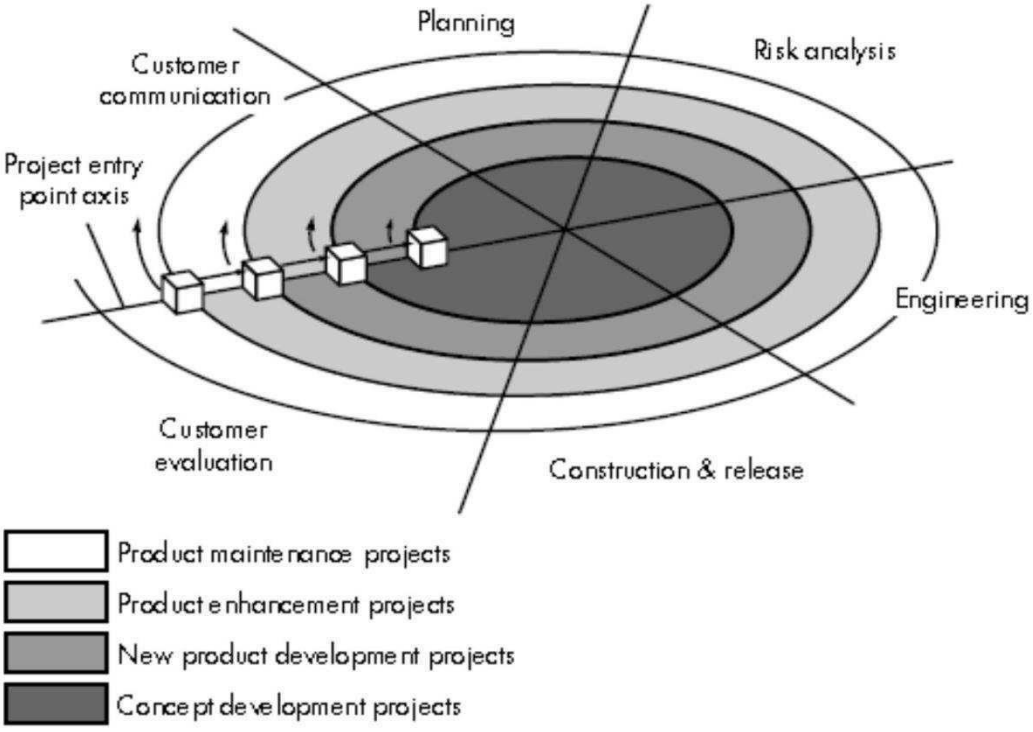
- This model can be used when the requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some details can evolve with time.
- There is a need to get a product to the market early.
- A new technology is being used
- Resources with needed skill set are not available
- There are some high-risk features and goals.

SPIRAL MODEL :

The spiral model, is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called task regions. Project entry point axis is defined this axis represents starting point for different types of project. Every framework activities represent one section of the spiral path. As the development process starts, the software team performs activities that are indirect by a path around the spiral model in a clockwise direction. It begins at the center of spiral model. Typically, there are between three and six task regions. In blow Figure depicts a spiral model that contains six task regions:

- **Customer communication**—tasks required to establish effective communication between developer and customer.
- **Planning**—tasks required to define resources, time lines, and other project related information.
- **Risk analysis**—tasks required to assess both technical and management risks.
- **Engineering**—tasks required to build one or more representations of the application.
- **Construction and release**—tasks required to construct, test, install, and provide user support(e.g., documentation and training).
- **Customer evaluation**—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation



stage.

Figure 1.9 Spiral Model

Advantages of Spiral model: -

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages of Spiral model: -

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

When to use Spiral model:

- When costs and risk evaluation is important.
- For medium to high-risk projects.
- Long-term project commitment unwise because of potential changes to economic priorities.
- Users are unsure of their needs.
- Requirements are complex.
- New product line.
- Significant changes are expected (research and exploration).

COMPONENT ASSEMBLY MODEL:

Component Assembly Model is just like the Prototype model, in which first a prototype is created according to the requirements of the customer and sent to the user for evaluation to get the feedback for the modifications to be made and the same procedure is repeated until the software will cater the need of businesses and consumers is realized. Thus it is also an iterative development model.

This model work in following manner:

1. Identify all required candidate component i.e. classes with the help of application data and algorithm.

2. If these candidate components are used in previous software project then thy must be present in library.
3. Such preexisting component can be extracted from the library and used for further development.
4. But if required component is not presented in the library then build or create the component as per requirement.
5. Place the newly created component in library. This makes one iteration of the system.
6. Repeat step 1 to 5 for creating 'n' iteration. Where 'n' denotes the no of iterations required to develop complete application.

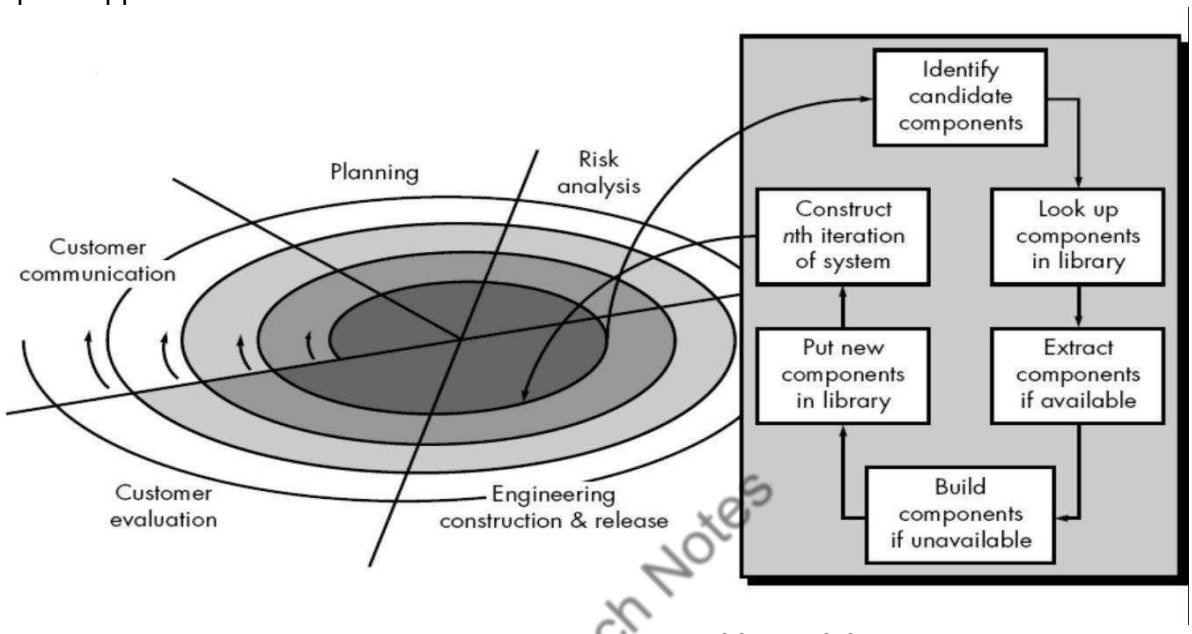


Figure 1.10: Component Assembly Model

Component Assembly Model Characteristics:

- Use of object-oriented technology.
- Components – classes that encapsulate both data and algorithms.
- Components developed to be reusable.
- Paradigm similar to spiral model, but engineering activity involves components.
- System produced by assembling the correct components.

RATIONAL UNIFIED PROCESS (RUP):

Rational Unified Process (RUP) is an object-oriented and Web-enabled program development methodology. RUP is a software application development technique with many tools to assist in coding the final product and tasks related to this goal. RUP is an object-oriented approach used to ensure effective project management and high-quality software production. It divides the development process into four distinct phases that each involves business modeling, analysis and design, implementation, testing, and deployment. The four phases are:

1. **Inception** - The idea for the project is stated. The development team determines if the project is worth pursuing and what resources will be needed.
2. **Elaboration** - The project's architecture and required resources are further evaluated. Developers consider possible applications of the software and costs associated with the development.
3. **Construction** - The project is developed and completed. The software is designed, written, and tested.
4. **Transition** - The software is released to the public. Final adjustments or updates are made based on feedback from end users.

The RUP development methodology provides a structured way for companies to envision create software programs. Since it provides a specific plan for each step of the development process, it helps prevent resources from being wasted and reduces unexpected development costs.

Advantages of RUP Software Development: -

- This is a complete methodology in itself with an emphasis on accurate documentation
- It is pro-actively able to resolve the project risks associated with the client's evolving requirements requiring careful change request management
- Less time is required for integration as the process of integration goes on throughout the software development life cycle.
- The development time required is less due to reuse of components.
- There is online training and tutorial available for this process.

Disadvantages of RUP Software Development: -

- The team members need to be expert in their field to develop a software under this methodology.
- The development process is too complex and disorganized.
- On cutting edge projects which utilize new technology, the reuse of components will not be possible. Hence the time saving one could have made will be impossible to fulfill.
- Integration throughout the process of software development, in theory sounds a good thing. But on particularly big projects with multiple development streams it will only add to the confusion and cause more issues during the stages of testing.

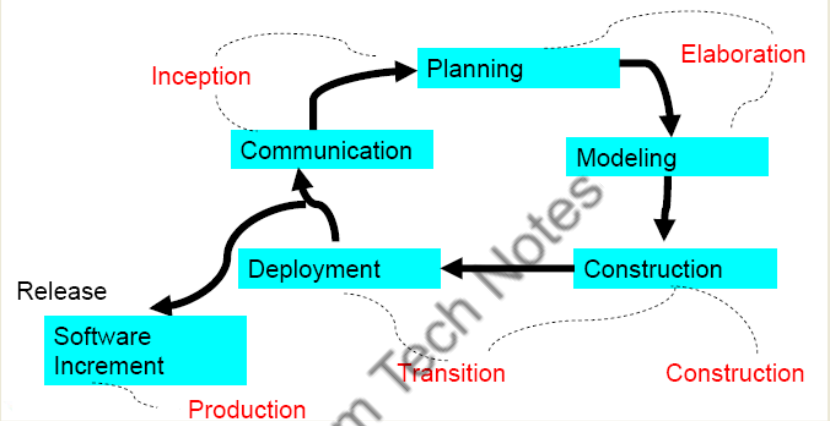


Figure 1.11 Rational Unified Process (RUP)

AGILE DEVELOPMENT MODEL

Agile Process:-The word 'agile' means able to think quickly and clearly. In business, 'agile' is used for describing ways of planning and doing work wherein it is understood that making changes as needed is an important part of the job.

Agile development model is also a type of Incremental model. Software is developed in incremental, rapid cycles. This results in small incremental releases with each release building on previous functionality. Each release is thoroughly tested to ensure software quality is maintained. It is used for time critical applications. Extreme Programming (XP) is currently one of the most well known agile development life cycle model.

Advantages of Agile model:

- Customer satisfaction by rapid, continuous delivery of useful software.
- People and interactions are emphasized rather than process and tools. Customers, developers and testers constantly interact with each other.
- Working software is delivered frequently (weeks rather than months).
- Face-to-face conversation is the best form of communication.
- Close, daily cooperation between business people and developers.
- Continuous attention to technical excellence and good design.
- Regular adaptation to changing circumstances.
- Even late changes in requirements are welcomed

Disadvantages of Agile model:

- In case of some software deliverables, especially the large ones, it is difficult to assess the effort required at the beginning of the software development life cycle.

- There is lack of emphasis on necessary designing and documentation.
- The project can easily get taken off track if the customer representative is not clear what final outcome that they want.
- Only senior programmers are capable of taking the kind of decisions required during the development process. Hence it has no place for new programmers, unless combined with experienced resources.

Extreme Programming

Extreme Programming (XP) is an agile software development framework that aims to produce higher quality software, and higher quality of life for the development team. XP is the most specific of the agile frameworks regarding appropriate engineering practices for software development.

Extreme Programming is based on the following values-

- Communication
- Simplicity
- Feedback
- Courage
- Respect

Extreme Programming takes the effective principles and practices to extreme levels.

- Code reviews are effective as the code is reviewed all the time.
- Testing is effective as there is continuous regression and testing.
- Design is effective as everybody needs to do refactoring daily.
- Integration testing is important as integrate and test several times a day.
- Short iterations are effective as the planning game for release planning and iteration planning.

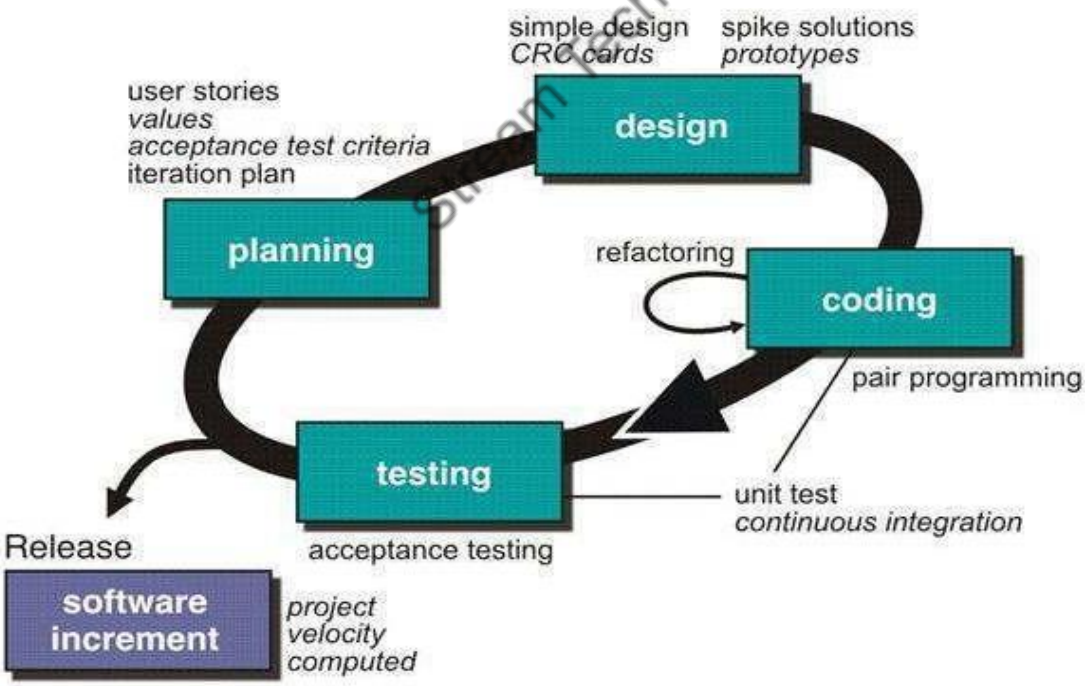


Figure 1.12 Extrem Programming

CAPABILITY MATURITY MODEL (CMM):

The Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization's current state of process maturity, the

SEI uses an assessment that results in a five-point grading scheme. The grading scheme determines compliance

with a capability maturity model (CMM) that defines key activities required at different levels of process maturity. The SEI approach provides a measure of the global effectiveness of a company's software engineering practices and establishes five process maturity levels that are defined in the following manner:

Level 1: Initial. The software process is characterized as ad hoc and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Level 2: Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Level 3: Defined. The software process for both management and engineering activities is documented, standardized, and integrated into an organization wide software process. All projects use a documented and approved version of the organization's process for developing and supporting software. This level includes all characteristics defined for level 2.

Level 4: Managed. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures. This level includes all characteristics defined for level 3.

Level 5: Optimizing. Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies. This level includes all characteristics defined for level 4.

The five levels defined by the SEI were derived as a consequence of evaluating responses to the SEI assessment questionnaire that is based on the CMM. The results of the questionnaire are distilled to a single numerical grade that provides an indication of an organization's process maturity.

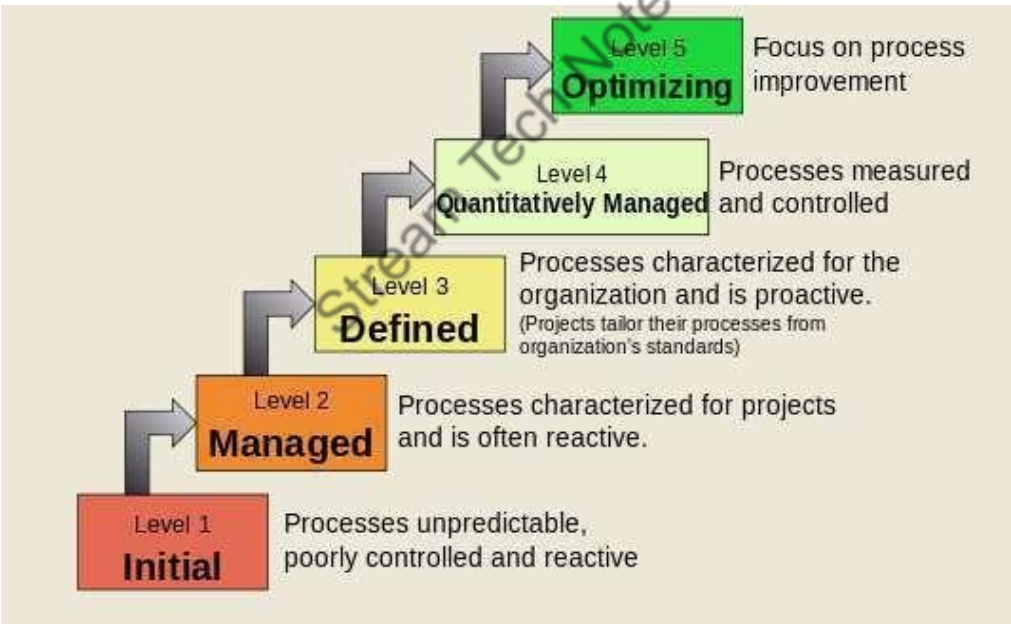


Figure 1.13 Capability Maturity Model

Process maturity level 2: -

- Software configuration management
- Software quality assurance
- Software subcontract management
- Software project tracking and oversight
- Software project planning
- Requirements management

Process maturity level 3: -

- Peer reviews
- Intergroup coordination

- Software product engineering
- Integrated software management
- Training program
- Organization process definition
- Organization process focus

Process maturity level 4: -

- Software quality management
- Quantitative process management

Process maturity level 5: -

- Process change management
- Technology change management
- Defect prevention

SOFTWARE PROCESS CUSTOMIZATION:

In software industry, most of the projects are customized software product 3 major factors that are involved in software process customization and those are:

- PEOPLE
- PRODUCT
- PROCESS

People: -

The primary element of any project is the people. People gather requirements, people interview users (people), people design software, and people write software for people. No people -- no software. I'll leave the discussion of people to the other articles in this special issue, except for one comment. The best thing that can happen to any software project is to have people who know what they are doing and have the courage and self-discipline to do it. Knowledgeable people do what is right and avoid what is wrong. Courageous people tell the truth when others want to hear something else. Disciplined people work through projects and don't cut corners. Find people who know the product and can work in the process.

