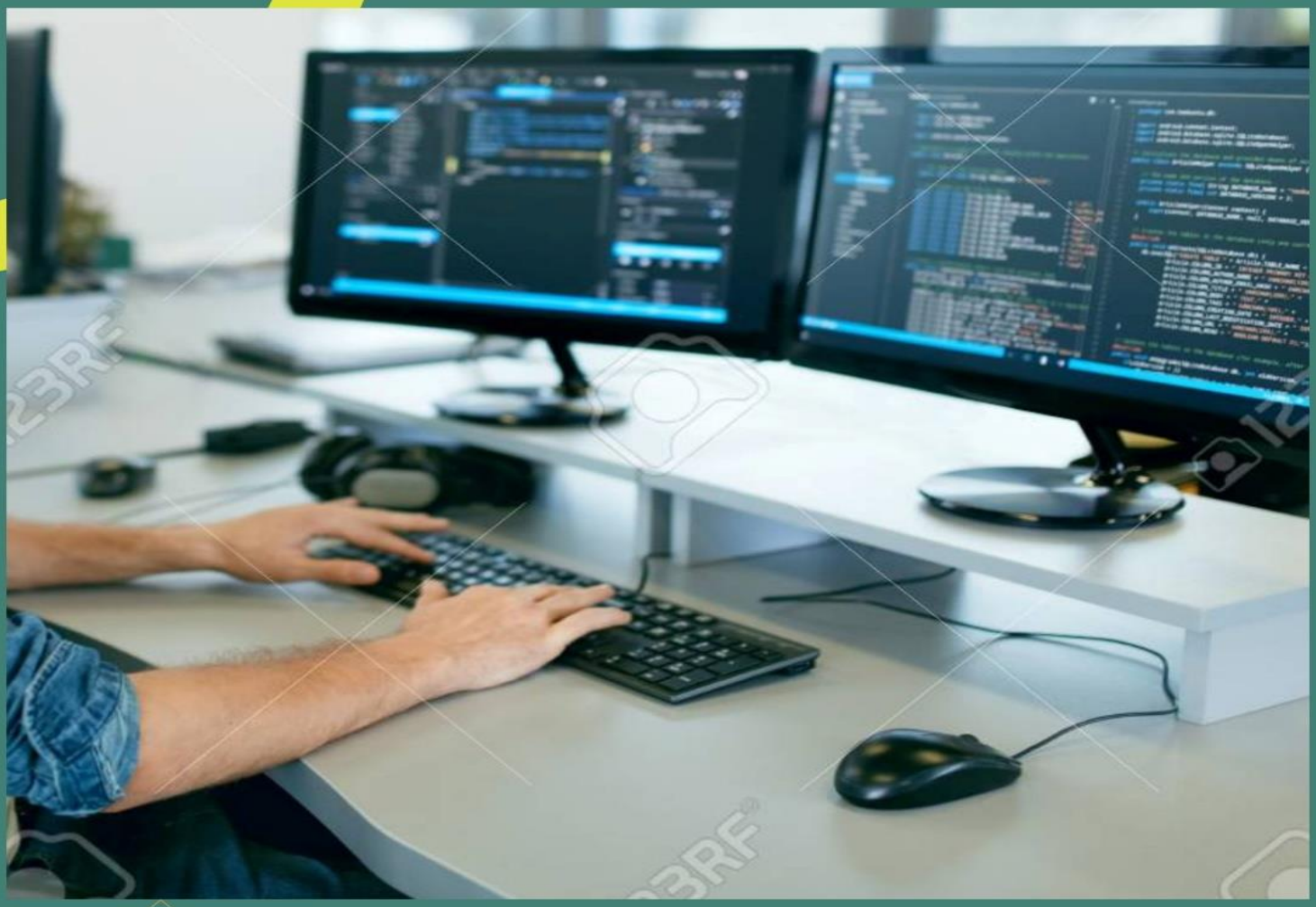


SEMESTER
III

SOFTWARE ENGINEERING

Bachelor of Computer Applications

P V V Durga PraSad
Department of Computer science

Software Engineering

UNIT I

Introduction to Software Engineering: Definitions - Size Factors - Quality and Productivity Factors –Managerial Issues.

Planning a software project: Defining the problem - Developing a Solution Strategy - Planning the Development Process - Planning an Organization structure - Other Planning Activities.

UNIT – II

Software Cost Estimation: Software cost factors - Software Cost.

Estimation Techniques – Staffing level Estimation- Estimating Software Maintenance Costs – The Software Requirements, Specification - Formal Specification Techniques - Languages and Processors for Requirements Specification.

UNIT – III

Software design: Fundamental Design Concepts - Modules and Modularization Criteria – Design Notations -Design Techniques - Detailed Design Considerations. Real-Time and Distributed System Design - Test Plans - Milestones, walkthroughs, and Inspections.

UNIT IV

User interface design and real time systems: User interface design - Human factors – Human computer interaction - Human - Computer Interface design - Interface design - Interface standards.

UNIT V

Software quality and testing: Software Quality Assurance - Quality metrics - Software Reliability - Software testing - Path testing – Control Structures testing - Black Box testing - Integration, Validation and system testing - Reverse Engineering and Reengineering.

CASE Tools: Projects management, tools - analysis and design tools – programming tools -

integration and testing tool - Case studies.

UNIT-1

❖ Software Engineering

Software engineering is a discipline within the field of computer science that focuses on the systematic design, development, testing, and maintenance of software. It involves applying engineering principles to software creation, ensuring that the software is reliable, efficient, and meets user requirements.

Here's an overview of some key concepts and areas within software engineering:

Key Concepts

1. **Software Development Life Cycle (SDLC):**

* Requirement Analysis: Understanding and documenting what the users need.

* Design: Planning the architecture and design of the software.

* Implementation (Coding): Writing the actual code.

* Testing: Verifying that the software works as intended.

* Deployment: Releasing the software to users.

* Maintenance: Ongoing support and improvement of the software.

2. Software Development Methodologies:

* Waterfall: A linear and sequential approach where each phase depends on the deliverables of the previous one.

* Agile: An iterative and incremental approach that promotes flexibility and customer feedback.

* Scrum: A specific Agile framework focusing on small, crossfunctional teams and timeboxed iterations called sprints.

* DevOps: A methodology that integrates development and operations to improve collaboration and productivity.

3. Programming Languages:

* Common languages include Python, Java, C++, JavaScript, and C#.

* The choice of language often depends on the project requirements and the development environment.

4. Software Design Patterns:

* Reusable solutions to common problems in software design.

* Examples include Singleton, Factory, Observer, and Strategy patterns.

5. Version Control:

* Systems like Git help manage changes to the source code over time.

* Enables collaboration among multiple developers and tracks the history of changes.

6. Quality Assurance:

* Ensuring that the software meets quality standards.

* Includes various testing methods like unit testing, integration testing, system testing, and acceptance testing.

7. Project Management:

* Involves planning, organizing, and managing resources to bring about the successful completion of specific project goals.

* Tools like JIRA, Trello, and Asana help manage software projects.

Importance of Software Engineering

* Quality Assurance: Ensures the software is reliable, efficient, and meets user needs.

* Cost Efficiency: Reduces the cost of development and maintenance by following a systematic approach.

* Project Management: Helps manage complex projects with clear goals, timelines, and deliverables.

* User Satisfaction: Involves users in the development process to ensure the final product meets their needs.

* Adaptability: Allows for better handling of changing requirements and technology advancements.

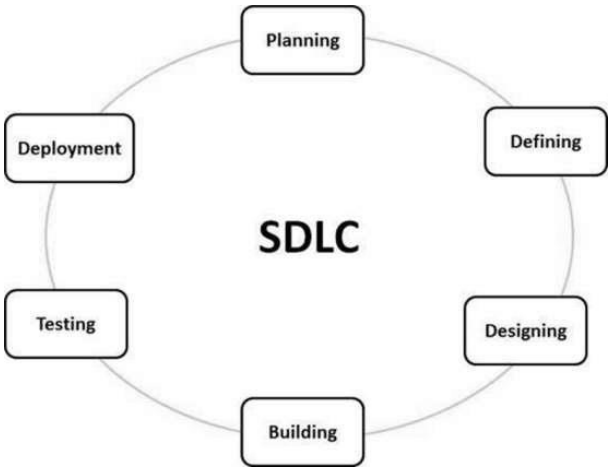
Software engineering is a continually evolving field, driven by advances in technology, changing user needs, and new methodologies. It plays a critical role in the development of software systems that power modern society, from simple applications to complex systems in healthcare, finance, transportation, and beyond.

❖ Software Development Life

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

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

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis

Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the

sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements

Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture

SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product

In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product

This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance

Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

❖ Software size factors

Software size estimation is a crucial aspect of the software engineering process.

Size estimation in software engineering involves predicting the size of a software project in terms of lines of code, function points, or other metrics.

Accurate size estimation is essential for effective project planning, resource allocation, and control.

The importance of size estimation in software engineering

Size estimation in software engineering provides a foundation for determining project timelines, effort required, and resource allocation.

Size estimation allows project managers to make informed decisions regarding project scope, budget, and staffing. It also helps in setting realistic expectations for stakeholders and ensuring project feasibility.

Size estimation in software engineering is not just about measuring the lines of code or the number of features.

It involves a comprehensive analysis of the functional and nonfunctional requirements, identifying the intricate features and components of the software project.

Defining software size estimation

Size estimation in software engineering refers to the process of quantifying the size of a software project.

Size estimation can be done using various techniques, including algorithmic methods, expert judgement, and machine learning approaches.

The role of size estimation in project planning

Accurate size estimation in software engineering enables project managers to create realistic schedules and allocate resources effectively in project planning.

By estimating the size of the software project, stakeholders can assess the level of effort required and make informed decisions regarding budget and staffing.

It also helps in identifying potential risks and challenges that may arise during the development process.

One of the key benefits of size estimation in project planning is the ability to set realistic expectations for stakeholders.

By clearly understanding the project's size, project managers can manage expectations and avoid misunderstandings or disappointments later.

Different methods of size estimation in software engineering

Several methods are available for software size estimation, each with strengths and limitations.

These methods can be broadly classified into algorithmic, expert judgement, and machine learning approaches.

Algorithmic methods for size estimation

Algorithmic methods use mathematical models to estimate the size of a software project.

These models are based on historical data and key project parameters such as complexity, functionality, and technology.

Standard algorithmic methods include Function Point Analysis and COCOMO (COnstructive COst MOdel).

Expert judgement in size estimation

Expert judgement relies on the knowledge and experience of software professionals to estimate project size.

It involves gathering input from domain experts, project managers, and developers to assess the complexity and size of the software project.

Machine learning approaches to size estimation

Machine learning techniques have gained popularity in recent years for software size estimation.

These approaches analyse historical data to identify patterns and develop prediction models.

Machine learning algorithms can provide accurate size estimates based on specific project attributes by training the models on past projects.

❖ Quality and productivity factors

Quality and productivity factors are critical concepts in various fields, including manufacturing, service industries, software development, and project management. Understanding these factors helps organizations optimize their processes, improve outputs, and achieve strategic goals. Here's a detailed look at these factors:

Quality Factors

Quality factors are attributes or characteristics that affect the standard of products or services. High quality ensures customer satisfaction, reduces costs, and enhances competitiveness. Key quality factors include:

1. Customer Satisfaction:

Meeting or exceeding customer expectations is crucial. Feedback, surveys, and reviews can help measure satisfaction levels.

2. Consistency:

Consistent product or service quality maintains reliability and builds customer trust. Standard operating procedures (SOPs) and quality control systems are essential.

3. Defect Rates:

Low defect rates indicate high quality. Statistical process control (SPC) and Six Sigma methodologies help in reducing defects.

4. Compliance with Standards:

Adhering to industry standards and regulations ensures quality and legal compliance. ISO 9001 is a common quality management standard.

5. Durability and Reliability:

Longlasting and reliable products enhance customer loyalty and reduce warranty costs. Reliability engineering and testing are vital.

6. Performance and Functionality:

Products or services should perform as expected and fulfill their intended functions. This is often verified through rigorous testing and quality assurance (QA).

7. Continuous Improvement:

Implementing methodologies like Total Quality Management (TQM) and Lean ensures ongoing improvements in quality.

Productivity Factors

Productivity factors influence the efficiency and effectiveness of production or service delivery. High productivity leads to better resource utilization, cost savings, and higher outputs. Key productivity factors include:

1. Workforce Efficiency:

Skilled, motivated, and welltrained employees enhance productivity. Providing continuous training and incentives boosts efficiency.

2. Technology and Automation:

Using advanced technology and automation reduces manual effort, increases speed, and minimizes errors.

3. Process Optimization:

Streamlining processes and eliminating waste using Lean, Six Sigma, or Kaizen improves productivity.

4. Resource Utilization:

Efficient use of materials, time, and equipment ensures higher productivity. This includes minimizing downtime and optimizing inventory levels.

5. Workflow Management:

Effective scheduling, task allocation, and workflow management ensure smooth operations and reduce bottlenecks.

6. Innovation:

Encouraging innovation leads to new methods, products, or services that can significantly boost productivity.

7. Measurement and Analysis:

Tracking key performance indicators (KPIs) and using data analytics helps in identifying areas for improvement.

Relationship Between Quality and Productivity

1. Quality Impact on Productivity:

Highquality processes reduce rework, errors, and defects, leading to better productivity. Conversely, poor quality increases costs and time, reducing productivity.

2. Productivity Impact on Quality:

Efficient processes and resource utilization can enhance quality by allowing more focus on quality control and continuous improvement initiatives.

3. Balancing Quality and Productivity:

Organizations must find the right balance to avoid compromising one for the other. For instance, overly focusing on speed (productivity) might reduce quality, while excessive quality checks might slow down processes.

❖ Software Project Management Complexities

Software project management complexities refer to the various challenges and difficulties involved in managing software development projects. The primary goal of software project management is to guide a team of developers to complete a project successfully within a given timeframe. However, this task is quite challenging due to several factors. Many projects have failed in the past due to poor project management practices. Software projects are often more complex to manage than other types of projects

Software Project Management Complexities refer to the various difficulties to manage a software project. It recognizes in many different ways. The main goal of software project management is to enable a group of developers to work effectively toward the successful completion of a project in a given time. But software project management is a very difficult task.

Types of Complexity

The following are the types of complexity in software project management:

Time Management Complexity: Complexities to estimate the duration of the project. It also includes the complexities to make the schedule for different activities and timely completion of the project.