Process: -

Process is how we go from the beginning to the end of a project. All projects use a process. Many project managers, however, do not choose a process based on the people and product at hand. They simply use the same process they've always used or misused. Let's focus on two points regarding process: (1) process improvement and (2) using the right process for the people and product at hand.

Product: -

The product is the result of a project. The desired product satisfies the customers and keeps them coming back for more. Sometimes, however, the actual product is something less. The product pays the bills and ultimately allows people to work together in a process and build software. Always keep the product in focus.

PRODUCT AND PROCESS METRICS:

Software process metrics measure the software development process and environment. Example productivity, effort estimates, efficiency and failure rate.

Software Product metrics measure the software product.

Example: - size, reliability, complexity and functionality.

Process Metrics and Software Process Improvement: -

The only rational way to improve any process is

- To measure specific attributes of the process
- Develop a set of meaningful metrics based on these attributes
- Use the metrics to provide indicators that will lead to a strategy for improvement

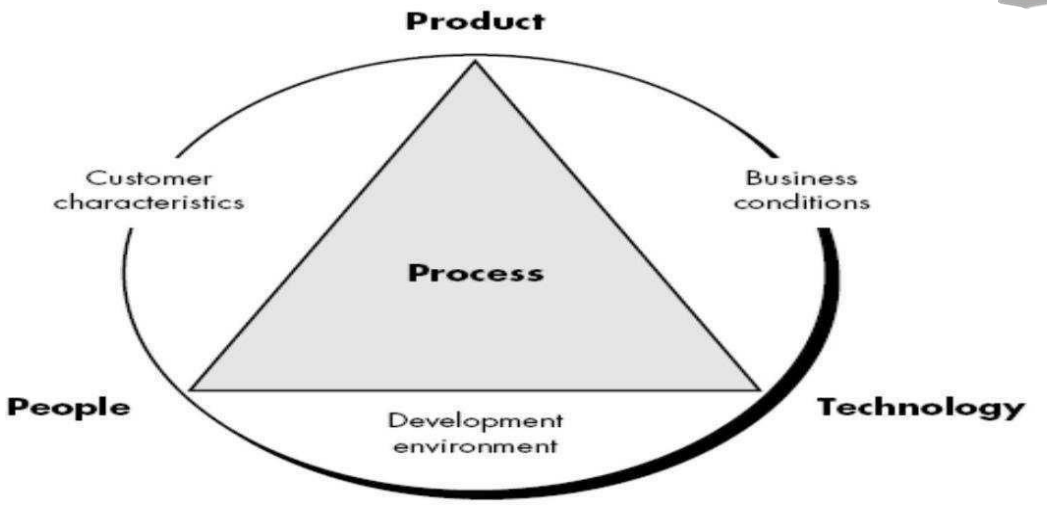


Figure 1.14 Product and Process Metrics

Processes it's at the center of a triangle connecting three factors that have profound influence on software quality and organizational performance

- The skill and motivation of **people** has most influential factor in quality and performance.
- The complexity of the **product** has impact on quality and team performance.
- The technology (the software engineering methods) the process triangle exists within a circle of environmental conditions that include the development environment, business conditions, customer characteristics.

UNIT-II

REQUIREMENT ELICITATION ANALYSIS & SPECIFICATION

Requirement: -

The process to gather the software requirements from client, analyze and document them is known as requirement engineering.

The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Types of Requirements: -

- **User Requirements**: It is a collection of statement in natural language and description of the services the system provides and its operational limitation. It is written for customer.
- **System Requirement**: It is a structured document that gives the detailed description of the system services. It is written as a contract between client and contractor.

Software Requirement Specification: -

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.
- Conditional and mathematical notations for DFDs etc.

Software Requirements: -

We should try to understand what sort of requirements may arise in the requirement elicitation phase and what kinds of requirements are expected from the software system.

Broadly software requirements should be categorized in two categories:

1. Functional Requirement
2. Non Functional Requirement

FUNCTIONAL REQUIREMENTS:

It should describe all requirement functionality or system services. The customer should provide statement of service. it should be clear how the system should be reacting to particular input and how a particular system should behave in particular situation. Functional requirement is heavily depending upon he type of software expected users and the type of system where the software is used. It describes system services in detail.

NON-FUNCTIONAL REQUIREMENTS:

Requirements, which are not related to functional aspect of software, fall into this category. They are implicit or expected characteristics of software; which users make assumption of. Non -functional are more critical than functional requirement if the non-functional requirement do not meet then the complete system is of no use.

Some typical **non-functional requirements** are:

- **Product requirement**-

Specify how a livered product should behave in particular way.

- Eg: efficiency, Usability, Reliability, portability

- **Organizational requirement**- The requirements which are unwelcome effect of organizational policies and procedures come under this category.
- Eg: Delivery, implementation, standard

- **External requirement**-
  These requirements arise due to the factors that are external of the system and its developed process.
- Eg: Interoperability, ethical, safety.

REQUIREMENT SOURCES AND ELICITATION TECHNIQUES:

Requirements Sources:-

In a typical system, there will be many sources of requirements and it is essential that all potential sources are identified and evaluated for their impact on the system. This subtopic is designed to promote awareness of different requirements sources and frameworks for managing them.

The main points covered are:

- **Goals: -**The term 'Goal' (sometimes called 'business concern' or 'critical success factor') refers to the overall, high-level objectives of the system. Goals provide the motivation for a. Requirements engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- **Domain knowledge**: - The requirements engineer needs to acquire or to have available knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements and sometimes to act as a 'user' champion.
- **System stakeholders: -**Many systems have proven unsatisfactory because they have stressed the requirements for one group of stakeholders at the expense of others. Hence, systems are delivered that are hard to use or which subvert the cultural or political structures of the customer organization. The requirements engineer needs to identify represent and manage the 'viewpoints' of many different types of stakeholder.
- **The operational environment: -**Requirements will be derived from the environment in which the software will execute. These may be, for example, timing constraints in a real-time system or interoperability constraints in an office environment. These must be actively sought because they can greatly affect system feasibility, cost, and restrict design choices.
- **The organizational environment: -**Many systems are required to support a business process and this may be conditioned by the structure, culture and internal politics of the organization. The requirements engineer needs to be sensitive to these since, in general, new software systems should not force unplanned change to the business process.

Elicitation techniques: -

When the requirements sources have been identified the requirements, engineer can start eliciting requirements from them. It also means requirement discovery. This subtopic concentrates on techniques for getting human stakeholders to articulate their requirements. This is a very difficult area and the requirements engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity and that even if cooperative and articulate stakeholders are available, the requirements engineer has to work hard to elicit the right information. A number of techniques will be covered, but the principal ones are:

- **Interviews:-**Interviews are a 'traditional' means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.

- **Scenarios**: - Scenarios are valuable for providing context to the elicitation of users' requirements. They allow the requirements engineer to provide a framework for questions about users' tasks by permitting 'what if?' and 'how is this done?' questions to be asked. (Conceptual modeling) because recent modeling notations have attempted to integrate scenario notations with object-oriented analysis techniques.
- **Prototypes*:* -**Prototypes are a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing a context within which users better understand what information they need to provide. There is a wide range of prototyping techniques, which range from paper mock-ups of screen designs to beta-test versions of software products. There is a strong overlap with the use of prototypes for requirements validation.
- **Facilitated meetings*:* -**The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight to their requirements than by working individually. They can brainstorm and refine ideas that may be difficult to surface using (e.g.) interviews.
- **Observation*:* -**The importance of systems' context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. The requirements engineer learns about users' tasks by immersing themselves in the environment and observing how users interact with their systems and each other. These techniques are relatively new and expensive but are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

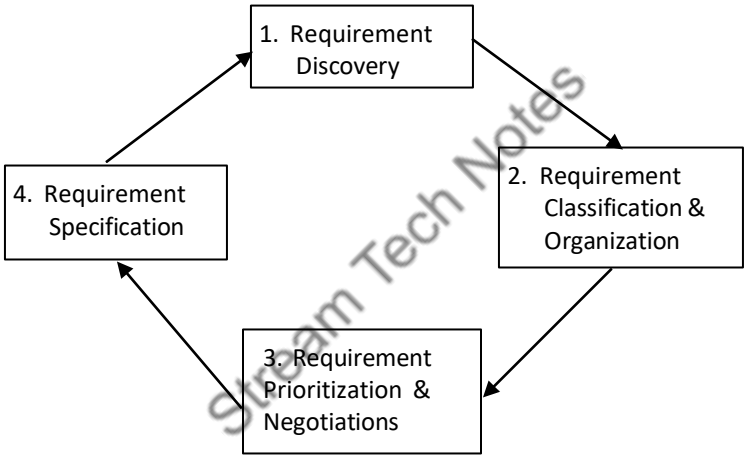


Figure 2.1 Requirement Prototype

ANALYSIS MODELING FOR FUNCTION ORIENTED AND OBJECT ORIENTED SOFTWARE DEVELOPMENT :

Analysis model: -

The analysis model must achieve three primary objectives:

- To describe what the customer requires(analysis)
- To establish a basis for the creation of software with a combination of text and design are used to represent the software requirement.
- To define a set of requirements that can be validated once the software is built.

The elements of analysis model: -

At the core of the model lies the data dictionary—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the core. The entity relation diagram (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

The data flow diagram (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and sub functions) that transform the data flow.

The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a process specification (PSPEC).

The **state transition diagram** (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modes of behavior (called states) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control flow in the **control specification** (CSPEC). **Process specification** describes each function in DFD. **Data object description** of various data object used.

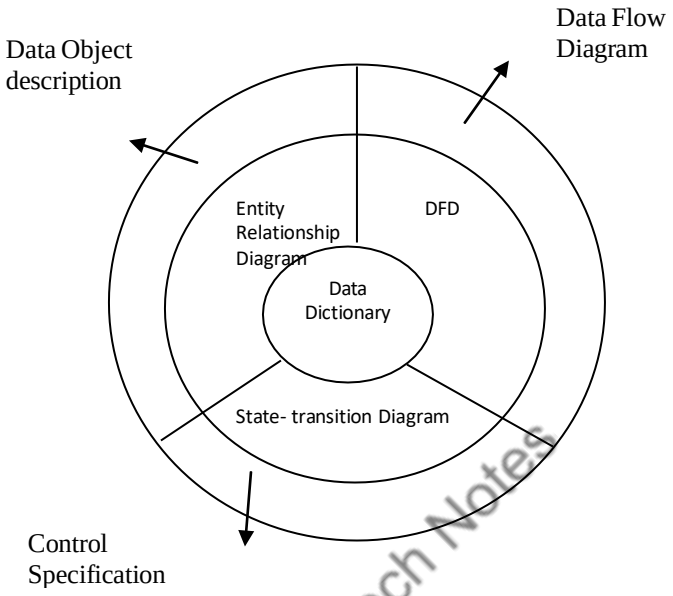


Figure 2.2: State Transition Diagram

Analysis and Modeling: -

Structured Approach

Data Modeling -> ERD

Functional Modeling -> DFD

Behavior Modeling -> State chart diagram

Object Oriented Approach (UML)

- Use case
- Class diagram
- Activity diagram
- Sequence diagram
- Deployment diagram
- Component diagram

The structured approach - plan the right way first

- plans to avoid crisis
- covers all eventualities
- useful for team working
- shorter in the end

In SA data object are modeled in a way in which data attributes and their relationship is defined in structured approach.

Data Flow Diagram (DFD):

- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the

different processing activities or functions that the system

- Performs and the data interchange among these functions. Each function is considered as a processing station (or process) that consumes some input data and produces some output data. The system is represented in terms of the input
- Data to the system, various processing carried out on these data, and the output
- Data generated by the system. A DFD model uses a very limited number of primitive symbols as shown in below Figure to represent the functions performed by a system and the data flow among these functions.

Levels of DFD

- **Level 0** - Highest abstraction level DFD is known as Level 0 DFD, which depicts the entire information system as one diagram concealing all the underlying details. Level 0 DFDs are also known as context level DFDs.

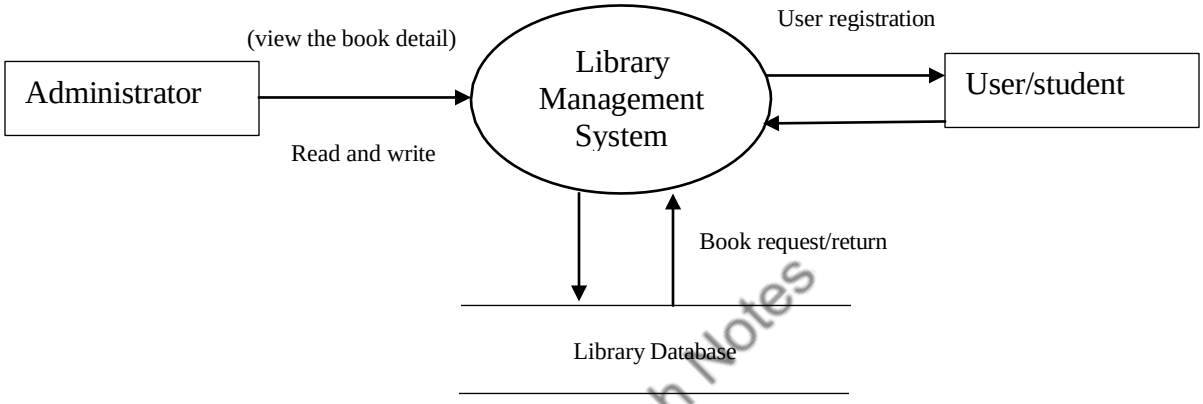


Figure 2.3: Level 0 DFD

Level 1 - The Level 0 DFD is broken down into more specific, Level 1 DFD. Level 1DFD depicts basic modules in the system and flow of data among various modules. Level 1 DFD also mentions basic processes and sources of information.

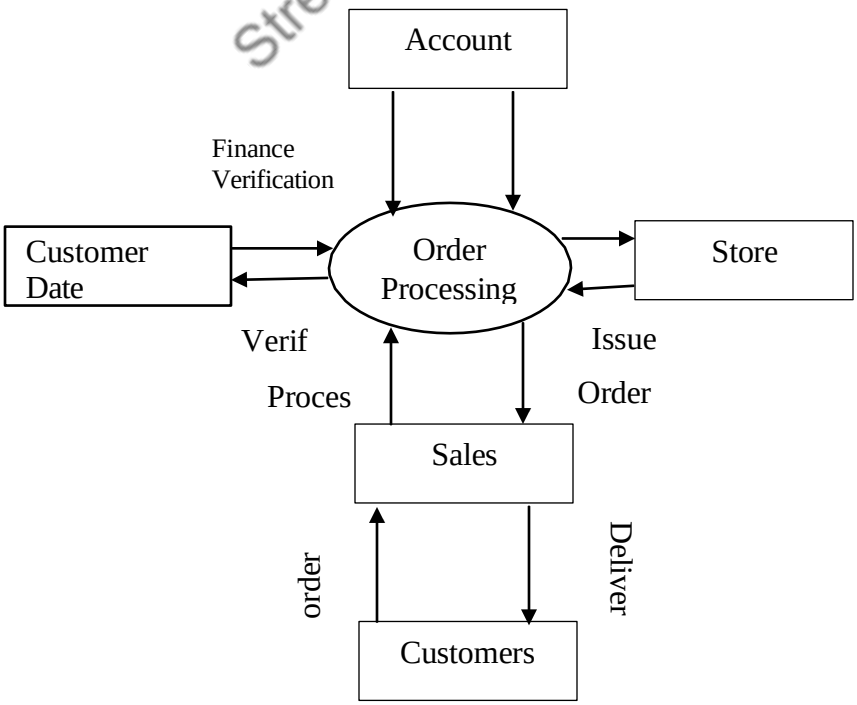


Figure 2.4: Level 1 DFD

Level 2 - At this level, DFD shows how data flows inside the modules mentioned in Level 1.
Higher level DFDs can be transformed into more specific lower level DFDs with deeper level of understanding unless the desired level of specification is achieved.

Behavior Modeling: -
Structure Charts: -
Structure chart is a chart derived from Data Flow Diagram. It represents the system in more detail than DFD. It breaks down the entire system into lowest functional modules, describes functions and sub-functions of each module of the system to a greater detail than DFD.
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. 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 one module to another module in the direction of the connecting arrow.
- **Data flow arrows:** Arrows are annotated with data name; named data passes from 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.