Cost Management Complexity: Estimating the total cost of the project is a very difficult task and another thing is to keep an eye that the project does not overrun the budget.

Quality Management Complexity: The quality of the project must satisfy the customer's requirements. It must assure that the requirements of the customer are fulfilled.

Risk Management Complexity: Risks are the unanticipated things that may occur during any phase of the project. Various difficulties may occur to identify these risks and make amendment plans to reduce the effects of these risks.

Human Resources Management Complexity: It includes all the difficulties regarding organizing, managing, and leading the project team.

Communication Management Complexity: All the members must interact with all the other members and there must be good communication with the customer.

Infrastructure complexity: Computing infrastructure refers to all of the operations performed on the devices that execute our code. Networking, load balancers, queues, firewalls, security, monitoring, databases, shading, etc. We are solely interested in dealing with data, processing business policy rules, and clients since we are software engineers that are committed to providing value in a continuous stream. The aforementioned infrastructure ideas are nothing more than irksome minutiae that don't offer any benefit to the clients. Since it is a necessary evil, we view infrastructure as accidental complexity. Our policies for scaling, monitoring, and other issues are of little interest to our paying clients.

Deployment complexity: A release candidate, or finalized code, has to be synchronized from one system to another. Conceptually, such an operation ought to be simple. To perform this synchronization swiftly and securely in practice proves to be difficult.

API complexity: An API should ideally not be any more difficult to use than calling a function. However, that hardly ever occurs. These calls are inadvertently complicated due to authentication, rate restrictions, retries, mistakes, and other factors.

Procurement Management Complexity: Projects need many services from third parties to complete the task. These may increase the complexity of the project to acquire the services.

Integration Management Complexity: The difficulties regarding coordinating processes and developing a proper project plan. Many changes may occur during the project development and it may hamper the project completion, which increases the complexity.

Factors that Make Project Management Complex

Give Below are factors that make project management complex

Changing Requirements: Software projects often involve complex requirements that can change throughout the development process. Managing these changes can be a significant challenge for project managers, who must ensure that the project remains on track despite the changes.

Resource Constraints: Software projects often require a large amount of resources, including software developers, designers, and testers. Managing these resources effectively can be a major challenge, especially when there are constraints on the availability of skilled personnel or budgets.

Technical Challenges: Software projects can be complex and difficult due to the technical challenges involved. This can include complex algorithms, database design, and system integration, which can be difficult to manage and test effectively.

Schedule Constraints: Software projects are often subject to tight schedules and deadlines, which can make it difficult to manage the project effectively and ensure that all tasks are completed on time.

Quality Assurance: Ensuring that software meets the required quality standards is a critical aspect of software project management. This can be a complex and timeconsuming process, especially when dealing with large, complex systems.

Stakeholder Management: Software projects often involve multiple stakeholders, including customers, users, and executives. Managing these stakeholders effectively can be a major challenge, especially when there are conflicting requirements or expectations.

Risk Management: Software projects are subject to a variety of risks, including technical, schedule, and resource risks. Managing these risks effectively can be a complex and timeconsuming process, and requires a structured approach to risk management.

Advantages of Software Project Management Complexity

Improved software quality: Software engineering practices can help ensure the development of highquality software that meets user requirements and is reliable, secure, and scalable.

Better risk management: Project management practices such as risk management can help identify and address potential risks, reducing the likelihood of project failure.

Improved collaboration: Effective communication and collaboration among team members can lead to better software development outcomes, higher productivity, and better morale.

Flexibility and adaptability: Software development projects require flexibility to adapt to changing requirements, and software engineering practices provide a framework for managing these changes.

Increased efficiency: Software engineering practices can help streamline the development process, reducing the time and resources required to complete a project.

Improved customer satisfaction: By ensuring that software meets user requirements and is delivered on time and within budget, software engineering practices can help improve customer satisfaction.

Better maintenance and support: Software engineering practices can help ensure that software is designed to be maintainable and supportable, making it easier to fix bugs, add new features, and provide ongoing support to users.

Increased scalability: By designing software with scalability in mind, software engineering practices can help ensure that software can handle growing user bases and increasing demands over time.

Higher quality documentation: Software engineering practices typically require thorough documentation throughout the development process, which can help ensure that software is welldocumented and easier to maintain over time.

Disadvantages of Software Project Management Complexity

Increased complexity: The dynamic nature of software development and the changing requirements can make software engineering and project management more complex and challenging.

Cost overruns: Software development projects can be expensive, and managing them effectively requires careful budget planning and monitoring to avoid cost overruns.

Schedule delays: Technical challenges, scope creep, and other factors can cause schedule delays, which can impact the project's success and increase costs.

Difficulty in accurately estimating time and resources: The complexity of software development and the changing requirements can make it difficult to accurately estimate the time and resources required for a project.

· **Dependency on technology:** Software development projects heavily rely on technology, which can be a doubleedged sword. While technology can enable efficient and effective development, it can also create dependencies and vulnerabilities that can negatively impact the project.

❖ Defining the problem

The first two steps assist the team in understanding the problem, the most crucial first step towards getting a solution. Person responsible for gathering requirement, defining the problem and designing the system is called **system analyst**.

Identifying the problem

1. *Needs of the client*

- A need is an instance in which a neccisity or want exists
- Solution needs to meet the requirements of these needs

2. **Functionality Requirements**

- Describe what the system will do & what the solution needs to achieve
- The requirements of the system give direction to the project
- Requirements are defined as features, properties and behaviours a system nust have to achieve its purpose

3. Compapatibility Issues

- Software of various types runs on a variety of enviroments
- When designing software developers must ensure products are able to be used on multiple devices and conditions

Examples include:

- Problems with different operating systems versions
- Browsers that do not implement HTML standards
- Hardware not supporting software

4. Performance Issues

- Testing and real world applications are very different
- Testing must be extremely rigourous and broad
- Common Issues include:

Appearaning to be not repsonding when time takes too long and Poor Response times in networking operations

Boundaries of the problem

- Boundaries of the problem define the limits of the problem of the system to be developed
- Anything outside the system is said to be part of the enviroment
- The system connects with the enviroment through an interface
- Determing the boundaries effectively determines the system and how the enivroment interacts with it

❖ Software development strategy

Software development strategy refers to the comprehensive plan and approach that guides how software products are conceptualized, developed, and delivered. A welldefined strategy ensures that software projects are executed efficiently, meet stakeholder expectations, and achieve business goals.

Here are key aspects typically involved in crafting a software development strategy:

1. **Define Clear Goals and Objectives**: Begin by understanding the business objectives that the software aims to achieve. This could be improving operational

efficiency, enhancing customer experience, or entering new markets. Clear goals provide direction and help prioritize development efforts.

2. **Understand User Needs**: Conduct thorough research to understand the needs, preferences, and pain points of your target users. This could involve user interviews, surveys, and usability testing. Aligning software features with user needs enhances adoption and satisfaction.

3. **Choose the Right Development Methodology**: Select a development methodology that suits the project's requirements and team dynamics. Common methodologies include Agile (Scrum, Kanban), Waterfall, and DevOps. Agile methodologies are popular for their flexibility and iterative approach, while Waterfall may be suitable for projects with welldefined requirements upfront.

4. **Plan and Manage Requirements**: Establish a process for gathering, documenting, and managing requirements throughout the development lifecycle. Use techniques such as user stories, use cases, and acceptance criteria to ensure a clear understanding between stakeholders and the development team.

5. **Technology Selection**: Choose appropriate technologies, frameworks, and tools based on project requirements, scalability needs, and team expertise. Consider factors such as performance, security, integration capabilities, and longterm maintenance.

6. **Team Composition and Collaboration**: Define roles and responsibilities within the development team. Foster a collaborative environment where team members can communicate effectively, share knowledge, and work towards common goals.

7. **Continuous Integration and Delivery (CI/CD)**: Implement CI/CD pipelines to automate and streamline the process of building, testing, and deploying software updates. This improves efficiency, reduces errors, and enables faster release cycles.

8. **Quality Assurance and Testing**: Establish a comprehensive testing strategy that includes unit testing, integration testing, system testing, and user acceptance testing (UAT). Implement testing early and often to identify defects early in the development process.

9. **Deployment and Release Management**: Plan for efficient deployment and release management. Consider factors such as deployment environments, rollback procedures, and postrelease monitoring to ensure smooth transitions and minimize disruptions.

10. **Monitor, Evaluate, and Iterate**: Implement monitoring and analytics tools to gather data on software performance, usage patterns, and user feedback. Use this data to evaluate the effectiveness of features and iterate on the software to continuously improve its quality and value.

11. **Security and Compliance**: Integrate security measures throughout the development process to protect against vulnerabilities and ensure compliance with

industry regulations and standards. Conduct regular security audits and implement best practices for data protection.

By integrating these elements into your software development strategy, you can effectively manage the development lifecycle, deliver highquality software products, and achieve business objectives while adapting to changing requirements and market conditions.

Planning the development process

Planning the development process is a critical aspect of software project management. It involves organizing tasks, allocating resources, setting timelines, and defining processes to ensure the project progresses smoothly and achieves its goals. Here's a detailed guide on how to effectively plan the development process:

1. Task Identification and Breakdown

Identify Project Phases: Divide the project into distinct phases (e.g., requirements gathering, design, development, testing, deployment).

Task Identification: List all tasks required to complete each phase. Tasks should be specific and measurable.

Task Breakdown: Break down complex tasks into smaller, manageable subtasks. This makes it easier to estimate time and effort accurately.

2. Estimation and Scheduling

Time Estimation: Estimate the time required for each task or subtask. Use historical data, expert judgment, and estimation techniques (e.g., PERT, threepoint estimation).

Resource Estimation: Determine the resources (e.g., developers, testers, hardware, software) needed for each task.

Dependency Identification: Identify dependencies between tasks. Determine which tasks can be executed concurrently and which are sequential.

3. Creating a Project Timeline

Define Milestones: Set key milestones for significant achievements (e.g., completion of design phase, beta release, final deployment).

Develop a Gantt Chart: Use a Gantt chart or similar tool to visualize the project timeline. Include tasks, dependencies, milestones, and deadlines.

Iterative Planning: For agile projects, create iteration plans (sprints) with specific goals and tasks for each iteration.

4. Resource Allocation

Assign Responsibilities: Assign tasks to team members based on their skills and expertise.

Allocate Resources: Ensure that necessary resources (e.g., software licenses, development tools) are available when needed.

Manage Workload: Monitor workload distribution to avoid overburdening team members.

5. Risk Management

Identify Risks: Identify potential risks that could affect project timeline, scope, or quality.

Risk Analysis: Assess the likelihood and impact of each risk. Prioritize risks based on their severity.

Risk Mitigation: Develop strategies to mitigate or manage identified risks. Include contingency plans for highimpact risks.

6. Quality Assurance and Testing

Plan Testing Activities: Define the testing strategy and types of testing (e.g., unit testing, integration testing, acceptance testing).

Allocate Time for Testing: Ensure adequate time is allocated for testing activities within the project timeline.

Quality Control: Implement processes to monitor and control the quality of deliverables throughout the development process.

7. Documentation and Communication

Documentation Plan: Plan for documentation of code, design documents, user manuals, and other project artifacts.

Communication Plan: Establish communication channels and protocols for team collaboration, status updates, and stakeholder communication.

Regular Updates: Schedule regular meetings (e.g., daily standups, weekly status meetings) to review progress, address issues, and adjust plans as needed.

8. Monitoring and Adaptation

Monitor Progress: Continuously monitor project progress against the planned timeline and milestones.

Track Metrics: Use metrics (e.g., burndown charts, velocity) to measure progress and identify areas for improvement.

Adaptation: Be prepared to adapt the development plan based on feedback, changes in requirements, or unexpected challenges.

9. Continuous Improvement

PostMortem Analysis: Conduct a postmortem or retrospective at the end of the project to evaluate successes, challenges, and lessons learned.

Implement Feedback: Incorporate feedback and lessons learned into future projects to improve development processes and outcomes.

By carefully planning the development process and actively managing its execution, software projects can minimize risks, optimize resource utilization, and increase the likelihood of delivering a successful product that meets stakeholder expectations.

❖ Planning an organizational structure

Planning an organizational structure for software development involves designing how teams and roles are organized to achieve efficient collaboration, clear communication, and effective project delivery. Here's a structured approach to planning an organizational structure for software development:

1. Understand Project Scope and Goals

Define Project Scope: Clearly define the scope of the software development project, including goals, timelines, budget, and expected outcomes.

Identify Team Requirements: Determine the skill sets and expertise required to successfully execute the project. Consider technical skills, domain knowledge, and experience levels needed.

2. Choose an Organizational Model

Functional Structure: Organize teams based on functional areas such as development, testing, design, and support.

ProjectBased Structure: Form teams around specific projects or products, with crossfunctional team members collaborating on a temporary basis.

Matrix Structure: Combine elements of functional and projectbased structures, where team members report both to functional managers and project managers.

3. Define Roles and Responsibilities

Identify Key Roles: Define essential roles such as developers, testers, designers, project managers, product owners, and architects.

Clarify Responsibilities: Clearly outline the responsibilities and expectations for each role. Ensure roles are welldefined to avoid confusion and promote accountability.

4. Establish Team Composition

Team Size: Determine the optimal team size based on project complexity, workload, and required expertise.

CrossFunctional Teams: Encourage crossfunctional teams where members bring diverse skills and perspectives to the project.

Specialization vs. Generalization: Balance specialization (deep expertise in specific areas) with generalization (broad skills across multiple areas) based on project needs.

5. Promote Communication and Collaboration

Communication Channels: Establish clear communication channels within and between teams (e.g., Slack, Microsoft Teams, regular meetings).

Collaboration Tools: Use collaborative tools for project management, version control, documentation, and knowledge sharing (e.g., Jira, Git, Confluence).

Team Dynamics: Foster a collaborative culture where team members feel empowered to share ideas, provide feedback, and solve problems together.

6. Define DecisionMaking Processes

Authority Levels: Clarify decisionmaking authority levels for different roles and teams. Determine who has the final say on technical decisions, scope changes, and resource allocation.

DecisionMaking Framework: Establish a framework for making decisions, considering factors such as impact on project timeline, budget, and alignment with strategic goals.

7. Plan for Growth and Scalability

Scalability: Design the organizational structure to accommodate growth, both in terms of team size and project complexity.

Flexibility: Maintain flexibility to adapt the organizational structure as project requirements evolve or new opportunities arise.