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.
Example: Structure chart for the RMS software
For this example, the context diagram was drawn earlier.

Object-oriented Software Development: -
Object-oriented design: -
In the object-oriented design approach, the system is viewed as collection of objects (i.e. entities). The state is decentralized among the objects and each object manages its own state information. For example, in a Library Automation Software, each library member may be a separate object with its own data and functions to operate on these data. In fact, the functions defined for one object cannot refer or change data of other objects. Objects have their own internal data which define their state. Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from super class. Conceptually, objects communicate by message passing.

What is a 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.

UML Diagrams:

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 to its users. The users' view captures the external user's view of the system in terms of the functionalities offered by the system. The user's 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.

Structural view: The structural view defines the kinds of objects (classes) important to the understanding 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 realize the 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 their dependencies.

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

USE-CASE MODELING:

Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). Use case modeling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

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

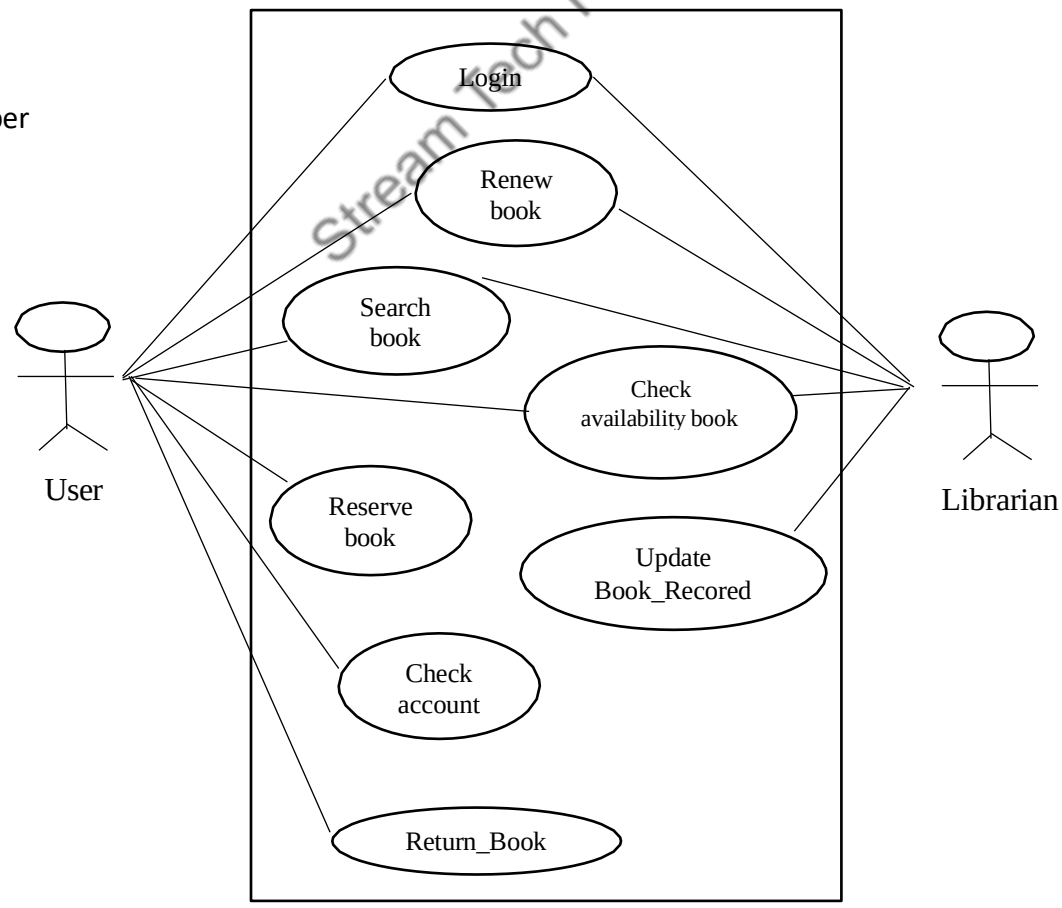


Figure 2.5: Use Case Diagram

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.

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.

Text Description: -

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.

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

Actors: In addition to identifying the actors, some information about actors using this use case which 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 case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, 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 of user'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 user interface 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 documents which may be useful to understand the system operation.

SYSTEM AND SOFTWARE REQUIREMENT SPECIFICATIONS:

Software Specification: -

Software Specification is an activity that is use to describe he thing you are trying to achieve to establish what services are required from the system and limitation on the system operation and development. His activity is often called Requirement Engineering. Requirement Engineering is a particularly critical stage of the software process as error at this stage certain to happen lead to later problem in the system design and implementation.

There are four main phases in the Requirement Engineering process:

- Feasibility study: user satisfaction, cost estimation.
- Requirement elicitation analysis: meeting for description of development.
- Requirement Specification: is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. 2 types of requirements may be including in his document.
- User Requirements (b) System Requirement
- Requirements Validation: this activity checks the requirement for realism, consistency and completeness.

Software Requirement Specification [SRS]: -

A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

The Software Requirements Specification is produced at the culmination of the analysis task. The function and performance allocated to software as part of system engineering are refined by establishing a complete information description, a detailed functional description, a representation of system behavior, an indication of performance requirements and design constraints, appropriate validation criteria, and other information pertinent to requirements.

- A description of each function required to solve the problem is presented in the Functional Description. A processing narrative is provided for each function, design constraints are stated and justified, performance characteristics are stated, and one or more diagrams are included to graphically represent the overall structure of the software and interplay among software functions and other system elements.
- The Behavioral Description section of the specification examines the operation of the software as a consequence of external events and internally generated control characteristics.
- Validation Criteria is probably the most important and, ironically, the most often neglected section of the Software Requirements Specification. How do we recognize a successful implementation? What classes of tests must be conducted to validate function, performance, and constraints? We neglect this section because completing it demands a thorough understanding of software requirements—something that we often do not have at this stage. Yet, specification of validation criteria acts as an implicit review of all other requirements. It is essential that time and attention be given to this section.
- Finally, the specification includes a Bibliography and Appendix. The bibliography contains references to all documents that relate to the software. These include other software engineering documentation, technical references, vendor literature, and standards. The appendix contains information that supplements the specifications. Tabular data, detailed description of algorithms, charts, graphs, and other material are presented as appendixes.

In many cases the Software Requirements Specification may be accompanied by an executable prototype (which in some cases may replace the specification), a paper prototype or a Preliminary User's Manual. The Preliminary User's Manual presents the software as a black box. That is, heavy emphasis is placed on user input and the resultant output. The manual can serve as a valuable tool for uncovering problems at the human/machine interface.

Characteristics of SRS:

- **Correct**: Requirement must be correctly mentioned and realistic by nature.
- **Unambiguous**: Transparent and plain SRS must be written.
- **Complete**: To make the SRS complete I should be specified what a software designer wants to create on software.
- **Consistent**: If there are not conflicts in the specified requirement then SRS is said to be consistent.
- **Stability**: The SRS must contain all the essential requirement. Each requirement must be clear and

explicit.

- **Verifiable**: the SRS should be written in such a manner that the requirement that is specified within it must be satisfied by the software.
- **Modifiable**: It can easily modify according to user requirement.
- **Traceable**: If origin of requirement is properly given of the requirement are correctly mentioned then such a requirement is called as traceable requirement.

REQUIREMENTS VALIDATION:

The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all system requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the formal technical review. The review team includes system engineers, customers, users, and other stakeholders who examine the system specification 5 looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies , conflicting requirements, or unrealistic (unachievable) requirements.

Although the requirements validation review can be conducted in any manner that results in the discovery of requirements errors, it is useful to examine each requirement against a set of checklist questions. The following questions represent a small subset of those that might be asked:

- Are requirements stated clearly? Can they be misinterpreted?
- Is the source (e.g., a person, a regulation, a document) of the requirement identified? Has the final statement of the requirement been examined by or against the original source?
- Is the requirement bounded in quantitative terms?
- What other requirements relate to this requirement? Are they clearly noted via a cross-reference matrix or other mechanism?
- Does the requirement violate any domain constraints?
- Is the requirement testable? If so, can we specify tests (sometimes called validation criteria) to exercise the requirement?
- Is the requirement traceable to any system model that has been created?
- Is the requirement traceable to overall system/product objectives?
- Is the system specification structured in a way that leads to easy understanding, easy reference, and easy translation into more technical work products?
- Has an index for the specification been created?
- Have requirements associated with system performance, behavior, and operational characteristics been clearly stated? What requirements appear to be implicit?

Requirements Management: -

Requirements management is a set of activities that help the project team to identify, control, and track requirements and changes to requirements at any time as the project proceeds

Once requirements have been identified, traceability tables are developed. Shown schematically in Figureure, each traceability table relates identified requirements to one or more aspects of the system or its environment. Among many possible traceability tables are the following:

- Features traceability table. Shows how requirements relate to important customer observable system/product features.
- Source traceability table. Identifies the source of each requirement
- Dependency traceability table. Indicates how requirements are related to one another.
- Subsystem traceability table. Categorizes requirements by the subsystem(s) that they govern.

- Interface traceability table. Shows how requirements relate to both internal and external system interfaces.

In many cases, these traceability tables are maintained as part of a requirements database so that they may be quickly searched to understand how a change in one requirement will affect different aspects of the system to be built.

TRACEABILITY:

Traceability is a property of an element of documentation or code that indicates the degree to which it can be traced to its origin or "reason for being". Traceability also indicates the ability to establish a predecessor-successor relationship between one work product and another.

A work product is said to be traceable if it can be proved that it complies with its specification. For example, a software design is said to be traceable if it satisfies all the requirements stated in the software requirements specification. Examples of traceability include:

- External source to system requirements
- System requirements to software requirements
- Software requirements to high level design
- High level design to detailed design
- Detailed design to code
- Software requirement to test case.

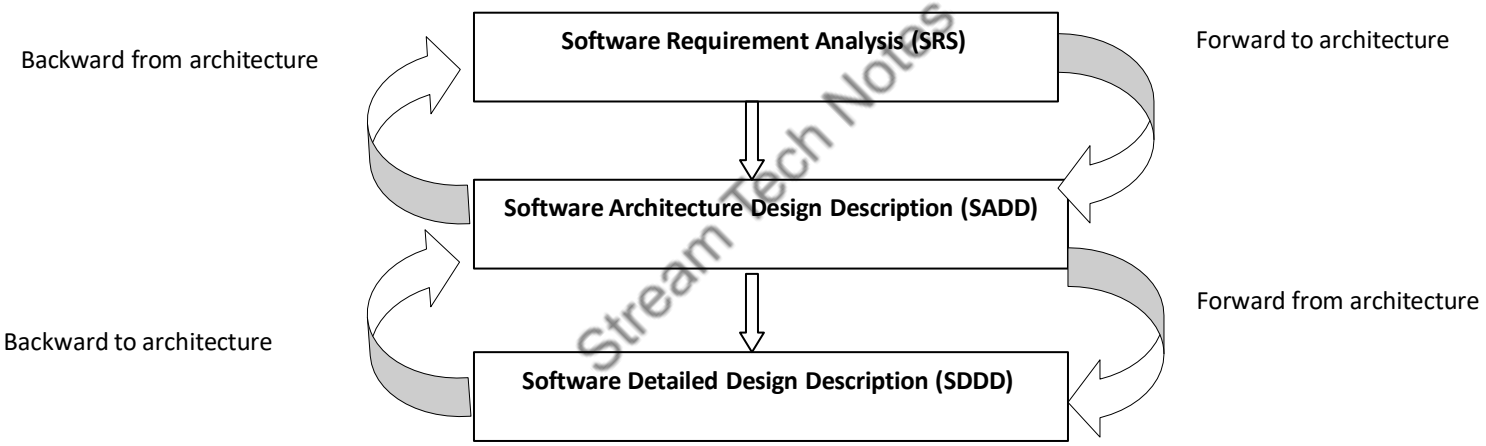


Figure 2.6: Tracing a Software Architecture Design Description

UNIT-III

SOFTWARE DESIGN PROCESS:

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.

The architectural design defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system.

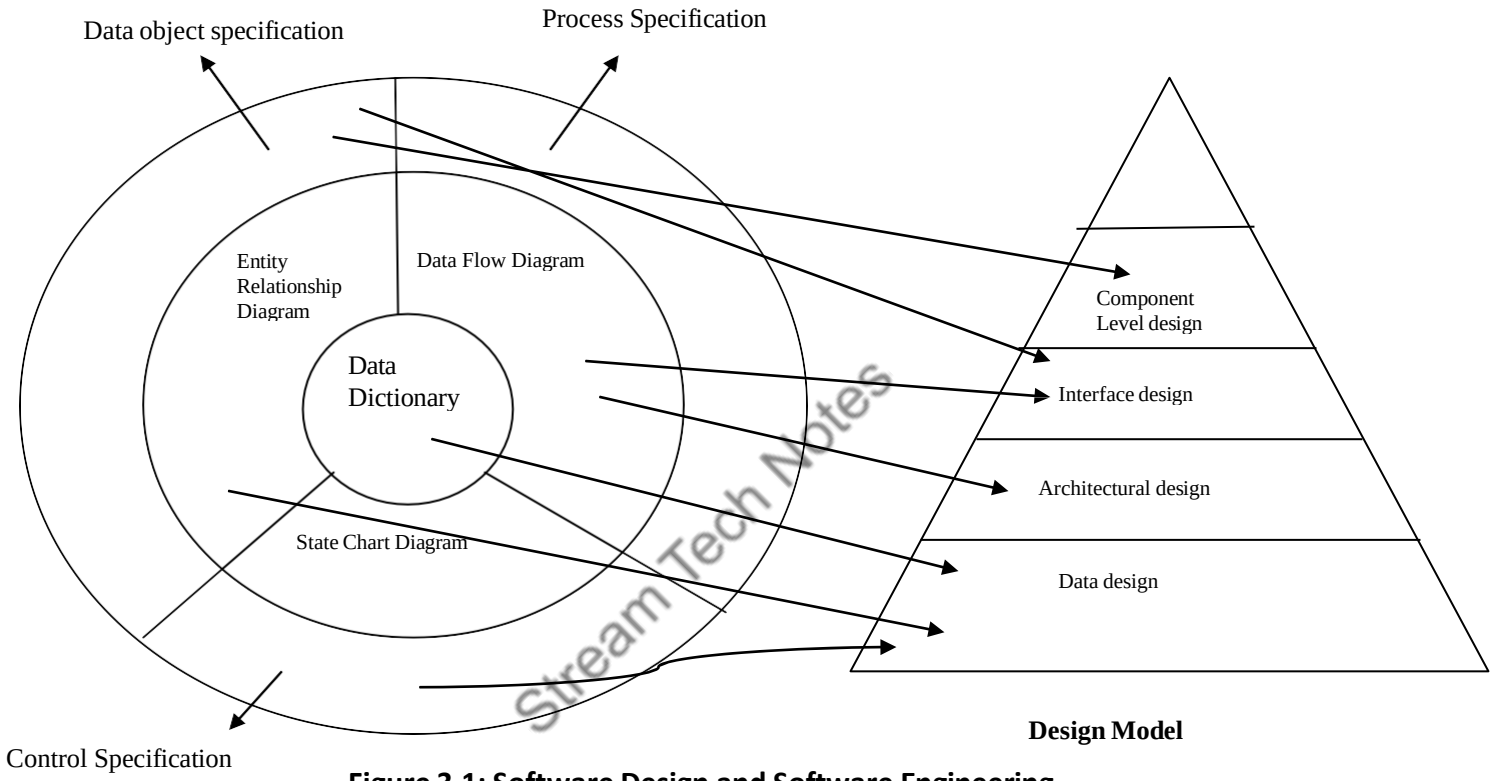


Figure 3.1: Software Design and Software Engineering

The Design Process:

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction at level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower level of abstraction. These can still be trace or requirements, but the connection is subtler.

DESIGN CONCEPTS AND PRINCIPLES:

Design Principles: -

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design.

The design model is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide

guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process.

Principles for software design, which have been adapted and extended in the following list:

- The design process should not suffer from "tunnel vision."
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the software and the problem as it exists in the real world.
- That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- The design should be assessed for quality as it is being created, not after the fact.
- The design should be reviewed to minimize conceptual (semantic) errors.

Design concepts:

Following issues are considered while designing the software.

- **Abstraction:** "Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level detail. At the highest level of abstraction a solution is stated in broad terms using the language of the problem environment. At lower level, a procedural orientation is taken. At the lowest level of abstraction the solution is stated in a manner that can be directly implemented.
  Types of abstraction: 1. Procedural Abstraction 2. Data Abstraction
- **Refinement:** Process of elaboration. Refinement function defined at the abstract level, decompose the statement of function in a stepwise fashion until programming language statements are reached.
- **Modularity:** software is divided into separately named and addressable components called modules. Follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces.

SOFTWARE MODELING AND UML:

Software modeling:

Software models are ways of expressing a software design. Usually some sort of abstract language or pictures are used to express the software design. For object-oriented software, an object modeling language such as UML is used to develop and express the software design.

Unified Modeling Language (UML):

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the Unified Modeling Language (UML), has become widely used throughout the industry. UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules. In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams. The following views are present in UML:

- **User model view.** This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective.