Feedback Mechanisms: Implement mechanisms for gathering feedback from team members and stakeholders to continuously improve the organizational structure and processes.

8. Support Career Development

Training and Development: Provide opportunities for skills development, training, and certifications to enhance team members' expertise and career growth.

Mentorship: Establish mentorship programs to support knowledge transfer and professional growth within the team.

Recognition and Rewards: Recognize and reward team members for their contributions and achievements, fostering motivation and retention.

9. Ensure Alignment with Company Culture and Values

Culture Alignment: Align the organizational structure with the company's values, mission, and culture. Ensure that the structure promotes inclusivity, transparency, and accountability.

Leadership Support: Gain leadership buyin and support for the chosen organizational structure to ensure alignment with strategic objectives and longterm goals.

10. Monitor and Iterate

Performance Metrics: Establish key performance indicators (KPIs) to monitor team performance, project progress, and organizational effectiveness.

Continuous Improvement: Regularly evaluate the organizational structure, processes, and team dynamics. Implement improvements based on lessons learned and feedback to optimize performance and achieve better outcomes.

Example Application: Software Development Company

For a software development company, a matrix structure might be suitable, where teams are organized by functional expertise (e.g., development, testing, design) and

also by project or product. This structure allows for specialization within functions while promoting crossfunctional collaboration and alignment with project goals.

In conclusion, effective organizational structure planning involves aligning with strategic objectives, understanding current capabilities, promoting communication and collaboration, and maintaining flexibility to adapt to changes. By following a structured approach and focusing on continuous improvement, organizations can design and implement a structure that supports longterm success and growth.

❖ Other Planning Activities

Here are some more planning activities that organizations can undertake to improve their efficiency, sustainability, and overall performance:

1. Knowledge Management

Knowledge Sharing: Implement systems and processes to capture, organize, and share knowledge across the organization.

Knowledge Transfer: Develop strategies to transfer critical knowledge from retiring employees or departing team members to successors.

Learning Organization: Promote a learning culture where continuous learning and knowledge acquisition are encouraged and supported.

2. Ethics and Corporate Governance

Ethics Framework: Develop and enforce ethical standards and guidelines that govern the behavior and decisionmaking of employees and leadership.

Corporate Governance Practices: Implement best practices for corporate governance to ensure transparency, accountability, and ethical behavior at all levels.

3. Health and Wellness Programs

Employee Wellness: Design and implement wellness programs that promote physical, mental, and emotional wellbeing among employees.

Healthcare Benefits: Review and optimize healthcare benefits to ensure they meet the needs of employees and contribute to a healthy workforce.

4. Diversity, Equity, and Inclusion (DEI) Initiatives

DEI Strategy: Develop a strategy to foster diversity, equity, and inclusion within the organization, promoting a respectful and inclusive workplace culture.

Training and Awareness: Provide training and awareness programs to educate employees on DEI issues and promote understanding and acceptance.

5. Customer Retention and Loyalty Programs

Customer Relationship Management (CRM): Implement CRM strategies and tools to enhance customer interactions, retention, and satisfaction.

Loyalty Programs: Design and manage customer loyalty programs to reward repeat customers and encourage brand loyalty.

6. Disaster Recovery and Business Continuity

Disaster Recovery Plan: Develop and maintain a comprehensive plan to recover IT systems and business operations in the event of a disaster or disruption.

Business Continuity Plan (BCP): Establish protocols and procedures to ensure critical business functions can continue during and after a crisis or emergency.

7. Data Privacy and Cybersecurity

Data Protection: Implement policies, procedures, and technologies to safeguard sensitive data and protect against cybersecurity threats.

Compliance: Ensure compliance with data privacy regulations (e.g., GDPR, CCPA) and industry standards to mitigate legal and reputational risks.

8. Employee Engagement and Satisfaction

Employee Surveys: Conduct regular surveys to assess employee satisfaction, engagement levels, and identify areas for improvement.

Recognition Programs: Implement employee recognition and reward programs to acknowledge achievements and contributions.

9. Philanthropy and Community Engagement

Corporate Philanthropy: Develop a philanthropic strategy that aligns with organizational values and priorities, supporting charitable initiatives and community causes.

Volunteer Programs: Encourage employee participation in volunteer activities and community service projects to contribute positively to society.

These planning activities can help organizations enhance their operational efficiency, strengthen stakeholder relationships, foster a positive work environment, and contribute to sustainable growth and success in the long term. Each activity should be tailored to fit the organization's unique needs, industry requirements, and strategic objectives.

UNIT - 2

INTRODUCTION TO SOFTWARE COST ESTIMATION

Estimating the cost of software product is one of the most difficult and error-prone tasks in software engineering. It is difficult to make an accurate cost estimate during the planning phase of software development.

A preliminary estimate is prepared during the planning phase and presented at the project feasibility review. An improved estimate is presented at the software requirements review, and the final estimate is presented at the preliminary design review. Each estimate is a refinement of the previous one, and is based on the additional information gained as a result of additional work activities.

❖ Software Cost Factors

The factors that influence the cost of a software product are Programmer Ability, Product Complexity, Product Size, Available Time, Required Reliability, Level of Technology. Primary among the cost factors are the individual abilities of project personnel and their familiarity with the application area; the complexity of the product; the size of the product, the available time, the required level of reliability; the level of technology utilized, and the availability, familiarity, and stability of the system used to develop the product.

Programmer Ability

A well-known experiment conducted in 1968 by Harold Sackman and colleagues. It determines the relative influence of batch and time-shared access on programmer productivity. Twelve experienced programmers were each given two programming problems to solve, some using batch facilities and some using time-sharing.

The differences between best and worst performance were facto9rs of 6 to 1 in program size, 8 to 1 in execution time, 9 to 1 in development time, 18 to 1 in coding time, and 28 to 1 in debugging time. On very large projects, the differences in individual programmer ability will tend to average out, but on projects utilizing five or fewer programmers, individual differences in ability can be significant.

Product Complexity

There are three categories of software product: **Application Programs**, which include data processing and scientific programs; **Utility Programs**, such as compilers, linkage editors, and inventory systems; and **System Programs**, such as database management systems, operating systems, and real-time systems.

Brooks states that utility programs are three times as difficult to write as application

programs, and that system programs are three times as difficult to write as utility programs. His levels of product complexity are thus 1-3-9 for applications-utility-systems programs.

Boehm uses three levels of product complexity and provides equations to predict total programmer- months of effort, PM, in terms of the number of thousands of delivered source instruction, KDSI, in the product. Programmer cost for a software project can be obtained by multiplying the effort in programmer-months by the cost per programmer-month. The equations were derived by examining historical data from a large number of actual projects.

In Boehm's terminology, the three levels of product complexity are organic, semidetached, and embedded programs.

Application programs: PM = 2.4*(KDSI)**1.05

Utility programs: PM = 3.0*(KDSI)**1.12

Systems programs: PM = 3.6*(KDSI)**1.20

development time for a program is

Application programs: TDEV = 2.5*(PM)**0.38

Utility programs: TDEV = 2.5*(PM)**0.35

Systems programs: TDEV = 2.5*(PM)**0.32

Product Size

A large software product is more expensive to develop than a small one. Boehm's equations indicate that the rate of increase in required effort grows with the number of source instructions at an exponential rate slightly greater than 1. Some investigators believe that the rate of increase in effort grows at an exponential rate slightly less than 1, but most use an exponent in the range of 1.05 to 1.83.

Available Time

Total project effort is sensitive to the calendar time available for project completion. Several investigators agree that software projects require more total effort if development time is compressed or expanded from the optimal time. The most striking feature is the Putnam curve. According to Putnam, project effort is inversely proportional to the fourth power of development time, E = k/(Td**4). This curve indicates an extreme penalty for schedule compression and an extreme reward for expanding the project schedule.

Putnam also states that the development schedule cannot be compressed below about 86% of the nominal schedule, regardless of the people or resources utilized.

Required Level of Reliability

Software reliability can be defined as the probability that a program will perform a required function under stated conditions for a stated period of time. Reliability can be expressed in terms of accuracy, robustness, completeness, and consistency of the source code. Reliability characteristics can be built into a software product, but there is a cost associated with the increased level of analysis, design, implementation, and verification and validation effort that must be exerted to ensure high reliability. The multipliers range from 0.75 for very low reliability to 1.4 for very high reliability. The effort ratio is thus 1.87 (1.4/0.75).

Level of Technology

The level of technology in a software development project is reflected by the programming language, the abstract machine (hardware plus software), the programming practices, and the software tools used. It is well known that the number of source instructions written per day is largely independent of the language used, and that program statements written in high-level languages such as FORTRAN and Pascal expand into several machine-level statements. Use of high-level language instead of assemble language thus increases programmer productivity by a factor of 5 to 10.

The type-checking rules and self-documenting aspects of high-level languages improve the reliability and modifiability. Ada provide additional features to improve programmer productivity and software reliability. These features include strong type-checking, data abstraction, separate compilation, exception handling, interrupt handling, and concurrency mechanisms.

Modern programming practices include use of systematic analysis and design techniques, structured design notations, walkthroughs and inspections, structured coding, systematic testing, and a program development library.

Software tools range from elementary tools, such as assemblers and basic debugging aids, to compilers and linkage editors, to interactive text editors and database management system,s to program design language processors and requirements specification analyzers, to fully integrated development environments that include configuration management and automated verification tools.

❖ Software Cost Estimation Techniques

Within most organizations, software cost estimates are based on past performance. Historical data are used to identify cost factors. Cost and productivity data must be

collected on current projects in order to estimate future ones. It can be done either top-down or bottom-up.

Top down estimation first focuses on system-level costs, such as computing resources and personnel required to develop the system, the costs of configuration management, quality assurance, system integration, training, and publications. Personnel costs are estimated by examining the cost of similar past projects.

Bottom up estimation first estimates the cost to develop each module or subsystem. Those costs are combined to arrive at an overall estimate.

Expert Judgement

The most widely used cost estimation technique is expert judgement, which is an top-down estimation technique. Expert judgement relies on the experience, background, and business sense of one or more key people in the organization.

An expert might arrive at a cost estimate in the following manner: The system to be developed is a process control system similar to one that was developed last year in 10 months at a cost of $1 million. The new system has similar control functions, but has 25 percent more activities to control; thus, we will increase our time and cost estimates by 25 percent. We will use the same computer and external sensing/controlling devices; and many of the same people are available to develop the new system, so we can reduce our estimate by 20 percent.

We can reuse much of low-level code from the previous product, which reduces the time and cost estimates by 25 percent. The net effect of these considerations is a time and cost estimates by 20 percent, which results in an estimate of $800,000 and 8 months development time. The customer has budgeted $1 million and 1 year delivery time for the system. Therefore, we add a small margin of safety and bid the system at $850,000 and 9 months development time.

These estimates reflect a well-balanced approach, ensuring that the offer falls within the customer's budget while allowing for additional security in the project timeline and costs. Thus, the bid aligns closely with customer expectations while incorporating the benefits of prior work and efficiencies gained through code reuse.

The biggest advantage of expert judgment, namely, **experience**, can also be a liability. The expert may be confident that the project is similar to a previous one. Groups of experts sometimes prepare a consensus estimate to minimize individual oversights and lack of familiarity.

The major disadvantage of group estimation is the effect that interpersonal group dynamics may have on individuals in the group.

Delphi Cost Estimation

The Delphi technique was developed by Rand Corporation in 1948 to gain expert consensus without introducing the adverse side effects of group meetings. The Delphi technique can be adapted to software cost estimation in the following manner:

A coordinator provides each estimator with the *System Definition* document and a form for recording a cost estimate.

Estimators study the definition and complete their estimates anonymously. They may ask questions of the coordinator, but they do not discuss their estimates with one another.

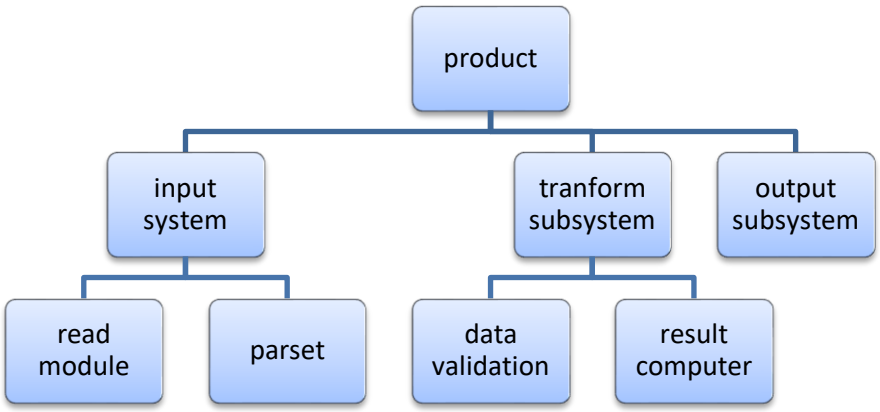
The coordinator prepares and distributes a summary of the estimators' responses, and includes any ususual rationales noted by the estimators.

Estimators complete another estimate, again anonymously, using the results from the previous estimate. Estimators whose estimates differ sharply from the group may be asked, anonymously, to provide justification for their estimates.

The process is iterated for as many rounds as required. No group discussion is allowed during the entire process.

Work Breakdown Structures

Expert judgment and group consensus are top-down estimation techniques. The work breakdown structure method is a bottom-up estimation tool. A work breakdown structure is a hierarchical chart that accounts for the individual parts of a system. A WBS chart can indicate either product hierarchy or process hierarchy



A product work breakdown structure

Product hierarchy identifies the product components and indicates the manner in which the components are interconnected. A WBS chart of process hierarchy identifies the work activities and the relationships among those activities. Using WBS technique, costs are estimated by assigning costs to each individual component in the chart and summing the costs.

Some planners use both product and process WBS chart for cost estimation. The primary advantages of the WBS technique are in identifying and accounting for various process and product factors, and in making explicit exactly which costs are included in the estimate.

Algorithmic Cost Models

Algorithmic cost estimators compute the estimated cost of a software system as the sum of the costs of the modules and subsystems that comprise the system. Algorithmic models are thus bottom-up estimators.

The Constructive Cost Model (COCOMO) is an algorithmic cost model described by Boehm in 1970 based on study of 63 projects. COCOMO is a regression model based on number of Lines of Code (LOC). COCOMO is based on procedural cost estimate model. COCOMO is used to reliably predict various parameters associated with making a project such as size, effort, cost, time and quality.