- **Structural model view.** Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.
- **Behavioral model view.** This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the inter actions or collaborations between various structural elements described in the user model and structural model views.
- **Implementation model view.** The structural and behavioral aspects of the system are represented as they are to be built.
- **Environment model view.** The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

UML Diagram Types:

There are several types of UML diagrams:

- **User model view represent through**:
  Use-case Diagram: Shows actors, use-cases, and the relationships between them.
- **Structural model view represents through**
  Class Diagram: Shows relationships between classes and pertinent information about classes themselves.

  Object Diagram: Shows a configuration of objects at an instant in time.
- **Behavioral model view represents through**
  Interaction Diagrams: Show an interaction between groups of collaborating objects.

  Two types: **Collaboration diagram** and **sequence diagram**

  Package Diagrams: Shows system structure at the library/package level.

  State Diagram: Describes behavior of instances of a class in terms of states, stimuli, and transitions.

  Activity Diagram: Very similar to a flowchart— shows actions and decision points, but with the ability to accommodate concurrency.

- **Environment model view represent through**
  Deployment Diagram: Shows configuration of hardware and software in a distributed system.
- **Implementation model view represent through**
  Component Diagram: It shows code modules of a system. This code module includes application program, ActiveX control, Java beans and back end databases. It representing interfaces and dependencies among software architect.

ARCHITECTURAL DESIGN:

Establishing the overall structure of a software system.

Objectives:

- To introduce architectural design and to discuss its importance
- To explain why multiple models are required to document software architecture to describe types of architectural model that may be used.
- A high-level model of a thing: -
- Describes critical aspects of the thing Understandable to many stakeholders
- Allows evaluation of the thing's properties before it is built
- Provides well understood tools and techniques for constructing the thing from its blueprint.

ARCHITECTURAL VIEWS AND STYLES:

Architectural Styles:

The builder has used an architectural style as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a pattern for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be determined, but the pattern—a "center hall colonial"—guides the builder in his work.

The software that is built for computer-based systems also exhibits one of many architectural styles.

Each style describes a system category that encompasses

- A set of components (e.g., a database, computational modules) that perform a function required by a system;
- A set of connectors that enable "communication, co-ordinations and cooperation" among components.
- Constraints that define how components can be integrated to form the system; and
- Semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts. In the section that follows, we consider commonly use d architectural patterns for software.

The commonly used architectural styles are:

- **Data-centered Architectures:** A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. A typical Data-centered style. Clients of the are accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.

  Data-centered architectures promote integrity. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

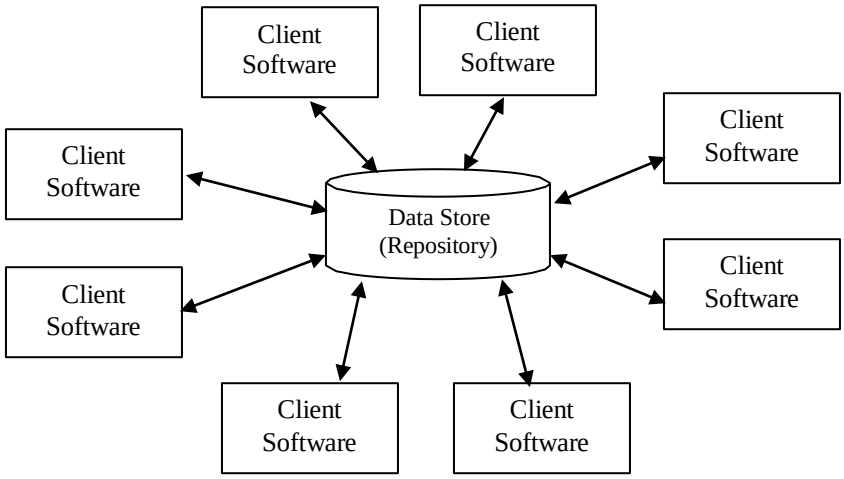
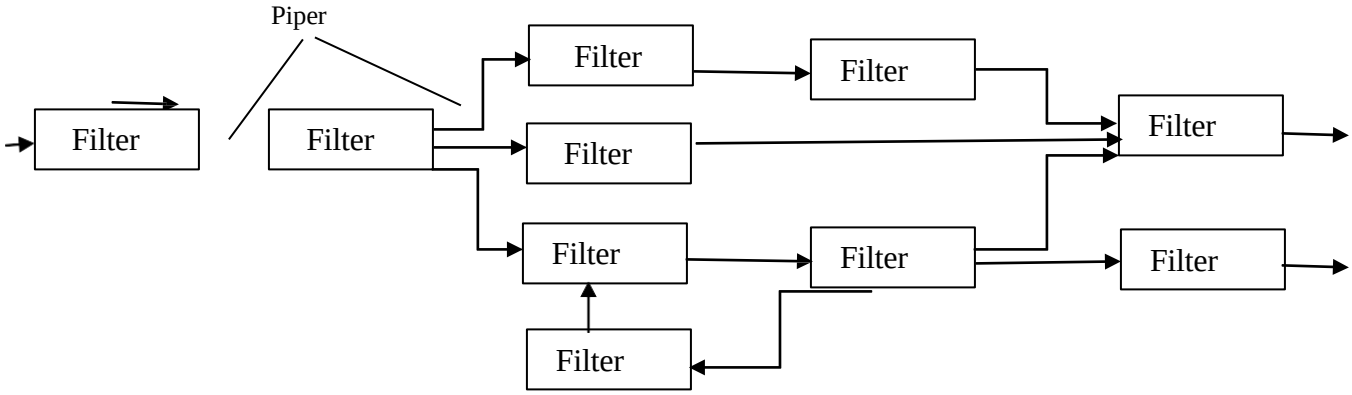


Figure 3.2 Data centered architecture

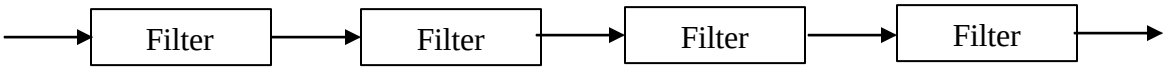
- **Data-flow Architectures:** This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter pattern (F igure 3.3 a) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its

neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern Figure (3.3 b) accepts a batch of data and then applies a series of sequential components (filters) to transform it.



(a) Pipes and Filters



(b) Batch Sequential

Figure 3.3: Data flow architecture

- **Call and return Architectures:** The program structure can be easily modified or scaled. The program structure is organized into modules within the program. In this architecture how modules call each other. The program structure decomposes the function into control hierarchy where a main program invokes number of program components.

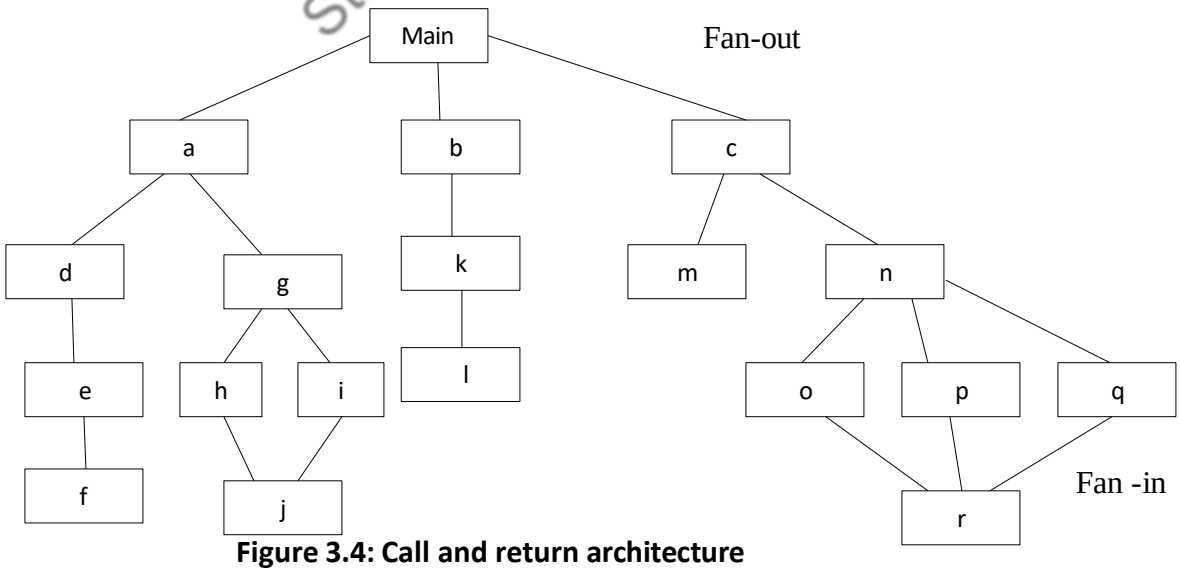


Figure 3.4: Call and return architecture

- **Object-oriented Architecture:** The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

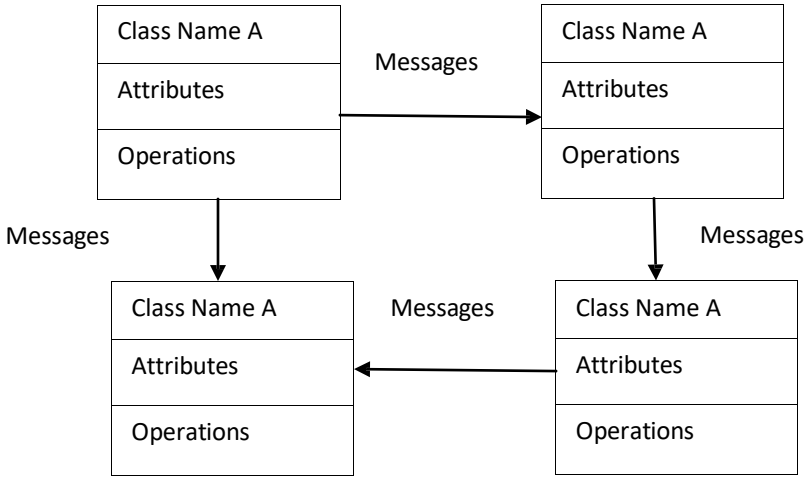


Figure 3.5: Object-oriented Architecture

- **Layered Architectures:** The basic structure of a layered architecture is illustrated in Figure 3.6. A number of different layers are defined, each accomplishing operation that progressively become closer to the machine instruction set.
  At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.
  These architectural styles are only a small subset of those available to the software designer. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

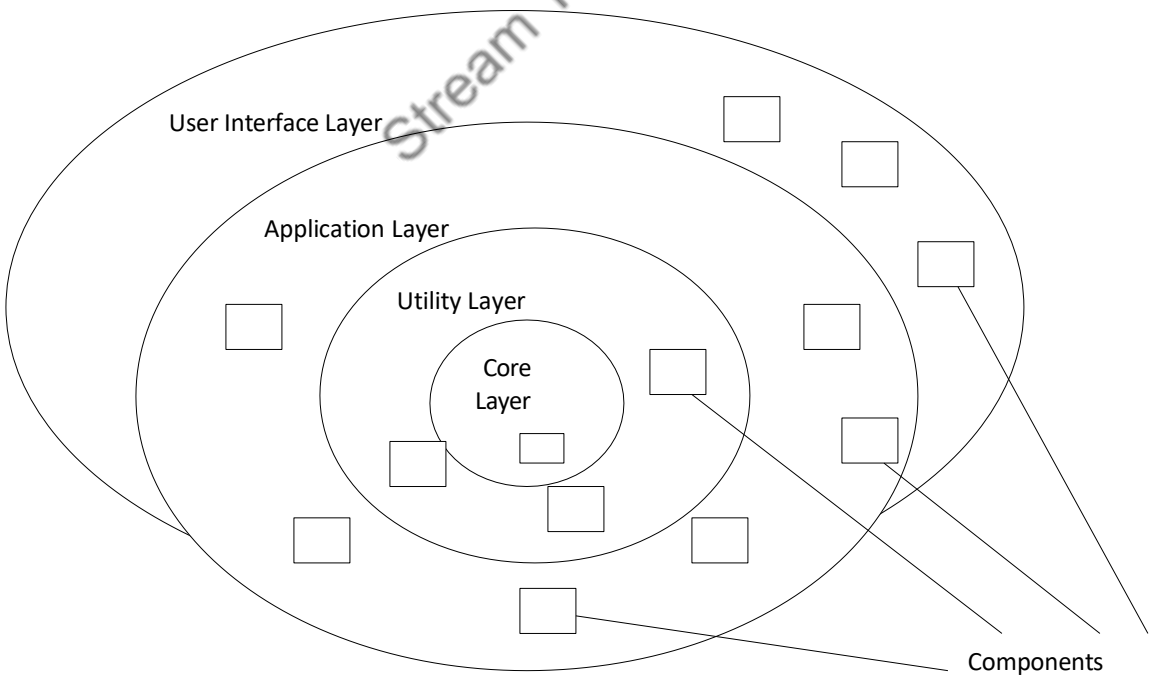


Figure 3.6: Layered Architecture Components

Architectural Views:

4+1 is a architectural view model used for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are

logical, development, process and physical view. In addition selected use cases or scenarios are used to illustrate the architecture serving as the 'plus one' view. Hence the model contains 4+1 views:

- **Development view:** The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.
- **Logical view:** The logical view is concerned with the functionality that the system provides to end-users. UML diagrams used to represent the logical view include, class diagrams, and state diagrams.
- **Physical view:** The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. This view is also known as the deployment view. UML diagrams used to represent the physical view include the deployment diagram.
- **Process view:** The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML diagrams to represent process view include the activity diagram.
- **Scenarios:** The description of architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. The scenarios describe sequences of interactions between objects and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as the use case view.

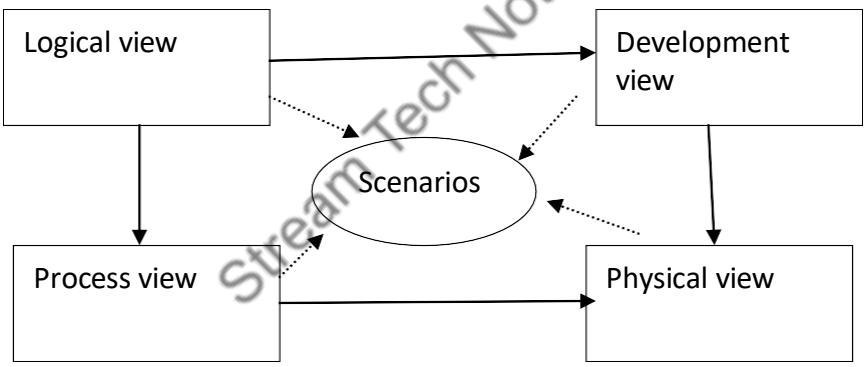


Figure 3.7: 4+1 Architectural View Model

USER INTERFACE DESIGN:

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. User interface provides fundamental platform for human-computer interaction.

User interface can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

User interface is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

User Interface Design Principles:

The principles of user interface design are intended to improve the quality of user interface design.

- **The structure principle:** Design should organize the user interface purposefully, in meaningful and useful ways based on clear, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.
- **The simplicity principle:** The design should make simple, common tasks easy, communicating clearly and simply in the user's own language, and providing good shortcuts that are meaningfully related to longer procedures.
- **The visibility principle:** The design should make all needed options and materials for a given task visible without distracting the user with extraneous or redundant information. Good designs don't overwhelm users with alternatives or confuse with unneeded information.
- **The feedback principle:** The design should keep users informed of actions or interpretations, changes of state or condition, and errors or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.
- **The tolerance principle:** The design should be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing, while also preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.
- **The reuse principle:** The design should reuse internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency, thus reducing the need for users to rethink and remember.

User Interface Analysis and Design:

User interface analysis and design can be done with the help of following steps:

- Create different models for the system functions.
- In order to perform these functions identify the human-computer interface tasks.
- Prepare all interface designs by solving various design issues.
- Apply modern tools and techniques to prototype the design.
- Implement design model.
- Evaluate the design from end user to bring quality in it.

These steps can be broadly categorized in two classes.