Boehm uses three levels of product complexity and provides equations to predict total programmer- months of effort, PM, in terms of the number of thousands of delivered source instruction, KDSI, in the product. Programmer cost for a software project can be obtained by multiplying the effort in programmer-months by the cost per programmer-month. The equations were derived by examining historical data from a large number of actual projects.

In Boehm's terminology, the three levels of product complexity are organic, semidetached, and embedded programs.

Application programs: PM = 2.4*(KDSI)**1.05

Utility programs: PM = 3.0*(KDSI)**1.12

Systems programs: PM = 3.6*(KDSI)**1.20

The development time for a program is

Application programs: TDEV = 2.5*(PM)**0.38

Utility programs: TDEV = 2.5*(PM)**0.35

Systems programs: TDEV = 2.5*(PM)**0.32

Given the total programmer-months for a project and the nominal development time required, the average staffing level can be obtained by simple divisions. For our 60 KDSI program, we obtain the following results:

Application programs: 176.6 PM/17.85 MO = 9.9 programmers

Utility programs: 294 PM/18.3 MO = 16 programmers

Systems programs: 489.6 PM/18.1 MO = 27 programmers

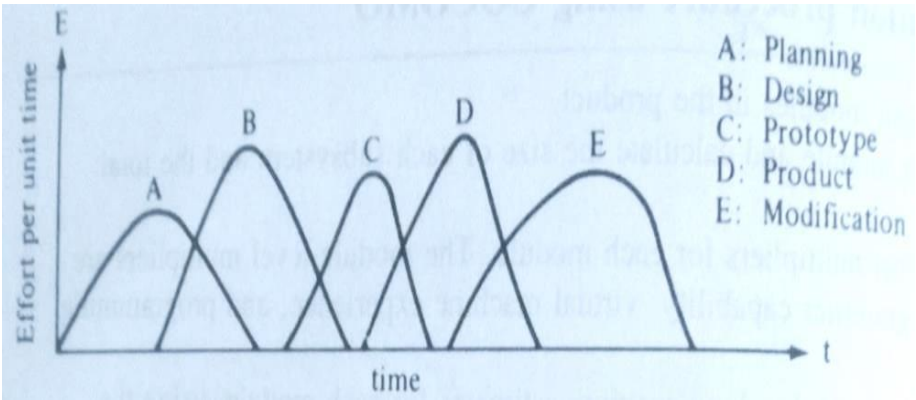
Effort multipliers are used to adjust the estimate for product attributes, computer attributes, personnel attributes, and project attributes.

The COCOMO equations incorporate a number of assumptions. For example, the nominal organic mode (application programs) equations apply in the following situations:

- Small to medium-size projects (2K to 32K DSI)
- Familiar applications area
- Stable, well-understood virtual machine
- In-house development effort

❖ Staffing Level Estimation

The number of personnel required throughout a software development project is not constant. Typically, planning and analysis are performed by a small group of people, architectural design by a larger, but still small, group, and detailed design by a larger number of people. Implementation and system testing require the largest numbers of people. The early phase of maintenance may require numerous personnel, but the number should decrease in a short time. In the absence of major enhancement or adaptation, the number of personnel for maintenance should remain small.



Cycles in a research and development project

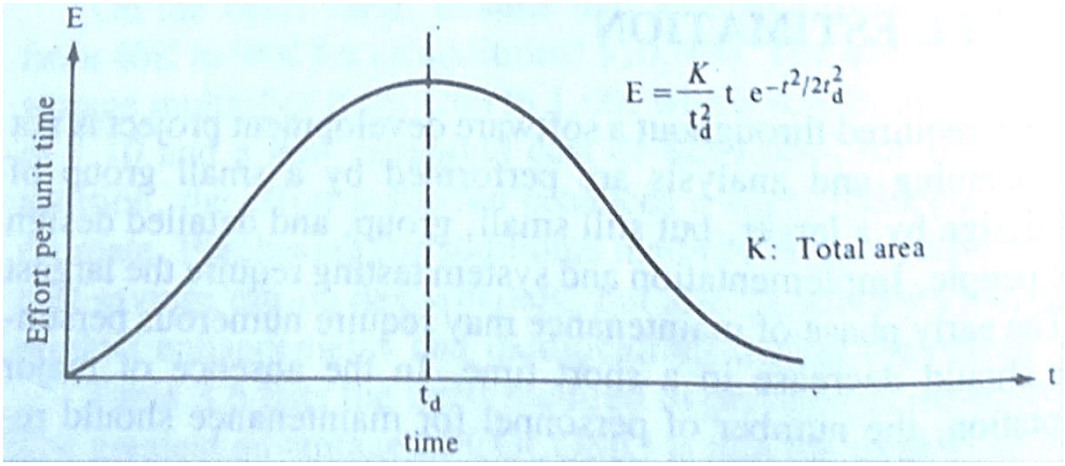
In 1958, Norden observed that research and development projects follow a cycle of planning, design, prototype, development, and use, with the corresponding personnel utilization. The sum of the areas under the curves can be approximated by the Rayleigh equation. Any particular point in Rayleigh curve represents the number of full-time equivalent personnel required at that instant in time.

Norden's Work

Norden studied the staffing patterns of several R & D projects. He found that the staffing pattern can be approximated by the Rayleigh distribution curve. Norden represented the Rayleigh curve by the following equation:

$$E = K/t^2_d * t * e^{-t^2 / 2t^2_d}$$

Where E is the effort required at time t. E is an indication of the number of engineers (or the staffing level) at any particular time during the duration of the project, K is the area under the curve, and td is the time at which the curve attains its maximum value. It must be remembered that the results of Norden are applicable to general R & D projects and were not meant to model the staffing pattern of software development projects



The Rayleigh curve of effort vs. time

In 1976, Putnam reported that the personnel level of effort required throughout the life cycle of a software product has a similar envelope. Putnam studied 50 Army software projects and 150 other projects to determine how the Rayleigh curve can be used to describe the software life cycle.

Putnam's Work

Putnam studied the problem of staffing of software projects and found that the software development has characteristics very similar to other R & D projects studied by Norden and that the Rayleigh-Norden curve can be used to relate the number of delivered lines of code to the effort and the time required to develop the project. By analyzing a large number of army projects, Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

- K is the total effort expended (in PM) in the product development and L is the product size in KLOC.

- t_d corresponds to the time of system and integration testing. Therefore, td can be approximately considered as the time required to develop the software.

- C_k is the state of technology constant and reflects constraints that impede the progress of the programmer. Typical values of C_k = 2 for poor development environment (no methodology, poor documentation, and review, etc.), C_k = 8 for good software development environment (software engineering principles are adhered to), C_k = 11 for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used). The exact value of C_k for a specific project can be computed from the historical data of the organization developing it.

Putnam suggested that optimal staff build-up on a project should follow the Rayleigh curve. Only a small number of engineers are needed at the beginning of a project to carry out planning and specification tasks. As the project progresses and more detailed work is required, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls. However, the staff build-up should not be carried out in large installments. The team size should either be increased or decreased slowly whenever required to match the Rayleigh-Norden curve.

Effect of schedule change on cost

By analyzing a large number of army projects, Putnam derived the following expression:

$$K = L^3/C_k^3 t_d^4$$

Where, **K** is the total effort expended (in PM) in the product development

L is the product size in KLOC

t_d corresponds to the time of system and integration testing

C_k Is the state of technology constant and reflects constraints that impede the progress of the program

Now by using the above expression, it is obtained that,

$$K = L^3/C_k^3 t_d^4$$

Or $$K = C/t_d^4$$

For the same product size, $C = L^3 / C_k^3$ is a constant.

Or $$\frac{K_1}{K_2} = t_{d2}^4/t_{d1}^4$$

Or $$K \propto 1/t_d^4$$

Or, $$cost \propto 1/t_d$$

(As project development effort is equally proportional to project development cost)

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in a substantial penalty of human effort as well as development cost.

For example, if the estimated development time is 1 year, then to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

Boehm also presents the distribution of effort and schedule in a software

development project.

Activity	Effort		Schedule	
	32 KDSI	128 KDSI	32 KDSI	128 KDSI
Plans and requirements	6%	6%	12%	13%
Architectural design	16%	16%	19%	19%
Detailed design	24%	23%	combined values:	
Coding and unit test	38%	36%	55%	51%
System test	22%	25%	26%	30%

Distribution of effort for application programs

❖ Estimating Software Maintenance Costs

Software maintenance typically requires 40-60%, and in some cases 90%, of the total life-cycle effort devoted to a software product. Maintenance activities include adding enhancements to the product, adapting the product to new processing environments, and correcting problems.

A widely used distribution of maintenance activities is 60% for enhancements, 20% for adaptation, and 20% for error correction. In a survey of 487 business data processing installations, Lientz and Swanson determined that the typical level of effort devoted to software maintenance was around 50% of total life-cycle effort, and that the distribution of maintenance activities was 51.3% for enhancement, 23.6% for adaptation, 21.7% for repair, and 3.4% for other.

Activity	% Effort
Enhancement	51.3
Improved Efficiency	4.0
Improved Documentation	5.5
User Enhancements	41.8
Adaptation	23.6
Input data, files	17.4

Hardware, Operating System	6.2
Corrections Emergency Fixes Scheduled Fixes	21.7 12.4 9.3
Others	3.4

Maintenance effort distribution

Lientz and Swanson determined that the maintenance programmer in a business data processing installation maintains 32K source instructions. For real-time and aerospace software, numbers in the range of 8K to 10K are more typical.

An estimate of the number of full-time software personnel needed for software maintenance can be determined by dividing the estimated number of source instructions to be maintained by a maintenance programmer. For example, if a maintenance programmer can maintain 32 KDSI, then two maintenance programmers are required to maintain 64 KDSI:

$$FSP = (64\ KDSI)/(32\ KDSI/FSP) = 2\ FSPm$$

Boehm suggests that maintenance effort can be estimated by use of an activity ratio, which is the number of source instructions to be added or modified in any given time period divided by the total number of instructions:

$$ACT = (DSI_{added} + DSI_{modified})/DSI_{total}$$

The activity ratio is then multiplied by the number of programmer-months required for development in a given time period to determine the number of programmer-months required for maintenance in the corresponding time period:

$$PMm = ACT * MM_{dev}$$

A further enhancement is provided by an effort adjustment factor EAF, which recognizes that the effort multipliers for maintenance may be different from the effort multipliers used for development:

$$PMm = ACT * EAF * MM_{dev}$$

Heavy emphasis on reliability and the use of modern programming practices during development may reduce the amount of effort required for maintenance, while low emphasis on reliability and modern practices during development may increase the difficulty of maintenance.

❖ COCOMO Model – Software Engineering

The Constructive Cost Model (COCOMO) is a software cost estimation model that helps predict the effort, cost, and schedule required for a software development project. Developed by Barry Boehm in 1981, COCOMO uses a mathematical formula based on the size of the software project, typically measured in lines of code (LOC).

The key parameters that define the quality of any software product, which are also an outcome of COCOMO, are primarily effort and schedule:

1. **Effort:** Amount of labor that will be required to complete a task. It is measured in person-months units.
2. **Schedule:** This simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put in. It is measured in the units of time such as weeks, and months.

Types of Projects in the COCOMO Model

In the COCOMO model, software projects are categorized into three types based on their complexity, size, and the development environment. These types are:

1. **Organic:** A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.
2. **Semi-detached**: A software project is said to be a Semi-detached type if the vital characteristics such as team size, experience, and knowledge of the various programming environments lie in between organic and embedded. The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered Semi-Detached types.
3. **Embedded:** A software project requiring the highest level of complexity, creativity, and experience requirement falls under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

Comparison of these three types of Projects in COCOMO Model

Aspects	Organic	Semidetached	Embedded
Project Size	2 to 50 KLOC	50-300 KLOC	300 and above KLOC
Complexity	Low	Medium	High
Effort Equation	E = 2.4(400)1.05	E = 3.0(400)1.12	E = 3.6(400)1.20
Example	Simple payroll system	New system interfacing with existing systems	Flight control software

Importance of the COCOMO Model

1. **Cost Estimation:** To help with resource planning and project budgeting, COCOMO offers a methodical approach to software development cost estimation.
2. **Resource Management:** By taking team experience, project size, and complexity into account, the model helps with efficient resource allocation.
3. **Project Planning**: COCOMO assists in developing practical project plans that include attainable objectives, due dates, and benchmarks.
4. **Risk management**: Early in the development process, COCOMO assists in identifying and mitigating potential hazards by including risk elements.
5. **Support for Decisions**: During project planning, the model provides a quantitative foundation for choices about scope, priorities, and resource allocation.
6. **Benchmarking**: To compare and assess various software development projects to industry standards, COCOMO offers a benchmark.
7. **Resource Optimization:** The model helps to maximize the use of resources, which raises productivity and lowers costs.

- **Basic COCOMO Model**

The Basic COCOMO model is a straightforward way to estimate the effort needed for a software development project. It uses a simple mathematical formula to predict how many person-months of work are required based on the size of the project, measured in thousands of lines of code (KLOC).

It estimates effort and time required for development using the following expression:

E = a(KLOC)^b PM*

Tdev = c(E)^d*

Person required = Effort/ Time

Where,

E is effort applied in Person-Months

KLOC is the estimated size of the software product indicate in Kilo Lines of Code

Tdev is the development time in months

a, b, c are constants determined by the category of software project given in below table.

The above formula is used for the cost estimation of the basic COCOMO model and also is used in the subsequent models. The constant values a, b, c, and d for the Basic Model for the different categories of the software projects are:

Software Projects	a	b	c	d
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

1. The effort is measured in Person-Months and as evident from the formula is dependent on Kilo-Lines of code. The development time is measured in months.
2. These formulas are used as such in the Basic Model calculations, as not much consideration of different factors such as reliability, and expertise is taken into account, henceforth the estimate is rough.

Example of Basic COCOMO Model

Suppose that a Basic project was estimated to be 400 KLOC (kilo lines of code). Calculate effort and time for each of the three modes of development. All the constants value provided in the following table:

Solution

From the above table we take the value of constant a,b,c and d.

1. For organic mode,
   - effort = 2.4 × $(400)^{1.05}$ ≈ 1295 person-month.
   - dev. time = 2.5 × $(1295)^{0.38}$ ≈ 38 months.
2. For semi-detach mode,
   - effort = 3 × $(400)^{1.12}$ ≈ 2462 person-month.
   - dev. time = 2.5 × $(2462)^{0.35}$ ≈ 38 months.
3. For Embedded mode,
   - effort = 3.6 × $(400)^{1.20}$ ≈ 4772 person-month.
   - dev. time = 2.5 × $(4772)^{0.32}$ ≈ 38 months.

Below are the programs for Basic COCOMO Model:

```
import java.util.Arrays;
public class BasicCOCOMO
{
   private static final double[][] TABLE =
   {
      {2.4, 1.05, 2.5, 0.38},
      {3.0, 1.12, 2.5, 0.35},
      {3.6, 1.20, 2.5, 0.32}
   };
   private static final String[] MODE =
   {
      "Organic", "Semi-Detached", "Embedded"
   };

   public static void calculate(int size)
   {
      int model = 0;

      // Check the mode according to size
      if (size >= 2 && size <= 50)
      {
         model = 0;
      } else if (size > 50 && size <= 300)
      {
         model = 1;
      } else if (size > 300)
      {
         model = 2;
```

```
        }
        System.out.println("The mode is " + MODE[model]);
        // Calculate Effort
        double effort = TABLE[model][0] * Math.pow(size,
                                        TABLE[model][1]);
        // Calculate Time
        double time = TABLE[model][2] * Math.pow(effort,
                                       TABLE[model][3]);
        // Calculate Persons Required
        double staff = effort / time;
        // Output the values calculated
        System.out.println("Effort = " + Math.round(effort) +
                           " Person-Month");
        System.out.println("Development Time = " + Math.round(time) +
                           " Months");
        System.out.println("Average Staff Required = " + Math.round(staff) +
                           " Persons");
    }
    public static void main(String[] args)
    {
        int size = 4;
        calculate(size);
    }
}
```

Output

```
The mode is Organic

Effort = 10.289 Person-Month

Development Time = 6.06237 Months

Average Staff Required = 2 Persons
```

Examples

1. **NASA Space Shuttle Software Development:** NASA estimated the time and money needed to build the software for the Space Shuttle program using the COCOMO model. NASA was able to make well-informed decisions on resource allocation and project scheduling by taking into account variables including project size, complexity, and team experience.

2. **Big Business Software Development:** The COCOMO model has been widely used by big businesses to project the time and money needed to construct intricate business software systems. These organizations were able to better plan and allocate resources for their software projects by using COCOMO's estimation methodology.

3. **Commercial Software goods:** The COCOMO methodology has proven advantageous for software firms that create commercial goods as well. These businesses were able to decide on pricing, time-to-market, and resource allocation by precisely calculating the time and expense of building new software products or features.

Advantages of the COCOMO Model

1. **Systematic cost estimation:** Provides a systematic way to estimate the cost and effort of a software project.
2. **Helps to estimate cost and effort:** This can be used to estimate the cost and effort of a software project at different stages of the development process.
3. **Helps in high-impact factors:** Helps in identifying the factors that have the greatest impact on the cost and effort of a software project.
4. **Helps to evaluate the feasibility of a project:** This can be used to evaluate the feasibility of a software project by estimating the cost and effort required to complete it.

Disadvantages of the COCOMO Model

1. **Assumes project size as the main factor:** Assumes that the size of the software is the main factor that determines the cost and effort of a software project, which may not always be the case.
2. **Does not count development team-specific characteristics:** Does not take into account the specific characteristics of the development team, which can have a significant impact on the cost and effort of a software project.
3. **Not enough precise cost and effort estimate:** This does not provide a precise estimate of the cost and effort of a software project, as it is based on assumptions and averages.

❖ Software Requirement Specification (SRS)

Software Requirement Specification (SRS) Format as the name suggests, is a complete specification and description of requirements of the software that need to be fulfilled for the successful development of the software system. These requirements can be functional as well as non-functional depending upon the type of requirement. The interaction between different customers and contractors is done because it is necessary to fully understand the needs of customers.

Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications

that is needed to be done to increase quality of product and to satisfy customer's demand.

- **Purpose of this Document –** At first, main aim of why this document is necessary and what's purpose of document is explained and described.
- **Scope of this document –** In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.
- **Overview –** In this, description of product is explained. It's simply summary or overall review of product.

General description

In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.

Functional Requirements

In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order. Functional requirements specify the expected behavior of the system-which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, detailed description all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

Interface Requirements

In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described or explained. Examples can be shared memory, data streams, etc.

Performance Requirements

In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc. The performance requirements part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic. Static requirements are those that do not impose constraint on the execution characteristics of the system.

Dynamic requirements specify constraints on the execution behaviour of the system.

Design Constraints

In this, constraints which simply mean limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc. There are a number of factors in the client's environment that may restrict the choices of a designer leading to design constraints such factors include standards that must be followed resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Non-Functional Attributes

In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

Preliminary Schedule and Budget

In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

Appendices

In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

Uses of SRS document

- Development team require it for developing product according to the need.
- Test plans are generated by testing group based on the describe external behaviour.
- Maintenance and support staff need it to understand what the software product is supposed to do.
- Project manager base their plans and estimates of schedule, effort and resources on it.
- customer rely on it to know that product they can expect.
- As a contract between developer and customer.

- in documentation purpose.

Properties of a good SRS document

Concise: The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.

Structured: It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

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

Conceptual integrity: It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.

Verifiable: All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

❖ Formal Specification Techniques

Formal specification techniques are mathematically based methods used to specify software systems clearly and unambiguously. These methods are essential for ensuring system reliability, especially as software increasingly impacts society

. They allow for the precise description of requirements and facilitate the validation and verification of software designs against these specifications.

Characteristics of Formal Specifications

A formal specification typically describes what a system should do rather than how it should be implemented. Good specifications possess traits such as being

adequate, consistent, unambiguous, complete, and manageable, which are crucial for effective communication among stakeholders. The ability to perform proofs based on formal specifications supports validation and verification activities, enhancing trust in the software products.

Types of Formal Specification Paradigms

Various paradigms exist within formal specification techniques, including state-based, transition-based, and functional specifications.

State-based specifications focus on the states of the system and the transitions between them, making them useful for reactive systems.

Transition-based specifications detail how a system transitions from one state to another, while functional specifications describe system functionalities in terms of mathematical functions.

Formal Specification Languages

The languages used for formal specifications are rigorously defined to eliminate ambiguity. Notable examples include Z notation, the Vienna Development Method (VDM), and the Abstract Machine Notation (AMN). These languages facilitate the modeling of software systems, including their behaviors and interfaces, making them effective for both implementation and verification processes.

Advantages of Using Formal Specification

The employment of formal specifications can lead to insights into software requirements and design, potentially allowing for the provable correctness of programs. The specification process aids in revealing inconsistencies and incompleteness in system requirements early, ultimately reducing the need for costly reworks during later phases of development. Furthermore, formal methods can be particularly beneficial in critical systems where failures are expensive or dangerous

❖ Languages and Processors for Requirements Specification

Language processor:-

1. We know that a computer understands instructions in machine code i.e. 0's & 1's and the programs are mostly written in High Level Language like C, C++,

JAVA etc. and they are called source code and it cannot be executed directly by the computer.

2. A **language processor** is special translator system software that is used to translate one code (source code) to another code (machine code).

3. **Language processor** can translate the source code or program code into machine code.

4. Source code or program code are written in HLL (High Level Language) which is easy for human understanding and machine code is written in LLL (Low Level language) which is easy for machine understanding.

5. We can also say that **a language processor** converts High Level Language into Low Level Language by the help of its types.

6. There are three types of **language processor**: -

a. Compiler

b. Assembler c. Interpreter

Language processing activities: -

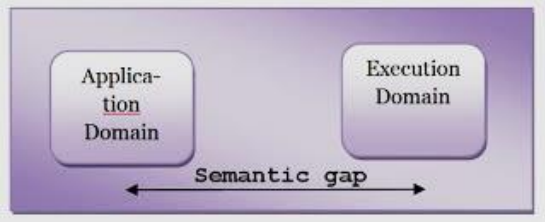
1. The activities of language processing arise to bridge the ideas of software designer with actual execution on the computer system.

2. The ideas express by the designer in terms related to the application domain of the software and to implement these ideas their description has to be interpreted in terms related to the execution domain of the computer system.

3. A software is a language processor which either bridges **specification gap** or **execution gap.**

4. The fundamental of language processing activities can be divided into those that bridge the **execution gap** and **specification gap.**

5. **Semantic gap** is the gap between application domain and execution domain.



6. For a language processor the input program is termed as source program and language use for it i.e. termed as source language and the output program is termed as target program.

7. The language process activities are divided into two groups as follow:-

a. **Program Generation Activities**

b. **Program Execution Activities**

Language specification: -

1. A language specification is a formal language i.e. used in computer science and this language is not directly executed.

2. Language specification describe the system at a much higher level than a programming design.

3. Language specification are used during the systems analysis, requirements analysis and systems design.

4. Enabling the creation of proofs of program correctness is an important use of specification languages.

5. A common assumption of many specification approaches is that programs are modelled as algebraic or model theoretic structures which include a collection of sets of data values together with functions.

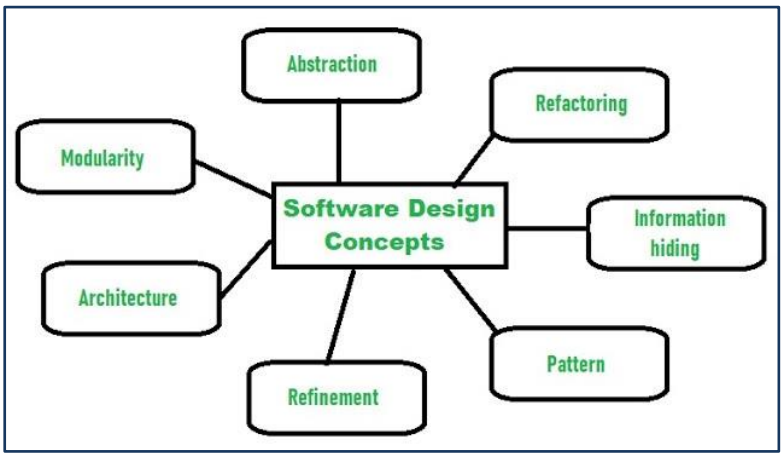
UNIT-3

➢ Introduction of Software Design Process

Software Design is the process of transforming user requirements into a suitable form, which helps the programmer in software coding and implementation. During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. Hence, this phase aims to transform the SRS document into a design document.

Software Design Concepts

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, and the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system software or product that is to be developed or built



There are many concepts of software design and some of them are given below

1. **Abstraction (Hide Irrelevant data):** Abstraction simply means to hide the details to reduce complexity and increase efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity (subdivide the system):** Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software

available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important.

3. **Architecture (design a structure of something):** Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement (removes impurities):** Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern (a Repeated form):** A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding (Hide the Information):** Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.
7. **Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

Different levels of Software Design

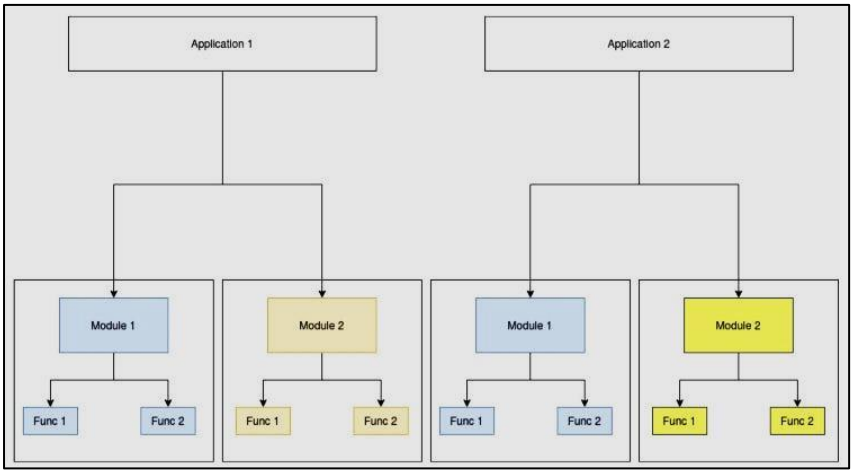
There are three different levels of software design. They are:

1. **Architectural Design:** The architecture of a system can be viewed as the overall structure of the system and the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.
2. **Preliminary or high-level design:** Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.
3. **Detailed design:** Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

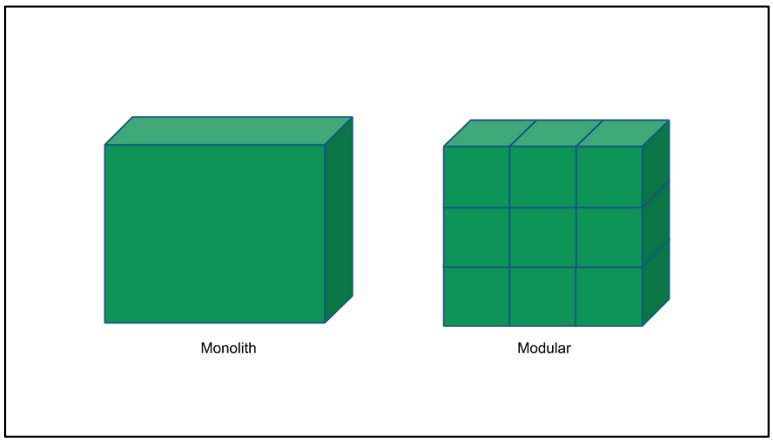
➢ modularization

In this current age of software, you would be hard-pressed to find a program that isn't continuously growing and evolving. Designing a program all at once, with all required functions, would be difficult due to its size, complexity and constant changes. This is where modularization comes in.

Modularization is the process of separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.



With modularization, we can easily work on adding separate and smaller modules to a program without being hindered by the complexity of its other functions. In short, it's about being flexible and fast in adding more software functions to a program. In a software engineering team, we could easily work independently on each module without affecting others' work.



Monolith vs modular program

It is the backbone of the microservice architecture, whose basic idea is also the simplicity of developing applications that are easier to extend and maintain when disassembled into small and independent parts. On the other hand, in monolithic architecture, there's always the risk of bringing down the whole program with a simple update.

Without modularization, this will lead to an increase in development time, the number of bugs and the duration it takes to test and release a program.

Benefits of modularization

Why should we be decomposing our projects into modules? As shown in my experience with a single-file program, programs without proper modularization would be a nightmare to maintain and extend.

In modularization, the modules have minimal dependency on other modules. So, we can easily make changes in a module without affecting other parts of the program.