1. Interface analysis and design models
2. The process

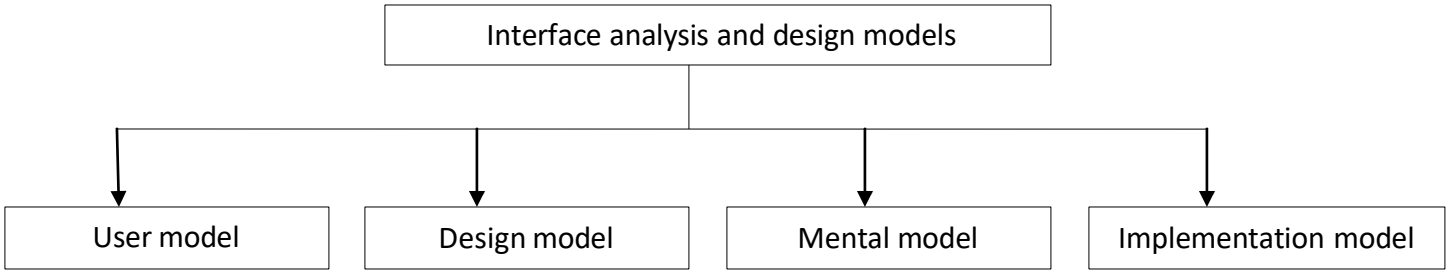


Figure 3.8: Interface Analysis and Design Model

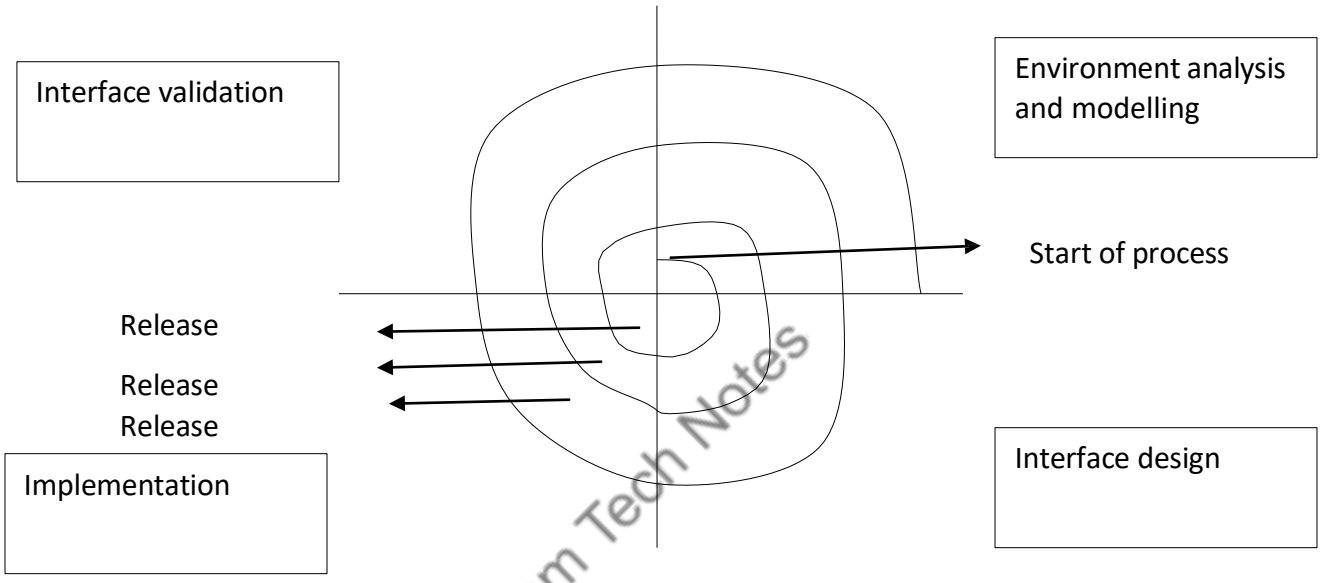


Figure 3.9: Interface Analysis and Design Process

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.

Functional design process:

- **Data-flow design:** Model the data processing in the system using data-flow diagrams.
- **Structural decomposition:** Model how functions are decomposed to sub-functions using graphical structure charts.
- **Detailed design:** The entities in the design and their interfaces are described in detail. These may be recorded in a data dictionary and the design expressed using a PDL.

SA/SD COMPONENT BASED DESIGN:

Structured Analysis and Structured Design: Structured analysis is a set of techniques and graphical tools that allow the analyst to develop a new kind of system specification that are easily understandable to the user.

Goals of SASD

- Improve Quality and reduce the risk of system failure
- Establish concrete requirements specifications and complete requirements documentation
- Focus on Reliability, Flexibility, and Maintainability of system

Component Based Design:

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.

The primary objective of component-based architecture is to ensure component reusability. A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit. Component-oriented software design has many advantages over the traditional object-oriented approaches such as −

- Reduced time in market and the development cost by reusing existing components.
- Increased reliability with the reuse of the existing components.

Principles of Component-Based Design

A component-level design can be represented by using some intermediary representation (e.g. graphical, tabular, or text-based) that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.

It has following salient features −

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plug-in to offer another plug-in API.

Component-Level Design Guidelines

It creates naming conventions for components that are specified as part of the architectural model and then refines or elaborates as part of the component-level model.

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.

- Recognizes and discover these independent entities as new components.
- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

DESIGN METRICS

In software development, a metric is the measurement of a particular characteristic of a program's performance or efficiency. Design metric measure the efficiency of design aspect of the software. Design model considering three aspects:

- Architectural design
- Object oriented design
- User interface design

Architectural Design Metrics:

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Object Oriented Design Metrics:

There are nine measurable characteristics of object oriented design and those are:

- **Size:** It can be measured using following factors:
  - Population: means total number of classes and operations.
  - Volume: means total number of classes or operation that is collected dynamically.
  - Length: means total number of interconnected design elements.
  - Functionality: is a measure of output delivered to the customer.
- **Complexity:** It is measured representing the characteristics that how the classes are interrelated with each other.
- **Coupling:** It is a measure stating the collaboration between classes or number of messages that can be passed between the objects.
- **Completeness:** It is a measure representing all the requirements of the design component.
- **Cohesion:** It is a degree by which we can identified the set of properties that are working together to solve particular problem.
- **Sufficiency:** It is a measure representing the necessary requirements of the design component.
- **Primitiveness:** The degree by which the operations are simple, i.e. number of operations independent from other.
- **Similarity:** the degree by which we measure that two or more classes are similar with respect to their functionality and behavior.
- **Volatility:** Is the measure that represents the probability of changes that will occur.

User Interface Design Metrics:

Although there is significant literature on the design of human/computer interfaces, relatively little information has been published on metrics that would provide insight into the quality and usability of the interface.

- **Layout appropriateness (LA)** is a worthwhile design metric for human/computer interfaces. A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next.
- **Cohesion metrics** can be defined as the relative connection of on screen content to other screen contents. UI cohesion for screen is high.

UNIT-IV

SOFTWARE STATIC AND DYNAMIC ANALYSIS:

Static Analysis:

Static analysis involves no dynamic execution of the software under test and can detect possible defects in an early stage, before running the program.

Static analysis is done after coding and before executing unit tests.

Static analysis can be done by a machine to automatically "walk through" the source code and detect non complying rules. The classic example is a compiler which finds lexical, syntactic and even some semantic mistakes.

Static analysis can also be performed by a person who would review the code to ensure proper coding standards and conventions are used to construct the program.

Static code analysis advantages:

- It can find weaknesses in the code at the exact location.
- It can be conducted by trained software assurance developers who fully understand the code.
- Source code can be easily understood by other or future developers
- It allows a quicker turn around for fixes
- Weaknesses are found earlier in the development life cycle, reducing the cost to fix.
- Less defects in later tests
- Unique defects are detected that cannot or hardly be detected using dynamic tests
    - Unreachable code
    - Variable use (undeclared, unused)
    - Uncalled functions
    - Boundary value violations

Static code analysis limitations:

- It is time consuming if conducted manually.
- Automated tools produce false positives and false negatives.
- There are not enough trained personnel to thoroughly conduct static code analysis.
- Automated tools can provide a false sense of security that everything is being addressed.
- Automated tools only as good as the rules they are using to scan with.
- It does not find vulnerabilities introduced in the runtime environment.

Dynamic Analysis:

Dynamic analysis is based on the system execution, often using tools.

Dynamic program analysis is the analysis of computer software that is performed with executing programs built from that software on a real or virtual processor (analysis performed without executing programs is known as static code analysis). Dynamic program analysis tools may require loading of special libraries or even recompilation of program code.

The most common dynamic analysis practice is executing Unit Tests against the code to find any errors in code.

Dynamic code analysis advantages:

- It identifies vulnerabilities in a runtime environment.
- It allows for analysis of applications in which you do not have access to the actual code.
- It identifies vulnerabilities that might have been false negatives in the static code analysis.
- It permits you to validate static code analysis findings.
- It can be conducted against any application.

Dynamic code analysis limitations:

- Automated tools provide a false sense of security that everything is being addressed.
- Cannot guarantee the full test coverage of the source code
- Automated tools produce false positives and false negatives.

- Automated tools are only as good as the rules they are using to scan with.
- It is more difficult to trace the vulnerability back to the exact location in the code, taking longer to fix the problem.

CODE INSPECTIONS:

Code Inspection is the most formal type of review, which is a kind of static testing to avoid the defect multiplication at a later stage.

- The main purpose of code inspection is to find defects and it can also spot any process improvement if any.
- An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.
- Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.
- Inspections are often led by a trained moderator, who is not the author of the code.
- The inspection process is the most formal type of review based on rules and checklists and makes use of entry and exit criteria.
- It usually involves peer examination of the code and each one has a defined set of roles.
- After the meeting, a formal follow-up process is used to ensure that corrective action is completed in a timely manner.

SOFTWARE TESTING FUNDAMENTALS:

Software testing is an activity performed to uncover errors. It is a critical element of software quality assurance and represents the ultimate review of specification, design and coding. The purpose of software testing is to ensure whether the software functions appear to be working according to specification and performance requirements.

Testing objective:

- Testing is a process of executing a program with the intend of finding an error.
- A good test case is one that has high probability of finding an undiscovered error.
- A successful test is one that uncovers an as yet undiscovered error.

Testing principles:

The following basic principles and fundamentals are general guidelines applicable for all types of real-time testing:

- Testing proves the presence of defects. It is generally considered better when a test reveals defects than when it is error-free.
- Testing the product should be accomplished considering the risk factor and priorities
- Early testing helps identify issues prior to the development stage, which eases error correction and helps reduce cost
- Normally a defect is clustered around a set of modules or functionalities. Once they are identified, testing can be focused on the defective areas, and yet continue to find defects in other modules simultaneously.
- Testing will not be as effective and efficient if the same kinds of tests are performed over a long duration.
- Testing has to be performed in different ways, and cannot be tested in a similar way for all modules. All testers have their own individuality, likewise the system under test.
- Just identifying and fixing issues does not really help in setting user expectations. Even if testing is performed to showcase the software's reliability, it is better to assume that none of the software products are bug-free.

SOFTWARE TEST PROCESS:

Testing is a process rather than a single activity. This process starts from test planning then designing test cases, preparing for execution and evaluating status till the test closure. So, we can divide the activities within the fundamental test process into the following basic steps:

- Planning and Control
- Analysis and Design
- Implementation and Execution
- Evaluating exit criteria and Reporting
- Test Closure activities

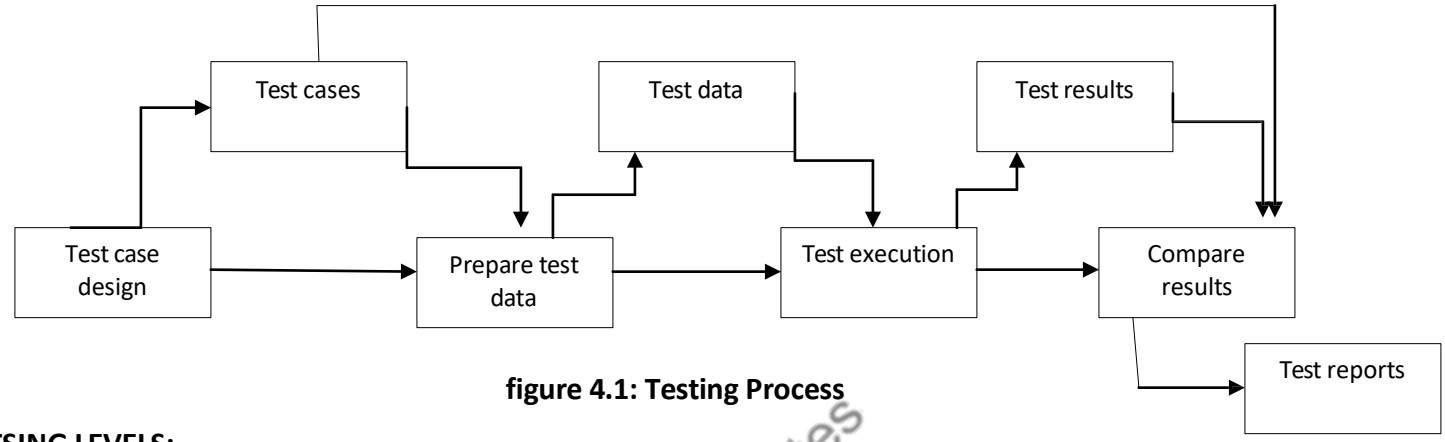


figure 4.1: Testing Process

TETSING LEVELS: -

The testing can be typically carried out in levels. In software development process at each phase some faults may get introduced. These faults are eliminated in the next software development phase but at the same time some new faults may get introduced. Each level of testing performs some typical activity. Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Unit Testing: In this type of testing errors are detected from each software component individually.

Integration Testing: In this type of testing technique interacting component are verified and the interface errors are detected.

System Testing: In this type of testing all the system elements forming the system is tested as a whole.

Acceptance Testing: Acceptance testing is a kind of testing conducted to ensure that the software works correctly in user's working environment.

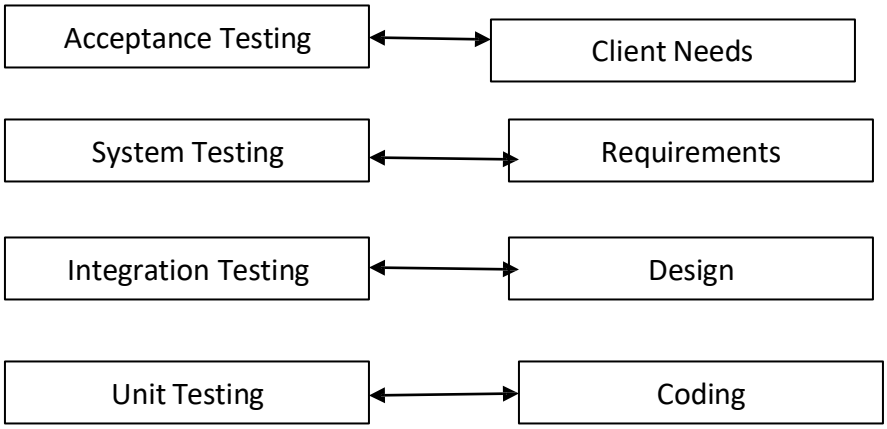


Figure 4.2: Levels of Testing

TEST CRITERIA AND TEST CASE DESIGN:

- Test cases are used to determine the presence of fault in the program.
- Executing test cases require money because- 1)machine time is required to execute test cases
  2) Human efforts are involved in executing test case. Hence in the project testing minimum number of test vases should be there as far as possible.
- The testing activity should involve two goals-1) Maximize the number of errors detected. 2) Minimize the number of test cases.
- The selection of test case should be such that faulty module or program segment must be exercised by at least one test case.
- Test selection criterion can be defined as the set of conditions that must be satisfied by the set of test cases.
- Testing criterion is based on two fundamental properties – reliability and validity.
- A test criterion is reliable if all the test cases detect same set of errors.
- A test criterion is valid if, for any error in the program there is some test case which causes error in the program.
- Generating test cases to satisfy criteria is complex task.

TEST ORACLES:

Test Oracles is a mechanism for determining whether a test has passed or failed. The use of oracles involves comparing the output(s) of the system under test, for a given test-case input, to the output(s) that the oracle determines that product should have. Suppose we have written 2 test cases one test case is for the program which we want to test and other for the test oracle. If the output of both is the same then that means program behaves correctly otherwise there is some fault in the program.

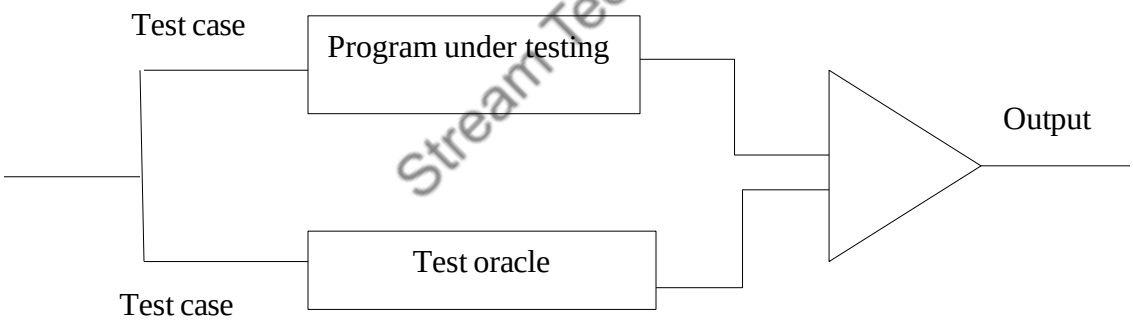


Figure 4.3: Testing with test oracle

TEST TECHNIQUES:

There are various testing techniques are available in which internal structure/design/implementation of the item being tested. There are different methods that can be used for software testing.

- Black box testing
- White box testing
- Integration testing
- Unit testing
- System testing

BLACK BOX TESTING:

Black box testing is also called as behavioral testing. Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life

cycle. This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

Black box testing uncovers following types of errors:

- Incorrect or missing functions
- Interface errors
- Errors in data structures
- Performance errors
- Initialization or termination errors

There are different techniques involved in Black Box testing.

- Equivalence partitioning
- Boundary Value Analysis
- Cause-Effect Graphing.
- Error-Guessing.

Advantages	Disadvantages
• Well suited and efficient for large code segments. • Code access is not required. • Clearly separates user's perspective from the developer's perspective through visibly defined roles. • Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	• Limited coverage, since only a selected number of test scenarios is actually performed. • Inefficient testing, due to the fact that the tester only has limited knowledge about an application. • Blind coverage, since the tester cannot target specific code segments or error-prone areas. • The test cases are difficult to design.

WHITE BOX TESTING:

White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called glass testing or open-box testing. In order to perform white-box testing on an application, a tester needs to know the internal workings of the code.

The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

White box testing techniques includes:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Advantages	Disadvantages
• As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.	• Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased. • Sometimes it is impossible to look into every

• It helps in optimizing the code. • Extra lines of code can be removed which can bring in hidden defects. • Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	nook and corner to find out hidden errors that may create problems, as many paths will go untested. • It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.

UNIT TESTING:

Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules.
The main aim is to isolate each unit of the system to identify, analyze and fix the defects.

Advantages:

- Reduces Defects in the newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

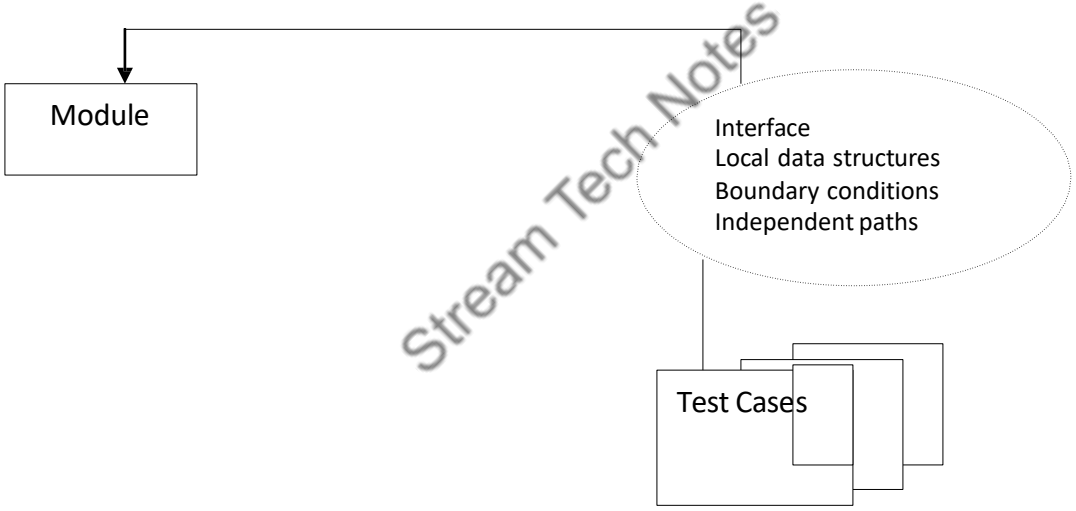


Figure 4.4: Unit Testing

Unit Testing Techniques:

- **Black Box Testing -** Using which the user interface, input and output are tested.
- **White Box Testing -** used to test each one of those functions behavior is tested.
- **Gray Box Testing -** Used to execute tests, risks and assessment methods.

TESTING FRAMEWORKS:

Testing frameworks are an essential part of any successful automated testing process. They can reduce maintenance costs and testing efforts and will provide a higher return on investment (ROI) for QA teams looking to optimize their agile processes.
A testing framework is a set of guidelines or rules used for creating and designing test cases. A framework is comprised of a combination of practices and tools that are designed to help QA professionals test more efficiently.

INTEGRATION TESTING:

In integration testing, individual software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers. Integration Testing focuses on checking data communication amongst these modules.

Need of integration testing:

Although each software module is unit tested, defects still exist for various reasons like:

- A Module in general is designed by an individual software developer whose understanding and programming logic may differ from other programmers. integration Testing becomes necessary to verify the software modules work in unity
- At the time of module development, there are wide chances of change in requirements by the clients. These new requirements may not be unit tested and hence system integration Testing becomes necessary.
- Interfaces of the software modules with the database could be erroneous
- External Hardware interfaces, if any, could be erroneous
- Inadequate exception handling could cause issues.

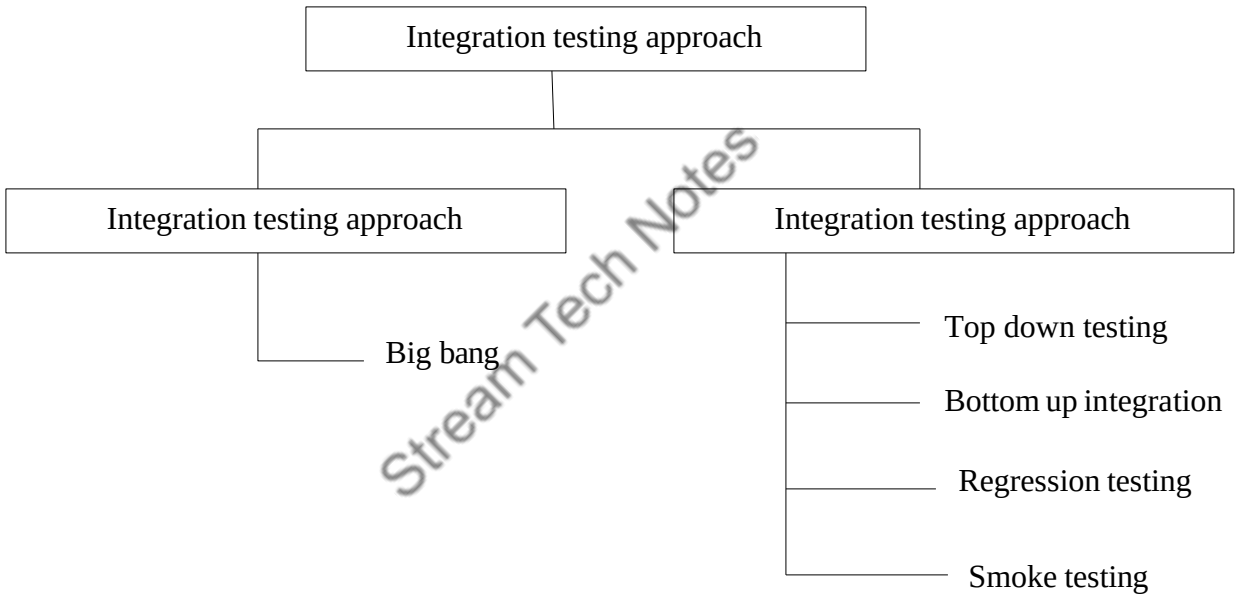


Figure 4.5: Integration testing approach

Bottom-up integration:
This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.

Top-down integration:
In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

Regression Testing:
Regression testing is used to check for defects propagated to other modules by changes made to existing program. Thus regression testing is used to reduce the side effects of the changes.

Smoke Testing:
Smoke testing is a type of software testing which ensures that the major functionalities of the application are

working fine. This testing is also known as 'Build Verification testing'. It is a non-exhaustive testing with very limited test cases to ensure that the important features are working fine and we are good to proceed with the detailed testing.

SYSTEM TESTING:

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.

System testing is important because of the following reasons:

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

OTHER SPECIALIZED TESTING:

There are lots of testing types. Below we have listed types of system testing a large software development company would typically use:

Usability Testing: Usability Testing mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives

Load Testing: Load Testing is necessary to know that a software solution will perform under real-life loads.

Regression Testing: Regression Testing involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.

Recovery Testing: Recovery testing is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.

Migration Testing: Migration testing is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.

Functional Testing: Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.

Hardware/Software Testing: IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

Acceptance Testing: The acceptance testing is a kind of testing to ensure that the software works correctly in the user work environment. Acceptance testing, a testing technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it is has met the required criteria for delivery to end users.

There are various forms of acceptance testing:

- User acceptance Testing
- Business acceptance Testing
- Alpha Testing
- Beta Testing

Stress Testing: Stress testing is the process of determining the ability of a computer, network, program or device to maintain a certain level of effectiveness under unfavorable conditions. It is used to test the stability & reliability of the system. This test mainly determines the system on its robustness and error handling under

extremely heavy load conditions.

TEST PLAN:

Test planning, the most important activity to ensure that there is initially a list of tasks and milestones in a baseline plan to track the progress of the project. It also defines the size of the test effort. It is the main document often called as master test plan or a project test plan and usually developed during the early phase of the project.

S.No.	Parameter	Description
1.	Test plan identifier	Unique identifying reference.
2.	Introduction	A brief introduction about the project and to the document.
3.	Test items	A test item is a software item that is the application under test.
4.	Features to be tested	A feature that needs to tested on the test ware.
5.	Features not to be tested	Identify the features and the reasons for not including as part of testing.
6.	Approach	Details about the overall approach to testing.
7.	Item pass/fail criteria	Documented whether a software item has passed or failed its test.
8.	Test deliverables	The deliverables that are delivered as part of the testing process, such as test plans, test specifications and test summary reports.
9.	Testing tasks	All tasks for planning and executing the testing.
10.	Environmental needs	Defining the environmental requirements such as hardware, software, OS, network configurations, tools required.
11.	Responsibilities	Lists the roles and responsibilities of the team members.
12.	Staffing and training needs	Captures the actual staffing requirements and any specific skills and training requirements.
13.	Schedule	States the important project delivery dates and key milestones.
14.	Risks and Mitigation	High-level project risks and assumptions and a mitigating plan for each identified risk.
15.	Approvals	Captures all approvers of the document, their titles and the sign off date.

Table 4.1 Test Plan Identifiers

Test Planning Activities:

- To determine the scope and the risks that need to be tested and that are NOT to be tested.
- Documenting Test Strategy.
- Making sure that the testing activities have been included.
- Deciding Entry and Exit criteria.
- Evaluating the test estimate.
- Planning when and how to test and deciding how the test results will be evaluated, and defining test exit criterion.
- The Test art facts delivered as part of test execution.
- Defining the management information, including the metrics required and defect resolution and risk issues.
- Ensuring that the test documentation generates repeatable test assets.

TEST METRICS:

In software testing, Metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute. Measurement is nothing but quantitative indication of size / dimension / capacity of an attribute of a product / process. Software metric is defined as a quantitative measure of an

attribute a software system possesses with respect to Cost, Quality, Size and Schedule.
Example-
Measure - No. of Errors
Metrics - No. of Errors found per person
The most commonly used metric is cyclomatic complexity and Hallstead complexity.

Cyclomatic complexity: Cyclomatic complexity is software metric used to measure the complexity of a program. These metric, measures independent paths through program source code. Independent path is defined as a path that has at least one edge which has not been traversed before in any other paths. Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.

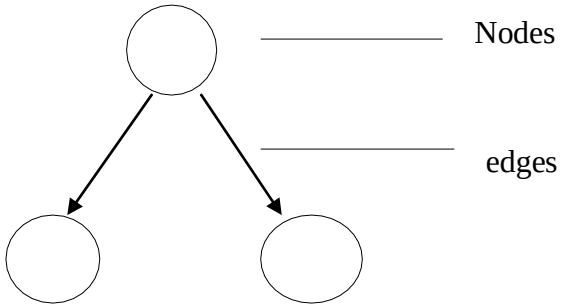


Figure 4.6: Nodes representation

Mathematically, it is set of independent paths through the graph diagram. The complexity of the program can be defined as -

$$V(G) = E - N + 2$$

Where,
E - Number of edges
N - Number of Nodes

$$V(G) = P + 1$$

Where P = Number of predicate nodes (node that contains condition)

Halstead complexity: Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity.
The Halstead effort can be defined as:

$$e = V/PL$$

Where V is the program volume and Pl is the program level. The program level can be computed as:
PL=1/[(n1/2)*(N2/n2)]
Where n1 is total distinct operators,
N2 is total distinct operands and
N2 is all the operand in the program.
The % of overall testing efforts = testing effort of specific module/testing efforts of all the modules

TESTING TOOLS:

Tools from a software testing context can be defined as a product that supports one or more test activities right from planning, requirements, creating a build, test execution, defect logging and test analysis.

Classification of Tools

Tools can be classified based on several parameters. They include:

- The purpose of the tool
- The Activities that are supported within the tool
- The Type/level of testing it supports

- The Kind of licensing (open source, free ware, commercial)

The technology used

S. No.	Tool Type	Used for	Used by
1.	Test Management Tool	Test Managing, scheduling, defect logging, tracking and analysis.	testers
2.	Configuration management tool	For Implementation, execution, tracking changes	All Team members
3.	Static Analysis Tools	Static Testing	Developers
4.	Test data Preparation Tools	Analysis and Design, Test data generation	Testers
5.	Test Execution Tools	Implementation, Execution	Testers
6.	Test Comparators	Comparing expected and actual results	All Team members
7.	Cover age measurement tools	Provides structural coverage	Developers
8.	Performance Testing tools	Monitoring the performance, response time	Testers
9.	Project planning and Tracking Tools	For Planning	Project Managers
10.	Incident Management Tools	For managing the tests	Testers

Table 4.2: Testing tool

Tools Implementation - process

- Analyze the problem carefully to identify strengths, weaknesses and opportunities.
- The Constraints such as budgets, time and other requirements are noted.
- Evaluating the options and Short listing the ones that are meets the requirement.
- Developing the Proof of Concept which captures the pros and cons.
- Create a Pilot Project using the selected tool within a specified team.
- Rolling out the tool phase wise across the organization.

INTRODUCTION TO OBJECT-ORIENTED ANALYSIS AND DESIGN:

Object-oriented analysis and design (OOAD) is a popular technical approach to analyzing, designing an application, system, or business by applying the object-oriented paradigm and visual modeling throughout the development life cycles for better stakeholder communication and product quality.

Object-Oriented Analysis:

Object–Oriented Analysis (OOA) is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.

The primary tasks in object-oriented analysis (OOA) are –

- Identifying objects
- Organizing the objects by creating object model diagram
- Defining the internals of the objects, or object attributes
- Defining the behavior of the objects, i.e., object actions
- Describing how the objects interact

The common models used in OOA are use cases and object models.

Object-Oriented Design:

Object–Oriented Design (OOD) involves implementation of the conceptual model produced during object-oriented analysis. In OOD, concepts in the analysis model, which are technology–independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain.

The implementation details generally include –

- Restructuring the class data (if necessary),
- Implementation of methods, i.e., internal data structures and algorithms,
- Implementation of control, and
- Implementation of associations.

COMPARISON WITH STRUCTURED SOFTWARE ENGINEERING:

Key differences between structured and object oriented analysis and design are as follows:

Phase	Structured	Object Oriented
Analysis	Structuring Requirements • DFD's • Structured English • Decision Table/Tree • ER Analysis	Requirement Engineering • Use Case Model(Find use cases, flow of events) • Object Model – Find classes and class relationship – Object interaction: Sequence and collaboration diagram ,state machine diagram – Object to ER mapping
Design	DB Design • DB normalization GUI Design • Forms and reports	Physical DB design Design elements • Design system architecture • Design classes • Design components GUI Design

Table 4.3: comparison table between structured and object oriented

NEED AND TYPES OF MAINTENANCE:

Software Maintenance: Software maintenance is an activity in which program is modified after it has been put into use. In this usually it is not preferred to apply major software changes to system's architecture. Maintenance is a process in which changes are implemented by either modifying the existing systems architecture or by adding new components into the system.

Need for maintenance:

Software maintenance is widely accepted part of SDLC now a day. It stands for all the modifications and updating done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions:** Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements:** Over the time, customer may ask for new features or functions in the software.
- **Host Modifications:** If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes:** If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

Types of maintenance:

Various types of software maintenance are:

- **Corrective Maintenance:** Means the maintenance for correcting the software faults.
- **Adaptive Maintenance:** Means maintenance for adapting the change in environment (different system or operating systems).
- **Perfective Maintenance:** Means modifying or enhancing the system to meet the new requirements.
- **Preventive Maintenance:** This includes modifications and updating to prevent future problems of the software.

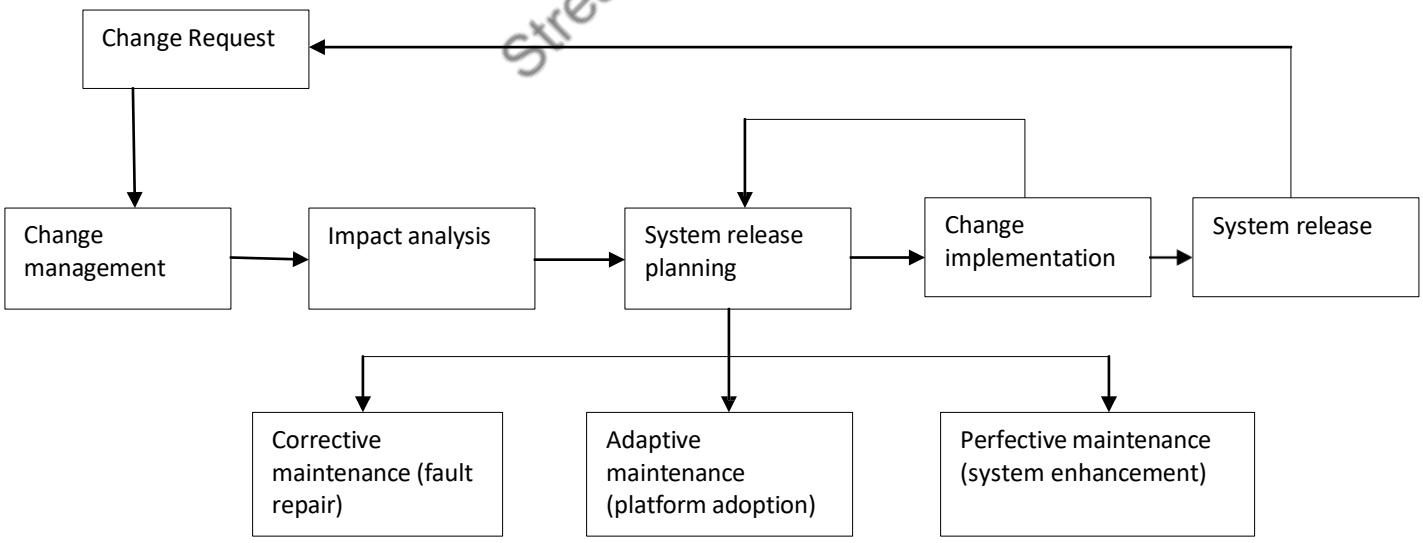


Figure 5.1: Maintenance Process

Issues in software maintenance:

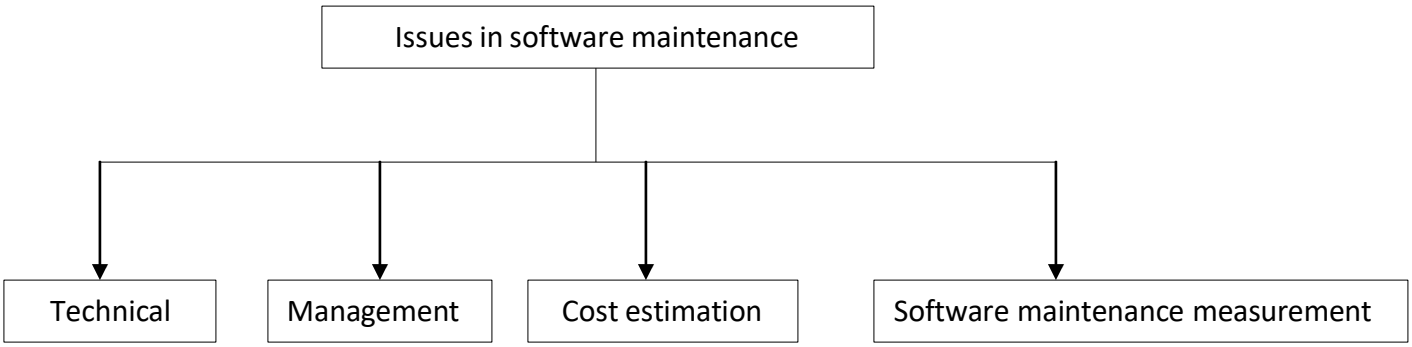


Figure 5.2: Software Maintenance Issues

SOFTWARE CONFIGURATION MANAGEMENT (SCM):

Software configuration management is a set of activities carried out for identifying, organizing and controlling changes throughout the life cycle of computer software.