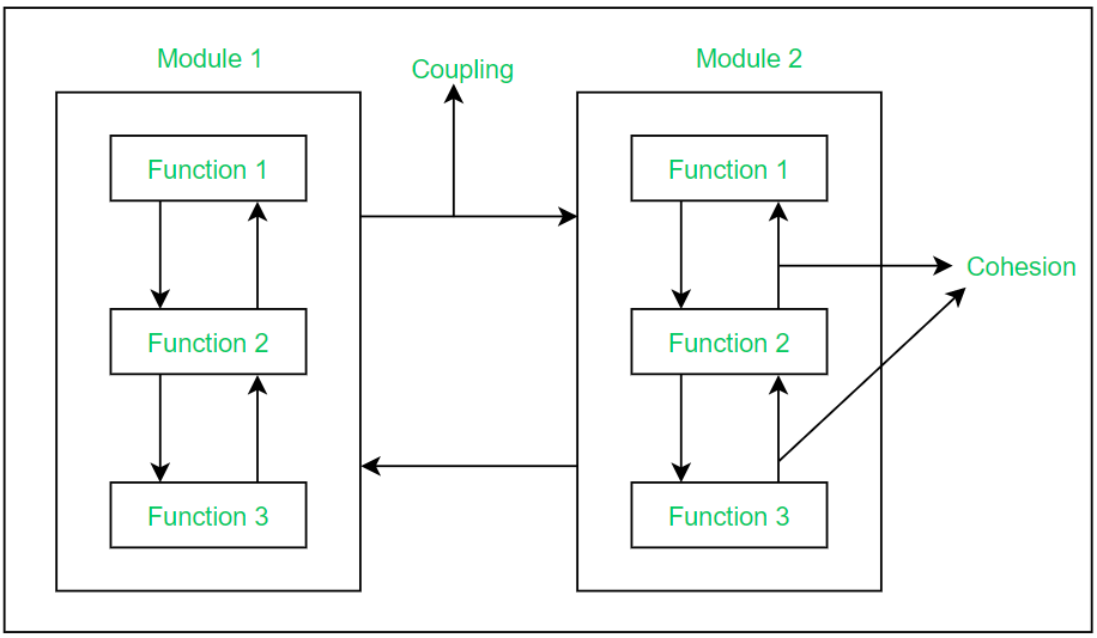
The following are just the gist of how modularization would improve the development process for a program.

- Easier to add and maintain smaller components
- Easier to understand each module and their purpose
- Easier to reuse and refactor modules
- Better abstraction between modules
- Saves time needed to develop, debug, test and deploy a program

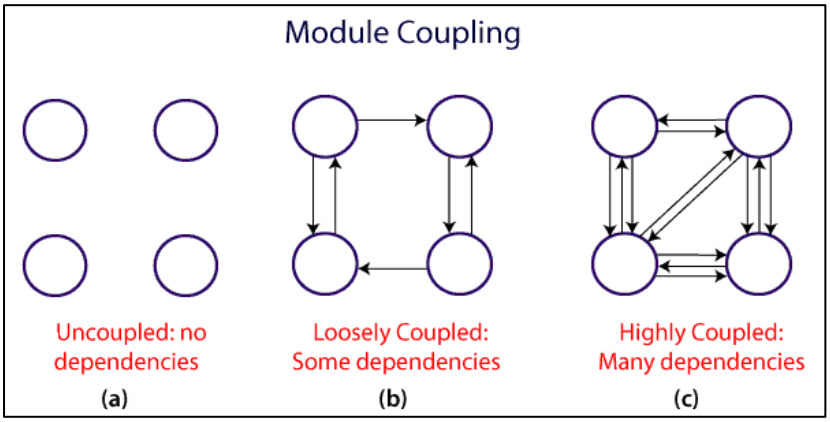
A module should have only a single responsibility, that is the Single Responsibility Principle. Thus, it should depend minimally on other modules. The independence of a module can be measured using coupling and cohesion.

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Cohesion and coupling



Module coupling, with (c) being what we want to avoid

Each module should have a clear and focused purpose, such that its developers have a clear idea of the requirement for each function. Its interface should be easy to understand and use, even without understanding its implementation details. Thus, leading into our next point, is that its implementation details, while not only correct, should be encapsulated and private, and that it should be changeable without affecting another module. Furthermore, the dependency between modules should be minimised.

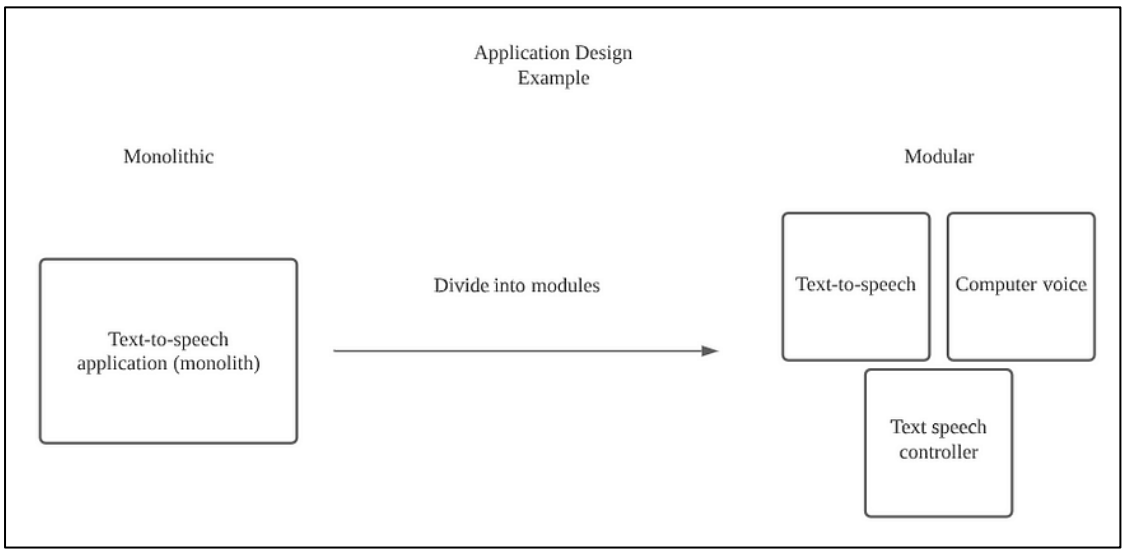
Example: Modularization on a text-to-speech application

Consider a text-to-speech application that will translate a user's input text into speech and read it out loud. It should be able to:

- Parse a user's input text
- Use a selected computer's voice to read out the text
- Have a controllers that can speed up or slow down the computer's speech if the user chooses

We can apply modularization to this application and decompose it to the following modules:

- Text-to-speech: Parses the user's text to be read out loud
- Computer voice: Stores and provides computer voices that the user can choose
- Text speech controller: Controls that speed of the speech that the user chooses



❖ Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following

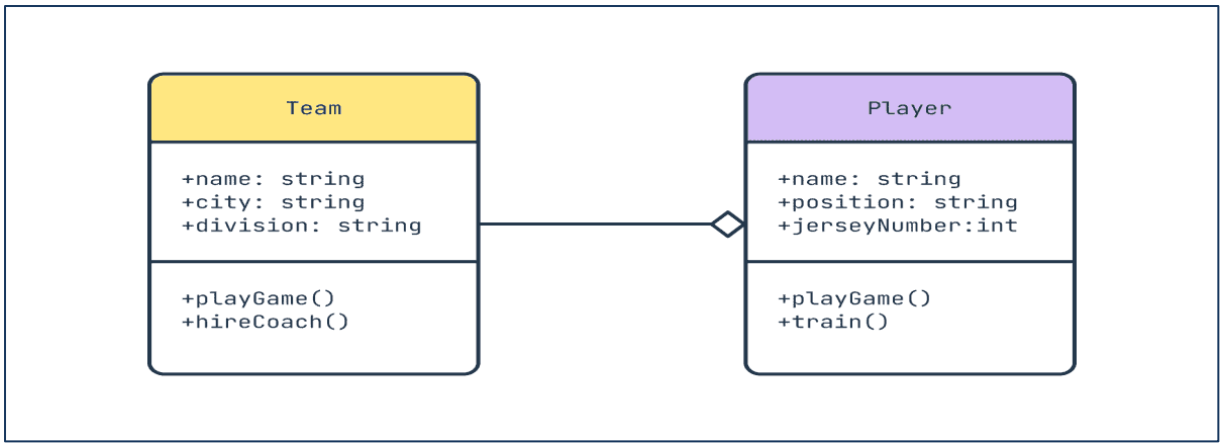
Design notations are the various ways in which a system's design is visually represented or described during the software development process. These notations help developers, designers, and stakeholders understand and communicate the structure, behavior, and interactions within a system. Here are some commonly used design notations

1. Unified Modeling Language (UML)

UML, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. The UML is a very important part of developing object oriented software and the software development process.

a) Class Diagram

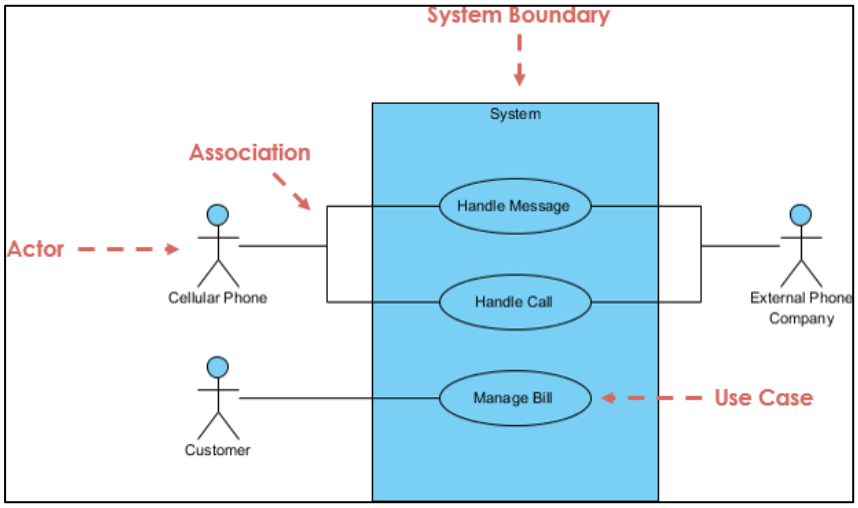
The class diagram is a central modeling technique that runs through nearly all object-oriented methods. This diagram describes the types of objects in the system and various kinds of static relationships which exist between them.



b) Use Case Diagram

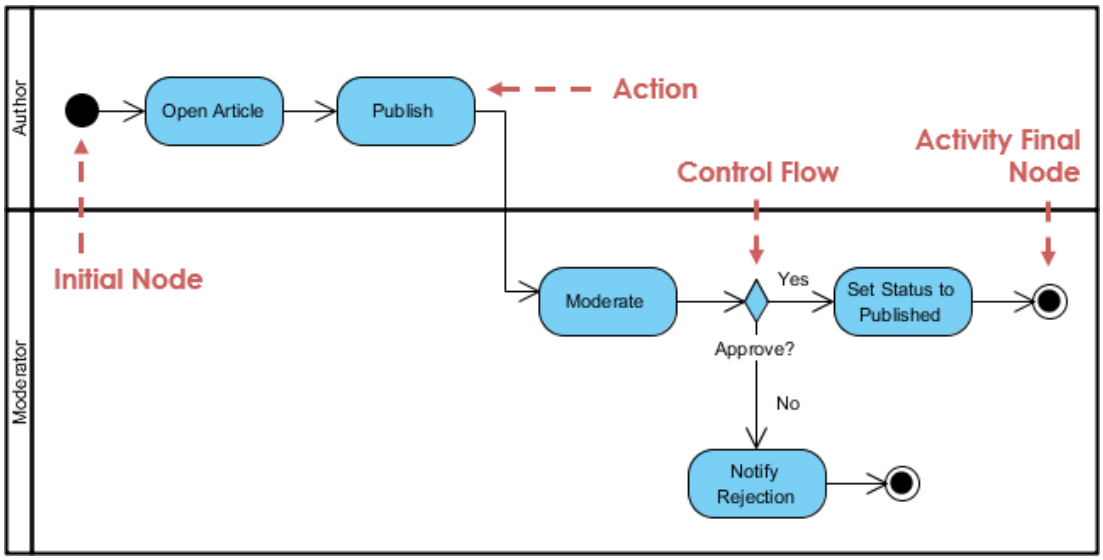
A use-case model describes a system's functional requirements in terms of use cases. It is a model of the system's intended functionality (use cases) and its

environment (actors). Use cases enable you to relate what you need from a system to how the system delivers on those needs.



c) Activity Diagram

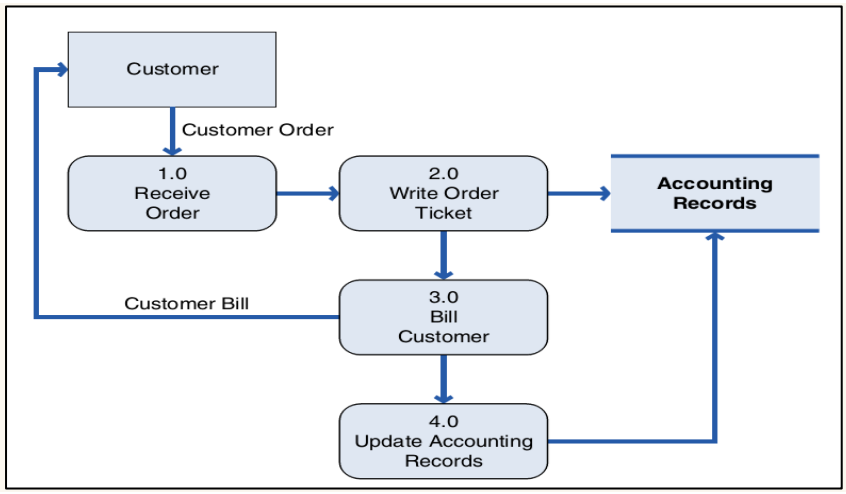
Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. It describes the flow of control of the target system, such as the exploring complex business rules and operations, describing the use case also the business process. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows).



2. Data flow diagram

A data flow diagram (DFD) maps out how information, actors, and steps flow within a process or system. It uses symbols to show the people and processes needed to move data correctly.