During the development of software "change must be managed and controlled" in order to improve quality and reduce error.

Hence software configuration management is a quality assurance activity that is applied throughout the software process.

SCM helps to eliminate the confusion often caused by miscommunication among team members. The SCM system controls the basic components such as software objects, program code, test data, test output, design documents and user manuals.

The SCM system has the following advantages:

- Reduced redundant work.
- Effective management of simultaneous updates.
- Avoids configuration-related problems.
- Facilitates team coordination.
- Helps in building management; managing tools used in builds.
- Defect tracking: It ensures that every defect has traceability back to its source.

Software Configuration Items (SCIs): Information that is created as part of the software engineering process.

Baselines: A Baseline is a software configuration management concept that helps us to control change. Signals a point of departure from one activity to the start of another activity. Helps control change without impeding justifiable change.

Elements of SCM

There are four elements of SCM:

1. Software Configuration Identification
2. Software Configuration Control
3. Software Configuration Auditing
4. Software Configuration Status Reporting

The SCM Process

The SCM process defines a series of tasks:

- Identification of objects in the software configuration
- Version Control
- Change Control
- Configuration Audit, and
- Reporting

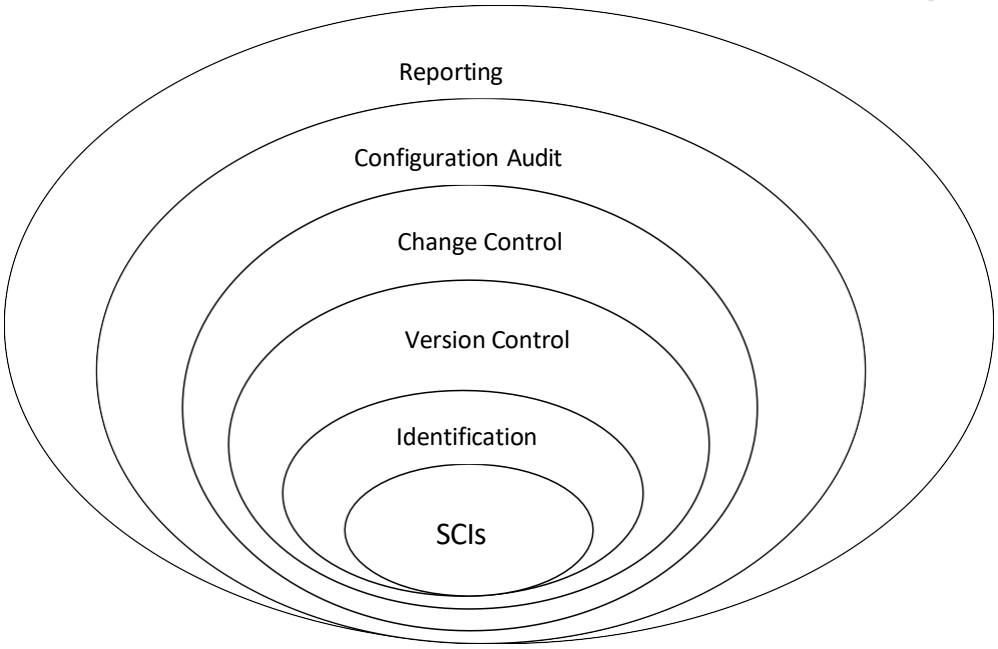


Figure 5.3: SCM Process

SOFTWARE CHANGE MANAGEMENT:

Change control is function of configuration management, which ensures that all changes made to software system are consistent and made as per organizational rules and regulations.

A change in the configuration of product goes through following steps –

- **Identification:** A change request arrives from either internal or external source. When change request is identified formally, it is properly documented.
- **Validation:** Validity of the change request is checked and its handling procedure is confirmed.
- **Analysis:** The impact of change request is analyzed in terms of schedule, cost and required efforts. Overall impact of the prospective change on system is analyzed.
- **Control:** It is decided that the changes are worth incorporation or not. If it is not, change request is refused formally.
- **Execution:** If the previous phase determines to execute the change request, this phase takes appropriate actions to execute the changes, through a thorough revision if necessary.
- **Close request:** The change is verified for correct implementation and merging with the rest of the system. This newly incorporated change in the software is documented properly and the request is formally closed.

VERSION CONTROL:

Version Control is a system or tool that captures the changes to a source code element: file, folder, image or binary. This is beneficial for many reasons, but the most fundamental reason is it allows you to track changes on a per file basis.

Version Control Benefits:

- Secure Access to your Source Code
- File History
- Facilitate Team Communication
- Baseline Trace Ability
- Automated Merge Capabilities
- Ensures no one Over-Writes Someone Else's Code
- Allows for Better Control for Parallel Development

CHANGE CONTROL AND REPORTING:

Change control is a systematic approach to managing all changes made to a product or system. The purpose is to ensure that no unnecessary changes are made, that all changes are documented, that services are not unnecessarily disrupted and that resources are used efficiently. Change control is an essential step in software life cycle. The change control can be carried out using following steps:

- A change request initiates a changes
- The configuration object is "checked out" of the database.
- The changes are applied to the object.
- The object is then "checked in" to the database where automatic version control is applied.

PROGRAM COMPREHENSION TECHNIQUES:

Program comprehension is a domain of computer science concerned with the ways software engineers maintain existing source code.

Program comprehension tools only play a supporting role in other software engineering activities of design, development, maintenance, and re-documentation. Software Engineering discipline which aims at understanding computer code written in a high-level programming language. Program Comprehension is useful for reuse, maintenance, reverse engineering and many other activities in the context of Software Engineering.

Program Comprehension Tool:

A program that aims at making the understanding of a software application easier, through the presentation of different perspectives (views) of the overall system or its components. A PC Tool has modules to:

- Extract information by parsing the source code
- Store and handle the information extracted
- Visualize all the retrieved information

RE-ENGINEERING:

Software re-engineering means re-structuring or re-writing part or all of the software engineering system. It is needed for the application which requires frequent maintenance.

Software re-engineering is a process of software development which is done to improve the maintainability of a software system.

Re-engineering a software system has two key advantages:

- **Reduced risk:** As the software already exists, the risk is less as compared to developing new software.
- **Reduced cost:** The cost of re-engineering is significantly less than the costs of developing new software.

Re-engineering process activities:

- **Source code translation:** In this phase code is converted into new language.
- **Reverse Engineering:** Under this activity the program is analyzed and understood thoroughly.
- **Program structure improvement:** Restructure automatically for understandability.
- **Program modularization:** The program structure is reorganized.
- **Data re-engineering:** Finally clean-up and restructure system data.

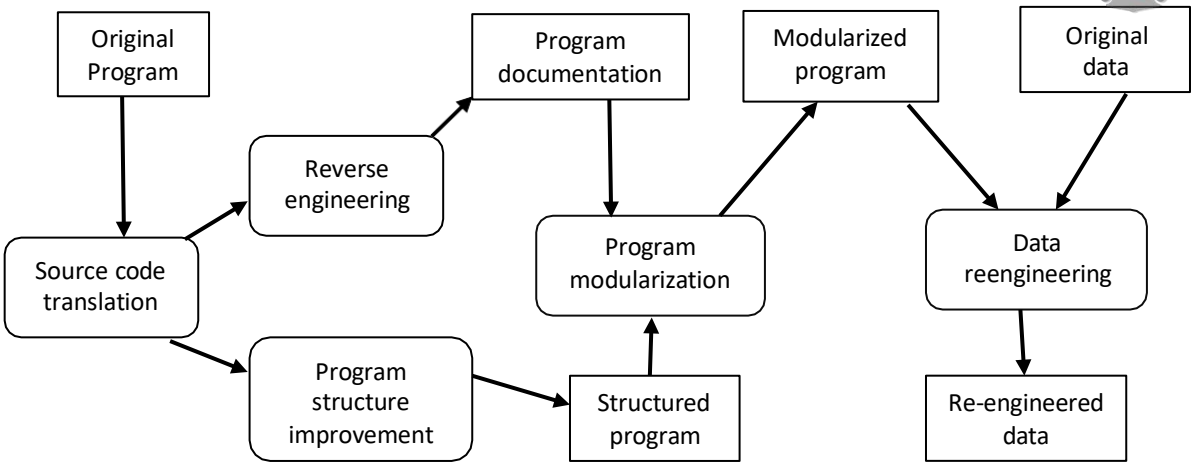


Figure 5.4: Re-engineering process activities

REVERSE ENGINEERING:

Reverse engineering is the process of design recovery. In reverse engineering the data, architectural and procedural information is extracted from a source code.

The reverse engineering is required because using this technique the dirty, ambiguous code can be converted to clear and unambiguous specification. This specification helps in understanding the source code.

There are 3 important issues in reverse engineering:

1. **Abstraction Level:** This level helps in obtaining the design information from the source code. It is expected that abstraction level should be high in reverse engineering.
2. **Completeness Level:** The completeness means detailing of abstract level. The completeness decreases as abstraction level increases.
3. **Directionality Level:** Directionality means extracting the information from source code and gives it to software engineer. The directionality can be one way or two way. The one way directionality means extracting all the information from source code and gives it to software engineer. The two way directionality means the information taken from source code is fed to re-engineering tool that attempts to restructure or regenerate old programs.

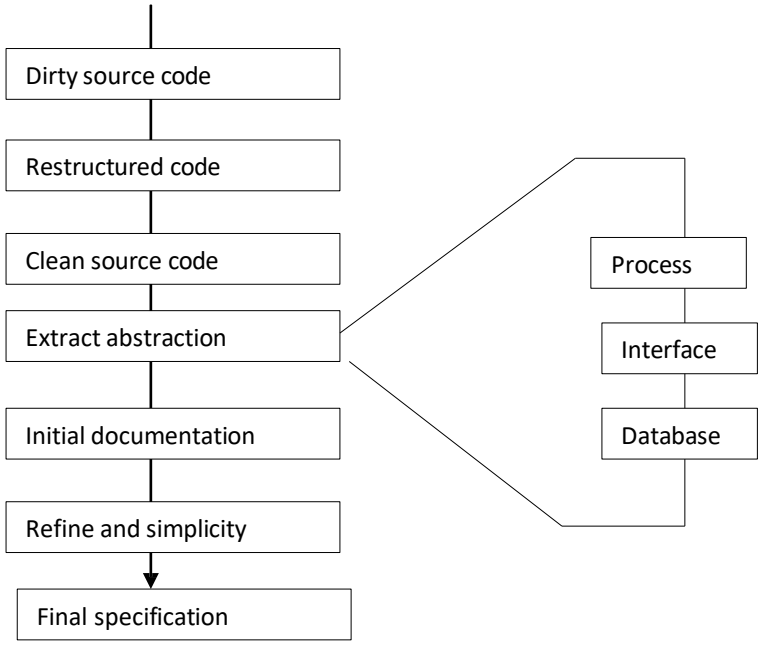


Figure 5.5: Reverse engineering process

Difference between reverse and forward and engineering:

S.No.	Reverse Engineering	Forward Engineering
1.	It performs transformation from a lower abstraction to higher one.	The traditional process of moving from high-level abstractions and logical, implementation independent designs to the physical implementation of a system.
2.	Usually done when docs are not appropriate or is missing.	Modification of the system is done. E.g. 1) Use of different programming language. 2) introduction of new DBMS 3) Transfer of s/w to new h/w platform.
3.	Reverse engineering is a process in which the dirty or unstructured code is taken, processed and it is restructured.	Forward-engineering is a process in which theories, methods and tools are applied to develop a professional software product.
4.	Reverse Engineering is trying to recreate the source code from the compiled code. That is trying to figure out how a piece of software works given only the final system.	Forward engineering is normal engineering. It builds devices that can do certain useful things for us:
5.	It is complex because cleaning the dirty or unstructured code requires more efforts.	It is simple and straight forward approach.
6.	Documentation or specification of the product is useful to the developer.	Documentation or specification of the product is useful to the end user.
7.	This process starts by understanding the existing unstructured code.	This process starts by understanding user requirements.

Table 5.1: Forward and Reverse Engineering

TOOL SUPPORT:

CASE Tool Support:

CASE tools are set of software application programs, which are used to automate SDLC activities. CASE tools are used by software project managers, analysts and engineers to develop software system.

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

Use of CASE tools accelerates the development of project to produce desired result and helps to uncover flaws before moving ahead with next stage in software development.

Components of CASE Tools

CASE tools can be broadly divided into the following parts based on their use at a particular SDLC stage:

- **Central Repository** - CASE tools require a central repository, which can serve as a source of common, integrated and consistent information. Central repository is a central place of storage where product specifications, requirement documents, related reports and diagrams, other useful information regarding management are stored. Central repository also serves as data dictionary.
- **Upper Case Tools** - Upper CASE tools are used in planning, analysis and design stages of SDLC.
- **Lower Case Tools** - Lower CASE tools are used in implementation, testing and maintenance.
- **Integrated Case Tools** - Integrated CASE tools are helpful in all the stages of SDLC, from Requirement gathering to Testing and documentation.

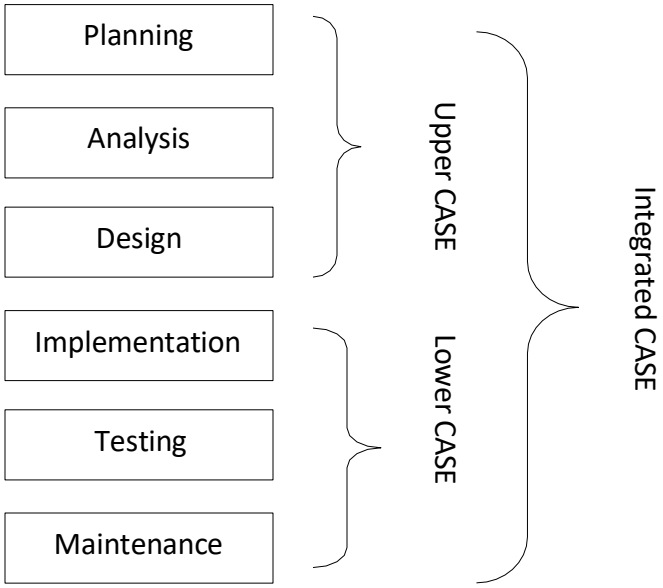


Figure 5.6: CASE Component

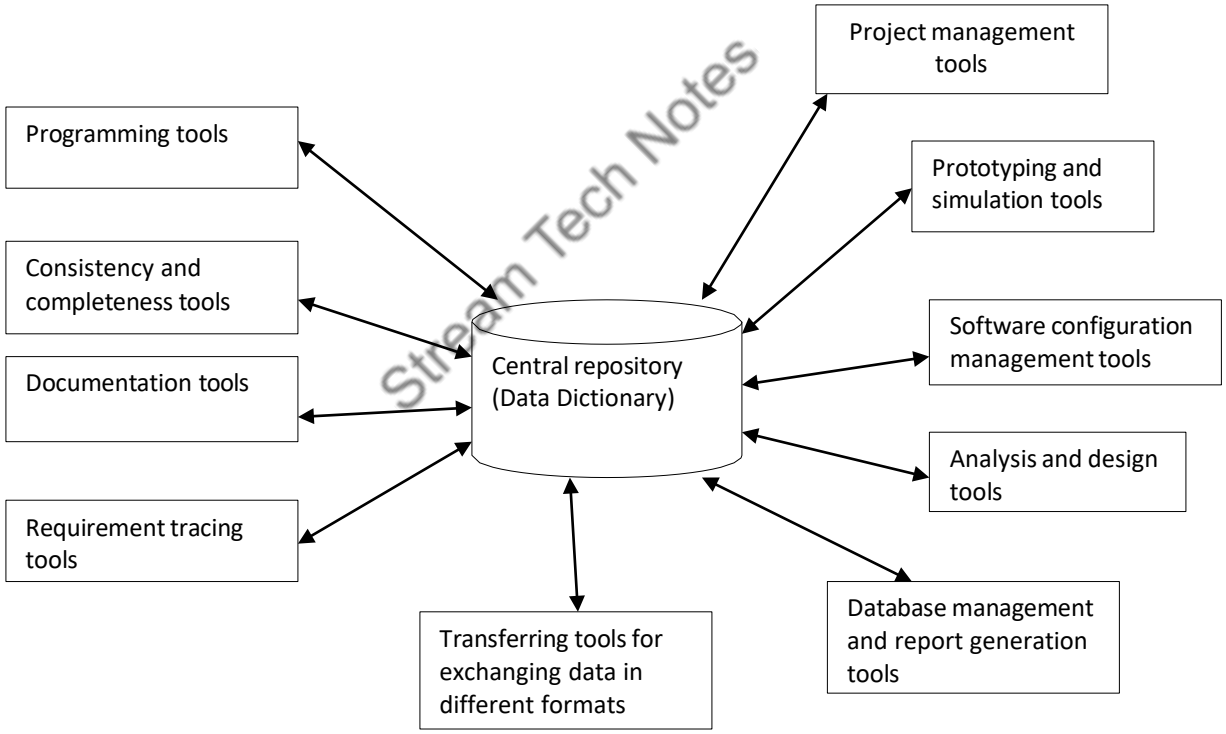


Figure 5.7: Block Diagram for CASE Environment

PROJECT MANAGEMENT CONCEPTS:

Software project management is an activity of organizing, planning and scheduling the software projects. The goal of software project management is to deliver the software product in given time and within the budget. It is also necessary that the software project should be developed in accordance with the requirements of the organization. The project management is the application of knowledge, skill, tools and techniques to project activities to meet the project requirements.

Objectives of project management:

- The objective of the project planning and management is to provide a framework for the project.
- Using the project framework, the project manger decides the estimates for the schedule, cost and resources.
- Another objective of the project planning and management is that- it should be possible to get the best case and worst case outcomes of the project.
- There should be sufficient information discovery through the project so that reasonable project estimate can be made.

FEASIBILITY ANALYSIS:

When the client approaches the organization for getting the desired product developed, it comes up with a rough idea about what all functions of the software must perform and which all features are expected from the software.

Referencing to this information, the analysts do a detailed study about whether the desired system and its functionality are feasible to develop or not.

This feasibility study is focused towards goal of the organization. This study analyses whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints, and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity, and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

PROJECT AND PROCESS PLANNING:

Project planning is part of project management, in which project manager should recognize the future problems in advance and should be ready with the tentative solution to those problems. A project plan must be prepared in advance from the available information. The project planning is an iterative process and it gets completed only on the completion of the project. This process is iterative because new information gets available at each phase of the project development. Hence the plan needs to be modified on regular basis for accommodation new requirements of the project.

Project planning is inherently uncertain as it must be done before the project is actually started. Therefore the duration of the tasks is often estimated through a weighted average of optimistic, normal, and pessimistic cases.

The main purpose of this phase is to plan time, cost, and resources adequately to estimate the work needed and to effectively manage risk. Initial planning generally consists of:

- Developing the scope statement
- Selecting the planning team
- Identifying deliverables
- Creating the work breakdown structure
- Identifying the activities needed to complete those deliverables
- Sequencing the activities in a logical way
- Estimating the resources needed
- Estimating the time needed
- Estimating the costs
- Developing the schedule
- Developing the budget
- Gaining formal approval to begin

RESOURCE ALLOCATIONS:

Once the objectives of the project management are achieved, the project management is to estimate the resources for the project. Various recourses of the project are:

- Human or people
- Reusable software components
- Hardware or software components

The resources are available in limited quantity and stay in the organization as a pool of assets. The shortage of resources hampers development of the project and it can lag behind the schedule. Allocating extra resources increases development cost in the end. It is therefore necessary to estimate and allocate adequate resources for the project.

Resource management includes:

- Defining proper organization project by creating a project team and allocating responsibilities to each team member.
- Determining resources required at a particular stage and their availability.
- Manage Resources by generating resource request when they are required and de-allocating them when they are no more needed.

SOFTWARE EFFORTS:

Project Estimation

For an effective management, accurate estimation of various measures is a must. With the correct estimation, managers can manage and control the project more efficiently and effectively.

Project estimation may involve the following:

- **Software size estimation:** Software size may be estimated either in terms of KLOC (Kilo Line of Code) or by calculating number of function points in the software. Lines of code depend upon coding practices. Function points vary according to the user or software requirement.
- **Effort estimation:** The manager estimates efforts in terms of personnel requirement and man-hour required to produce the software. For effort estimation software size should be known. This can either be derived by manager's experience, historical data of organization, or software size can be converted into efforts by using some standard formula.
- **Time estimation:** Once size and efforts are estimated, the time required to produce the software can be estimated. An effort required is segregated into sub categories as per the requirement specifications and interdependency of various components of software. Software tasks are divided into smaller tasks, activities or events by Work Breakthrough Structure (WBS). The tasks are scheduled on day-to-day basis or in calendar months. The sum of time required to complete all tasks in hours or days is the total time invested to complete the project.
- **Cost estimation:** This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider –
    - Size of the software
    - Software quality
    - Hardware
    - Additional software or tools, licenses etc.
    - Skilled personnel with task-specific skills
    - Travel involved
    - Communication
    - Training and support

PROJECT SCHEDULING:

Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks and project milestones and then arrange them keeping various factors in mind. They look for tasks like in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. During the project scheduling the total work is separated into various small activities.

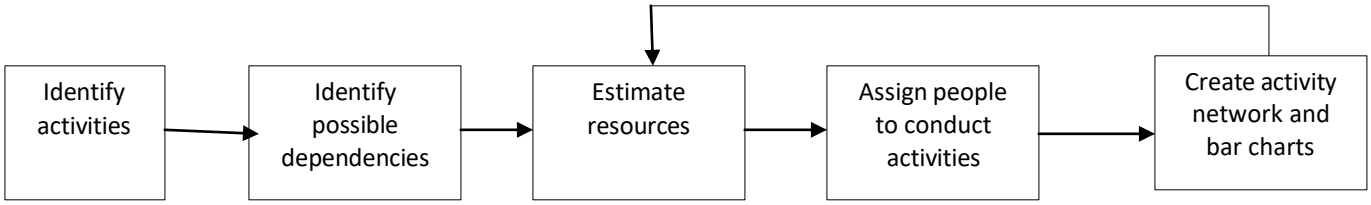


Figure 5.8: Project scheduling process

For scheduling a project, it is necessary to:

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them
- Estimate time frame required for each task
- Divide time into work-units
- Assign adequate number of work-units for each task
- Calculate total time required for the project from start to finish

COST ESTIMATIONS:

Cost estimation can be defined as the approximate judgments of the costs for project. Cost estimation is usually measured in terms of effort. The effort is the amount of time for one person to work for a certain period of time. COCOMO is one the most widely used software estimation models in the world. The Constructive Cost Model (COCOMO) is a procedural software cost estimation model .COCOMO is used to estimate size, effort and duration based on the cost of the software.

COCOMO predicts the effort and schedule for a software product development based on inputs relating to the size of the software and a number of cost drivers that affect productivity.

COCOMO has three different models that reflect the complexities:

- **Basic Model:** this model would be applied early in a projects development. It will provide a rough estimate early on that should be refined later on with one of the other models.
- **Intermediate Model:** this model would be used after you have more detailed requirements for a project.
- **Detailed Model:** when design of the project is complete you can apply this model to further refine your estimate.

Within each of these models there are also three different modes. The mode you choose will depend on your work environment, and the size and constraints of the project itself. The modes are:

- **Organic:** this mode is used for "relativity small software teams developing software in a highly familiar, in-house environment".
- **Embedded:** operating within tight constraints where the product is strongly tied to a "complex of hardware, software, regulations and operational procedures".
- **Semi-detached:** an intermediate stage somewhere in between organic and embedded. Projects are usually of moderate size of up to 300,000 lines of code.

Basic Model: The basic COCOMO model estimates the software development effort using only Lines Of Code (LOC). Various equations in this model are:

$$\text{Effort Applied (E)} = a_b(\text{KLOC})^{b_b} \text{ [man-months]}$$

$$\text{Development Time (D)} = c_b(\text{Effort Applied})^{d_b} \text{ [months]}$$

$$\text{People required (P)} = \text{Effort Applied} / \text{Development Time [count]}$$

Where, KLOC is the estimated number of delivered lines (expressed in thousands) of code for project. The coefficients a_b, b_b, c_b and d_b are given in the following table:

Software Projects	a_b	b_b	c_b	d_b

Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 5.2: List of Constants Based on Mode for Basic COCOMO

Intermediate Model: This is an extension of basic COCOMO model. This estimation model makes use of set of "cost driver attributes"to compute the cost of software.
The formula for effort calculation is:

$$E=a_i(KLOC)^{(b_i)}(EAF)$$

Where E is the effort applied in person-months, **KLOC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient $\mathbf{a_i}$ and the exponent $\mathbf{b_i}$ are given in the next table.

Software Projects	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

Table 5.3: List of Constants Based on Mode for Intermediate Model

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO.
Detailed Model: Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.
The detailed model uses different effort multipliers for each cost driver attribute. These Phase Sensitive effort multipliers are each to determine the amount of effort required to complete each phase. In detailed COCOMO, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.
The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software life cycle.

PROJECT SCHEDULING AND TRACKING:

Project schedule is the most important factor for software project manager. It is the duty of project manager to decide the project schedule and track the schedule.
Tracking the schedule means determine the tasks and milestone in the project as it proceeds.
Following are the various ways by which tracking of the project can be achieved:

- Conduct periodic meetings.
- Evaluate results of all the project reviews.
- Compare actual start date and scheduled start date of each of the project task.
- Determine if milestones of the project are achieved on scheduled date or not.
- Meet informally to the software practitioners.
- Assess the progress of the project quantitatively.

RISK ASSESSMENT AND MITIGATION:

Risk management involves all activities pertaining to identification, analyzing and making provision for predictable and non-predictable risks in the project. Risk may include the following:

- Experienced staff leaving the project and new staff coming in.
- Change in organizational management.
- Requirement change or misinterpreting requirement.
- Under-estimation of required time and resources.
- Technological changes, environmental changes, business competition.

Process of Risk Management:

Risk management performed in following stages:

1. **Risk identification:** In this phase all possible risks are anticipated and a list of potential risks are prepared.
2. **Risk analysis:** After risk identification, a list is prepared in which risks are prioritized.
3. **Risk planning:** The risk avoidance or risk minimization plan is prepared in this phase.
4. **Risk monitoring:** Identified risks must be mitigated. Hence risk mitigation plan must be prepared once the risks are discovered.

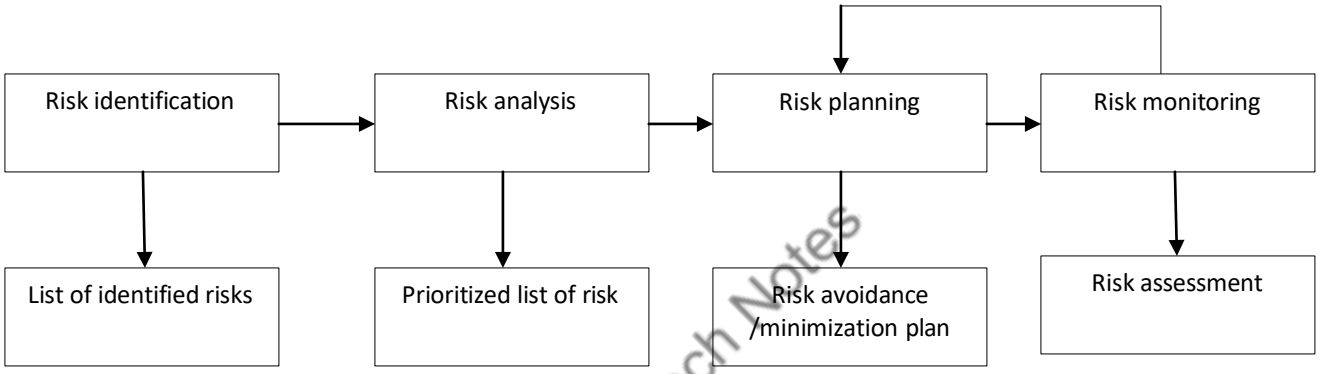


Figure 5.9: Risk management process

Risk Mitigation, Monitoring and Management (RMMM):

Risk analysis supports the project team in constructing a strategy to deal with risks.

There are three important issues considered in developing an effective strategy:

- **Risk avoidance or mitigation -** It is the primary strategy which is fulfilled through a plan.
- **Risk monitoring -** The project manager monitors the factors and gives an indication whether the risk is becoming more or less.
- **Risk management and planning -** It assumes that the mitigation effort failed and the risk is a reality.

RMMM Plan:

- It is a part of the software development plan or a separate document.
- The RMMM plan documents all work executed as a part of risk analysis and used by the project manager as a part of the overall project plan.
- The risk mitigation and monitoring starts after the project is started and the documentation of RMMM is completed.

SOFTWARE QUALITY ASSURANCE (SQA):

Software Quality:

In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance).

Quality Control:
Quality control (QC) is a procedure or set of procedures intended to ensure that a manufactured product or performed service adheres to a defined set of quality criteria or meets the requirements of the client or customer.

Quality Assurance:
It is planned and systematic pattern of activities necessary to provide a high degree of confidence in the quality of a product. It provides quality assessment of the quality control activities and determines the validity of the data or procedures for determining quality.

SQA Activities to Assure the Software Quality

The Software Quality Assurance of the software is analyzed and ensured by performing a series of activities. The activities are performed as step by step process and the result analysis is reported for the final evaluation process. The activities are performed as step by step process and the result analysis is reported for the final evaluation process.

A Quality Management Plan is prepared

- Application of Technical Methods (Employing proper methods and tools for developing software)
- Conduct of Formal Technical Review (FTR)
- Testing of Software
- Enforcement of Standards (Customer imposed standards or management imposed standards)
- Control of Change (Assess the need for change, document the change)
- Measurement (Software Metrics to measure the quality, quantifiable)
- Records Keeping and Recording (Documentation, reviewed, change control etc. i.e. benefits of docs).

PROJECT PLAN:
A project plan is a formal document designed to guide the control and execution of a project. A project plan is the key to a successful project and is the most important document that needs to be created when starting any business project.
A project plan is used for the following purposes:

- To document and communicate stakeholder products and project expectations
- To control schedule and delivery
- To calculate and manage associated risks

PROJECT METRICS:
Metrics is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
Project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. Project metrics are used by a project manager and a software team to adapt project work flow and technical activities.

Size Oriented Metrics:

- Size oriented measure is derived by considering the size of software that has been produced.
- The organization builds a simple record of size measure for the software projects. It is built on past experiences of organizations.
- It is a direct measure of software
- A simple set of size measure that can be developed is as given below:
    - Size= KLOC
    - Effort = Person/month

- Productivity=KLOC/ Person-month
- Quality= number of faults/KLOC
- Cost=$/KLOC
- Documentation= Pages of documentation/KLOC

Function Oriented Metrics:

- Use a measure of the functionality delivered by the application as a normalization value.
- Functionality cannot be measured directly; it must be derived indirectly using other direct measures.
- A measure called the function point.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.

How to calculate Function Point?

Domain Characteristics	Count		Weighting factor			Count
			Simple	Average	Complex	
Number of user input		x	3	4	6	
Number of user output		x	4	5	7	
Number of user inquiries		x	3	4	6	
Number of files		x	7	10	15	
Number of external interfaces		x	5	7	10	
Count Total						

Table 5.4: Function Point calculation table

Number of user inputs. Each user input that provides distinct application oriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.

Number of user outputs. Each user output that provides application oriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.

Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.

Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.

Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.

To compute function points (FP), the following relationship is used:

$$FP = \text{count total } [0.65 + 0.01\ \Sigma(Fi)] \text{ where count total is the sum of all FP entries .}$$

The Fi (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions:

1. Does the system require reliable backup and recovery?
2. Are data communications required?

3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?

Object Oriented Metrics:

Following are the set of metrics for object oriented projects:

1. **Weighted Methods per Class**
   Number of methods defined in class
   - Complex methods weigh more
   - Higher numbers = more faults
   - Classes are more specific = hard to reuse
   - Changes have more impact on subclasses
2. **Depth of Inheritance Tree**
   Number of super classes
   - Measures depth of hierarchy
   - Deep trees imply more complexity
   - Inheritance should reduce complexity not increase it
   - Deep trees promote reuse
   - Bugs are found in middle of tree
3. **Number of Children**
   Number of immediate subclasses
   - Measures breadth of hierarchy
   - Depth (DIT) implies reuse in a way breath (NOC) does not
   - Large numbers mean high reuse of base class; test it!
   - High NOC related to lower faults.
   - BUT, perhaps improper use of base class
4. **Coupling between Object Classes**
   Number of classes to which I am coupled
   - Coupling → We use each other's data or methods
   - High CBO is undesirable
   - Prevents reuse
   - Sensitivity to changes in others increases maintenance
   - > 14 is too high
5. **Response For a Class**
   Total number of methods executed
   - Large RFC = more faults
   - If most methods have a small RFC but one method has a
   - Hugh RFC

- you have a structured (not OO) application

6. **Lack of Cohesion of Methods**
   How well the methods of a class are are related to each other?
   - A cohesive class performs one function
   - Highly cohesive classes promotes encapsulation
   - but have highly coupled methods
   - Low cohesion = more errors