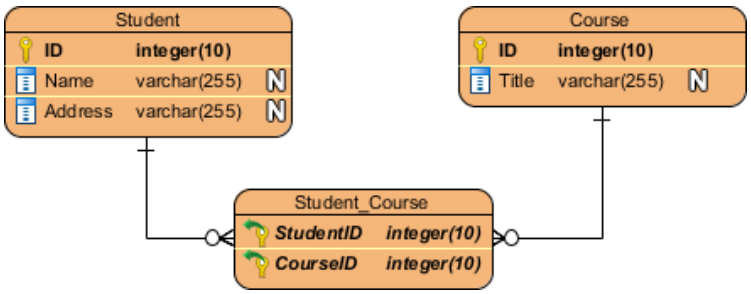
DFDs are important because they help you visualize how data moves through your system, spot inefficiencies, and find opportunities to improve overall functionality. This leads to more efficient operations, better decision-making, and enhanced communication among team members



data flow diagram are generally used to show how data moves through a system, emphasizing data flow and processes rather than detailed software behavior.

ER diagram

An Entity Relationship (ER) Diagram is a type of flowchart that illustrates how "entities" such as people, objects or concepts relate to each other within a system. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, business information systems, education and research. Also known as ERDs or ER Models, they use a defined set of symbols such as rectangles, diamonds, ovals and connecting lines to depict the interconnectedness of entities, relationships and their attributes.



❖ System Design Strategies (or) Techniques

A good system design is to organize the program modules in such a way that are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.

Software Engineering is the process of designing, building, testing, and maintaining software. The goal of software engineering is to create software that is reliable, efficient, and easy to maintain. System design is a critical component of software engineering and involves making decisions about the architecture, components, modules, interfaces, and data for a software system.

There are many strategies or techniques for performing system design.

1. Structured Design

Structured design is primarily about breaking problems down into several well-organised components. The benefit of utilizing this design technique is that it simplifies difficulties. This allows for the minor pieces to be problem-solved so they can fit into the larger image. The solution components are organized hierarchically.

Structured design is primarily based on the **divide and conquer** technique, in which a large problem is divided into smaller ones, each of which is tackled independently until the larger problem is solved. Solution modules are used to address the individual problems. The structured design stresses the importance of these modules' organization to produce exact results. A good structured design has high cohesion and low coupling arrangements.

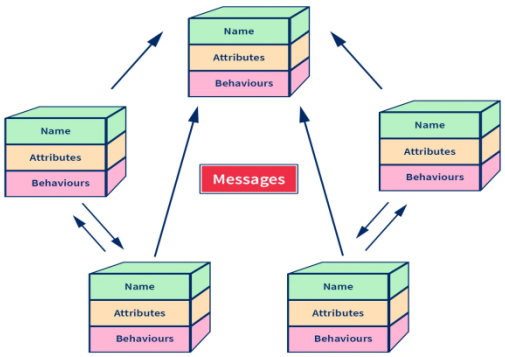
2. Function Oriented Design

Function-oriented design is related to structured design in that it splits the entire system into subsystems known as functions. The system is viewed as a map or top-down perspective of all the bundled functions. However, when compared to structured design, there is more information travelling between the functions, whilst the smaller functions promote abstraction. The software can also work on input rather than state thanks to the function-oriented design.

3. Object Oriented Design

This design approach differs from the other two in that it focuses on objects and classes. This technique is centred on the system's objects and their attributes. Furthermore, the characteristics of all these objects' attributes are encapsulated together, and the data involved is constrained so that polymorphism can be enabled. Object-oriented design is centered on recognizing objects and categorizing them

based on their attributes. The class hierarchy is then established, and the relationships between these classes are defined.

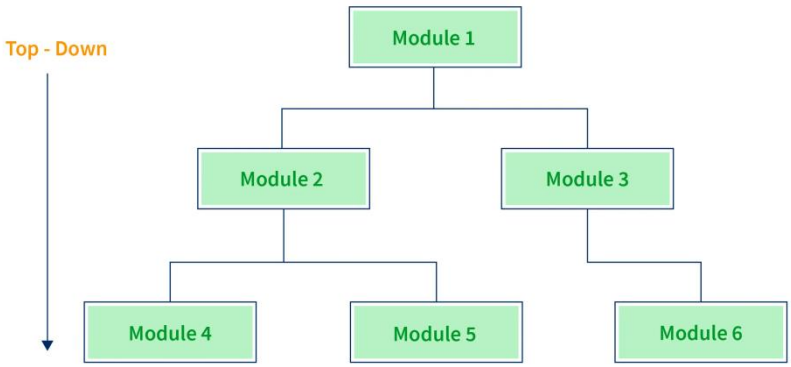


An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

The object-oriented design technique is considered superior to the function-oriented design approach because real-world entities may be easily incorporated in the computer world. This method also allows for the implementation of several very basic object behaviors like as polymorphism, inheritance, abstraction, and encapsulation.

Software Design Approaches

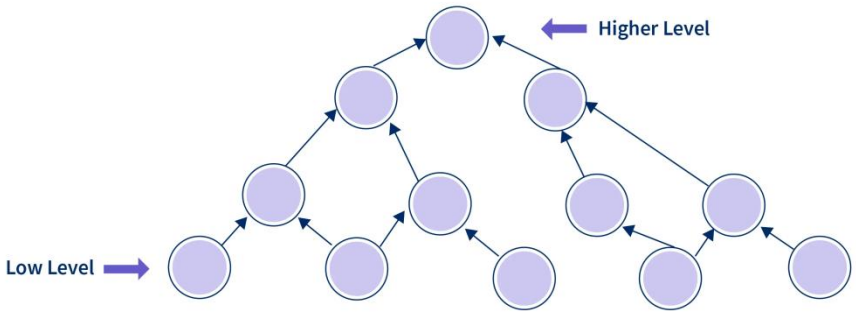
A) Top Down Approach



This design technique is entirely focused on first subdividing the system into subsystems and components. Rather to constructing from the bottom up, the top-down approach conceptualizes the entire system first and then divides it into multiple subsystems. These subsystems are then designed and separated into smaller

subsystems and sets of components that meet the larger system's requirements. Instead of defining these subsystems as discrete entities, this method considers the entire system to be a single entity. When the system is finally defined and divided based on its features, the subsystems are considered separate entities. The components are then organised in a hierarchical framework until the system's lowest level is designed.

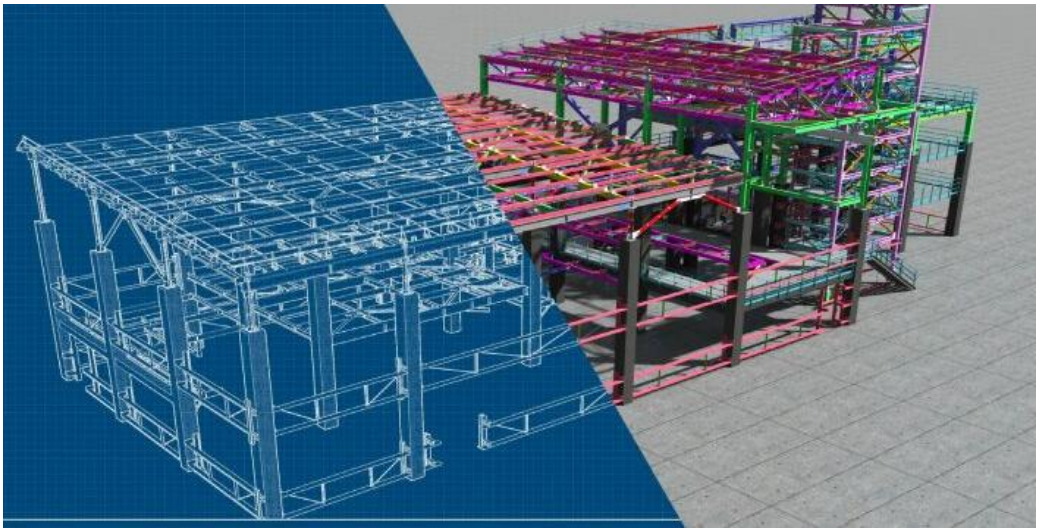
B) Bottom-Up Approach



This system design technique prioritises the design of subsystems and the lowest-level components (even sub-components). Higher-level subsystems and larger components can be produced more readily and efficiently if these components are designed beforehand. This reduces the amount of time spent on recon and troubleshooting. The process of assembling lower-level components into larger sets is repeated until the entire system is composed of a single component. This design technique also makes generic solutions and low-level implementations more reusable.

❖ Detailed Engineering Considerations for Project Success

Engineering projects today require meticulous planning and execution across various phases as they serve as the cornerstones of success. As part of this, one of the most critical stages is detailed engineering –where the project blueprint takes shape and becomes a reality. It is, therefore, considered to be the project's backbone. It serves as the bridge between conceptual design and actual construction while encompassing a multitude of tasks that are vital for the project's success. Further, it helps ensure seamless alignment of resources to achieve the desired outcome.



What is Detailed Engineering Design?

Detailed engineering design refers to comprehensive and precise technical drawings, calculations, and documentation, primarily 3D models that are conceptual plans for the construction and implementation phases. It would involve specifying equipment, materials, layouts, and operational details to ensure accurate execution. This model can be used as a virtual walk through for interference/ clash checks to simulate and test your solution/system's performance before constructing it for real-world usage.

Detail Engineering Team

An engineering team signifies a group of skilled professionals with diverse expertise collaborating to solve complex problems, design innovative solutions, and execute projects. This team collectively combines various technical disciplines, such as process, mechanical, instrumentation, electrical, civil, structural, architectural, and more, to achieve common goals. They leverage their specialized knowledge, experience, and creativity to efficiently analyze, plan, and implement projects. Effective communication, collaboration, and a shared vision are crucial for an engineering team's success, as they work cohesively to tackle challenges, optimize processes, and deliver high-quality results in the field of engineering and technology.

A team can consist of many people from different backgrounds with varying skills. However, at each stage, there needs to be one person designated as a project manager who has overall responsibility for coordinating the various activities necessary for completing a specific portion of your design.

❖ The Real-Time and Distributed Systems

The Real-Time and Distributed Systems research group is concerned with fundamental and applied research into the development and analysis of systems where the distributed nature of the computation, the need for communication and coordination, and/or the timeliness of the system's actions are of critical importance

to the overall functionality and to the end-users. The group's research spans many areas, including embedded systems, Internet of Things (IoT), communications, robotics, automotive systems, large scale process control, avionics, distributed computing, and High-Performance Computing (HPC).

Real-time systems are those that are required to respond to inputs within a finite and specified time interval. In some systems, the required response times are measured in milliseconds, in others it is seconds, minutes, or even hours. Nevertheless, they all have timing requirements that must be satisfied. In the production of real-time systems, it is insufficient to use testing of the final system to ensure its compliance with the requirements (as it is infeasible to test all possible timing interference patterns in a system of reasonable complexity). A comprehensive and systematic approach to specification, design, implementation and analysis is required.

Distributed systems are those that divide their workload across networked 'nodes' (e.g. processors, computers, embedded devices, robots), which coordinate their actions through message passing. These nodes may be tightly integrated via wired connections (e.g. High Performance Computing platforms), or loosely connected through wireless communication (e.g. IoT devices and robot swarms). Nodes can also be distributed across a variety of spatial scales, from cloud platforms with computation spread across international data centres, to devices located throughout a home, or even networked processors within a single silicon chip. This presents unique challenges in terms of programmability, coordination, communication, and fault tolerance, each demanding consideration of the distributed nature of the system.

The research conducted by the Real-Time and Distributed Systems group is unified around the notions of understanding, modelling, analysing, simulating, optimising, and predicting the performance and use of systems that are real-time and/or distributed in nature.

❖ Software Test Plan

Software testing is integral to ensuring the software meets all the requirements and works correctly. A test plan is a document explaining how you'll test the software, what resources you'll need, and when it should be done. Creating a good test plan is vital to spot any issues. In this guide, we'll give you step-by-step instructions on how to make a practical test plan so that your software testing process will be successful.

A test plan is a document that outlines the strategy, objectives, resources, and schedule of a software testing process. The test plan will typically include details such as the type and number of tests that need to be conducted, the purpose of each test, the required tools, and how test results will be analyzed and reported. It is regularly

updated throughout the testing process to reflect any discoveries or changes in strategy.

Test plan importance

A test plan is essential for several reasons. Firstly, it is a communication tool between stakeholders and testing team members. This ensures that everyone understands what, why, and how to test. It will also outline how to report test findings, what to consider as a pass or fail, and any other criteria that may be applicable. Besides, it will outline the expected outcomes and ensure that testing happens according to plan. This is why it is important to know how to create a test plan.

Objectives of a test

The objectives of a test plan are to define the scope, approach, and resources required for testing. It aims to establish test objectives and deliverables, identify test tasks and responsibilities, outline the test environment and configuration, and define the test schedule to ensure efficient and effective testing.

A detailed test plan further assists individuals in working together to complete the project and maintain consistent and transparent communication throughout the testing process.

Components of test plan

A Test Plan is a document that outlines the strategy, scope, objectives, resources, and schedule of a testing process. It is an essential part of software development and testing, as it provides a roadmap for the execution of tests. The components of a Test Plan include:

1. **Test goal:** The test plan should explain what the testing is meant to accomplish, including the features and functions that will be tested and any requirements that must be met.
2. **Scope and approach:** It should also outline what will be tested, how it will be tested, and which testing methods or approaches will be used.
3. **Test environment:** It should specify the hardware, software, and network configurations needed for the tests and any third-party tools or systems used.
4. The test plan should include details about what you'll be doing when you'll be doing it, and what resources you need to do it.
5. It should list the deliverables, like test cases, scripts, and reports that will be created during testing. It should also have a schedule outlining the testing timeline, including the start and end dates.

6. The plan should identify the personnel (people), equipment, and facilities you'll need to complete the tests.
7. **Potential problems:** The test plan should list any potential issues arising during the testing process and how they will be dealt with.
8. **Approval:** The test plan needs to have a clear approval process where all stakeholders and project team members agree on the goals of the testing effort and sign off on it.

There are three types of test plans,

- Master Test Plan
- Phase Test Plan
- Specific Test Plan

Master Test Plan: Contains multiple testing levels and has a comprehensive test strategy.

Phase Test Plan: Tailored to address a specific phase within the overall testing strategy.

Specific Test Plan: Explicitly designed for other testing types like performance, security, and load testing. Simply put, it is a test plan focusing only on the non-functional aspects.

Write a Test Plan

Making a test plan is the most crucial task of the test management process. The following seven steps to prepare a test plan.

- First, analyze product structure and architecture.
- Now design the test strategy.
- Define all the test objectives.
- Define the testing area.
- Define all the useable resources.
- Schedule all activities in an appropriate manner.
- Determine all the Test Deliverables.

❖ Milestones, walkthrough and Inspection

1. Milestones

Each completed step is a "milestone". Knowing what work is expected to be done by what dates and keeping track of progress helps to keep the project on time and on budget.

Project planning simply means to plan how to set up and complete the project with given time period. It includes defined stages that are required for project objectives with designated resources. It is very important task for business development as it helps in identifying desired goals, reduces risk, and lastly delivers product that fulfills requirements of customers.

The project manager should recognize in advance problems that might occur in future and should be ready with desired solution to fix those problems. A project plan should be prepared in advance from all gathered information that is required. A project planning process is iterative because new information gets available at each phase of project development. Hence, plan needs to be modified on regular basis for accommodating new requirements of the project.

When project begins then it is expected that project related activities must be initiated. In project planning, series of milestones must be established. Milestone can be defined as recognizable endpoint of software project activity. At each milestone, report must be generated.

Milestone is distinct and logical stage of the project. It is used as signal post for project start and end date, need for external review or input and for checking budget, submission of the deliverable, etc. It simply represents clear sequence of events that are incrementally developed or build until project gets successfully completed. It is generally referred to as task with zero-time duration because they are used to symbolize an achievement or point of time in project. It helps in signifying change or stage in development.

2. Walkthrough

Imagine gathering around a virtual campfire with your fellow developers, each armed with a flashlight to shine on different corners of the codebase. That's basically what a software walkthrough is – a collective effort to understand, discuss, and review the software.

Unlike some of its more formal siblings, such as code reviews or inspections, a walkthrough is a bit more laid-back. It's an opportunity for developers, testers, and stakeholders to share insights into the software's architecture and behavior.

In this blog, we will discuss software walkthrough, how it works, its importance in the software development lifecycle, and more.

Software walkthrough is a type of peer review where a designer or programmer guides members of the development team and other stakeholders through a software product. During this process, participants ask questions and comment on potential errors, deviations from development standards, and other issues.

Also, participants typically go through the various features and functionalities of the software step by step, discussing and analyzing its behavior, design, and functionality. It can include examining the user interface, testing specific functionalities, and reviewing the underlying code or architecture.

A software walkthrough is done for several reasons. The main reason is that the walkthrough process contributes to the overall improvement of the software development process and the quality of the software product.

Here are some of the key reasons why it is performed,

- To gather feedback on the technical quality or content of the document.
- To familiarize the audience with the software.
- It helps identify and rectify errors in the early stages of SDLC.
- Ensures that the software aligns with specific quality standards, guidelines, and best practices.
- Fosters communication and collaboration among team members. This helps ensure that everyone understands the software's design, code, and functionality.
- Promote a culture of continuous improvement by identifying areas for enhancement in the software development process.

3. Inspections

Inspections are a formal type of review that involves checking the documents thoroughly before a meeting and is carried out mostly by moderators. A meeting is then held to review the code and the design.

Inspection meetings can be held both physically and virtually. The purpose of these meetings is to review the code and the design with everyone and to report any bugs found.

Benefits

- It is easier to find defects for the people who have not done the implementation themselves and are unaware of its correctness beforehand.
- Knowledge sharing about specific software artifacts and designs.
- Knowledge sharing regarding defect detection practices.
- Flaws are identified at early stages.
- It reduces the rework and testing effort.

Who is involved?

- **Moderator:** Inspector who is responsible for organizing and reporting on inspection.
- **Author:** Owner of the report.
- **Reader:** A person who guides the examination of the product.
- **Recorder:** An inspector who notes down all the defects on the defect list.
- **Inspector:** Member of the inspection team.

Steps of inspection

Planning

The planning phase starts when the entry criteria for the inspection state are met. A moderator verifies that the product entry criteria are met.

Overview

In the overview phase, a presentation is given to the inspector with some background information needed to review the software product properly.

Preparation

This is considered an individual activity. In this part of the process, the inspector collects all the materials needed for inspection, reviews that material, and notes any defects.

Meeting

The moderator conducts the meeting. In the meeting, the defects are collected and reviewed.

Rework

The author performs this part of the process in response to defect disposition determined at the meeting.

Follow-up

In follow-up, the moderator makes the corrections and then compiles the inspection management and defects summary report.

UNIT-4

❖ 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. UI provides fundamental platform for human-computer interaction.

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

The software becomes more popular if its user interface is:

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

UI is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

1. Command Line Interface (CLI)

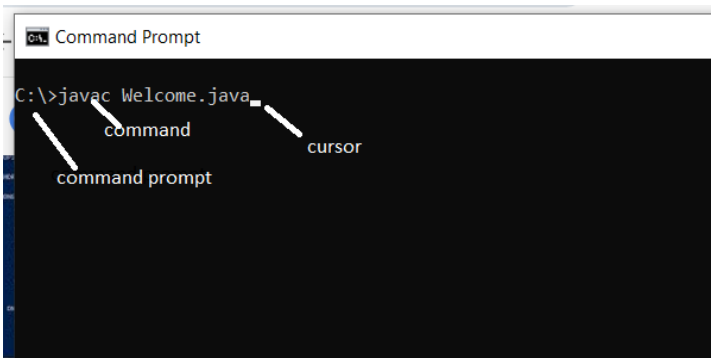
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

CLI Elements:



A text-based command line interface can have the following elements:

- **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
- **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

2. Graphical User Interface

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

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

GUI Elements:

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:

- **Window** - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.
- **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.
- **Menu** - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.

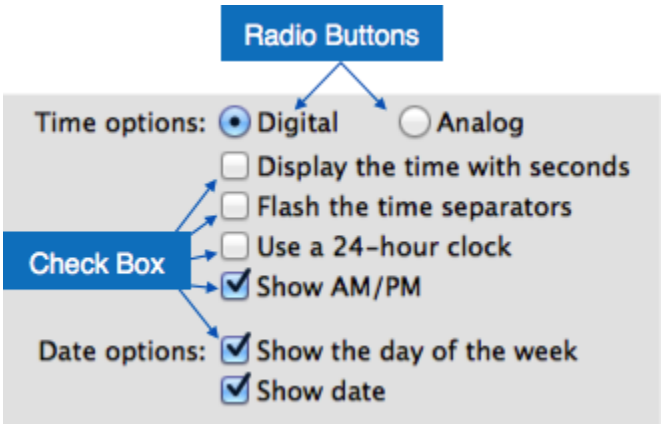
- **Icon** - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.
- **Cursor** - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

Application specific GUI components

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

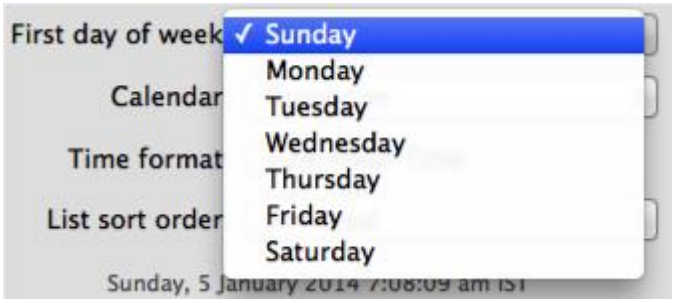
- **Application Window** - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
- **Dialogue Box** - It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.

- **Text-Box** - Provides an area for user to type and enter text-based data.
- **Buttons** - They imitate real life buttons and are used to submit inputs to the software.



- **Radio-button** - Displays available options for selection. Only one can be selected among all offered.

- **Check-box** - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
- **List-box** - Provides list of available items for selection. More than one item can be selected.



Other impressive GUI components are:

- Sliders
- Combo-box
- Data-grid
- Drop-down list

❖ Human factors" in design

Human factors refer to how users interact with systems, machines, platforms, or even tasks. As a discipline, it has strong military ties and is often associated with the aviation industry.

The complexity of cockpit controls, for example, increases the cognitive load on pilots, leading to accidents due to human error (that's why it's called 'human' factors). Product creators, similarly, look for ways to reduce cognitive load on users.

The cockpit of an Airbus A380 looks complex to anyone who doesn't have flight experience; similar, human factors design aims to make things easier when you're using a product or platform.

Human factors design (or *people-centered design*), specifically, focuses on improving areas within a product or design where interaction happens. Examples include when you use a touchscreen smartphone and when you perform tasks on your desktop computer.

The goal is to reduce the number of mistakes that users make and produce more comfortable interactions with a product. Human factors design is about understanding human capabilities and limitations and then applying this knowledge to product design. It's also a combination of many disciplines, including psychology, sociology, engineering, and industrial design.

Key human factor design principles

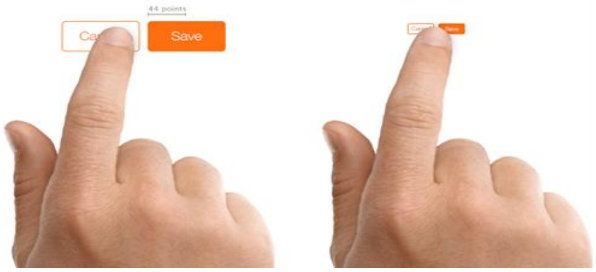
As we mentioned above, human factors are all about improving the interactions between machines and humans. To make this happen, you'll need to think about your user's capabilities and limitations, and then apply those to your product or web design.

Most of the human factors principles listed below come from the ISO 9241 standards for ergonomics of human-computer interaction. The principles mentioned in this section have one goal: helping the user engage with a product and get into a state of 'flow' when using it.

1. **Physical ergonomics**

Physical ergonomics refers to the human body's responses to physical work demands—for example, using the physical muscles of your hand to hold your smartphone or touch a screen. Proper ergonomic design is necessary to create comfortable interaction with a product.

This information helps human factor specialists design a product or device so that users can complete tasks efficiently and effectively. For example, when we apply human factors design in mobile app design, we size touch controls to minimize the risk of false actions.



Human factors design takes into account how a user interacts with the product, including using properly sized buttons versus buttons that are too small. Image credit Apple.

2. **Consistency**

This principle states that a system should look and work the same throughout. Consistency in design plays a key role in creating comfortable interactions. If a product uses consistent design, a user can transfer a learned skill to other parts of the product.

It's also important to maintain both internal and external consistency:

- **Internal consistency** – Apply the same conventions across all elements of the user interface. For example, when you design a graphical user interface (GUI), use the same visual appearance of UI elements throughout.
- **External consistency** – Use the same design across all platforms for the product, such as desktop, mobile, and so on.

3. **Familiarity**

The principle of familiarity states the importance of using familiar concepts and metaphors in the design of a human-computer interface. The design industry loves innovation, and it's very tempting for designers to create something new and unexpected. But at the same time, users love familiarity. As they spend time using products other than ours (Jakob's Law of Internet User Experience), they become familiar with standard design conventions and come to expect them.

Designers who reinvent the wheel and introduce unusual concepts increase the learning curve for their users. When the usage isn't familiar, users have to spend extra time learning how to interact with your product. To combat this, strive for intuitiveness by using patterns that people are already familiar with.

4. **Efficiency**

Users should be able to complete their tasks in the shortest possible time. As a designer, it's your job to reduce the user's cognitive load—-that is, it shouldn't require a ton of brain power to interact with the product.

Some tips to keep in mind:

- **Break down complex tasks into simple steps.** By doing that, you can reduce the complexity and simplify decision-making.
- **Reduce the number of operations required to complete the task.** Remove all extra actions and make navigation paths as short as possible. Make sure your user can dedicate all their time (and brainpower) to the task at hand, not the interface of a product.
- **Guide the user.** Guide your user to learn how to use the system by giving them all information upfront. Anticipate places where users might need extra help.
- **Offer shortcuts.** For seasoned users, it's important to offer shortcuts that can improve their productivity. An example would be keyboard shortcuts that help users complete certain operations without using a mouse.

5. **Error management**

To err is human. But that doesn't mean your users like it! The way a system handles errors has a tremendous impact on your users. This includes error prevention, error correction, and helping your user get back on track when an error does occur.

Here are a few things to remember when designing error handling:

- **Prevent errors from occurring whenever possible.** Create user journeys and analyze them to identify places in which users might face troubles.
- **Protect users from making fatal errors.** Create defensive layers that prevent users from getting fatal error states. For example, design system dialogs that ask users to confirm their action (such as deleting files or their entire account).

System dialog for 'Delete Account' operation.

- **Support 'Undo' operations.** Make it possible to reverse actions.
- **When an error does occur, provide messages** that help users solve the problem.
- **Never blame users.** If you practice user-centered design, you know that it's not the user's fault; instead, it's your design flaws that lead users to make mistakes.

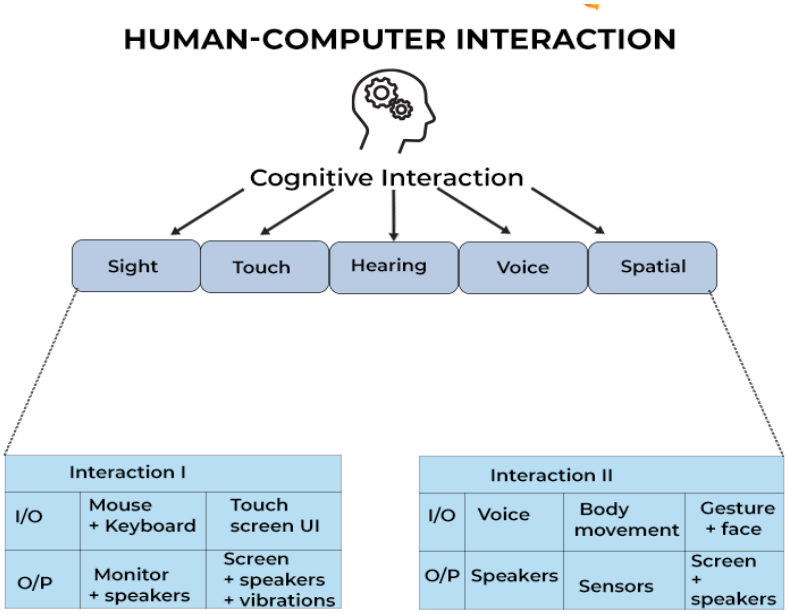
❖ Human computer interaction

Human-computer interaction (HCI) is the field of study that focuses on optimizing how users and computers interact by designing interactive computer interfaces that satisfy users' needs. It is a multidisciplinary subject covering computer science, behavioral sciences, cognitive science, ergonomics, psychology, and design principles.

The emergence of HCI dates back to the 1980s, when personal computing was on the rise. It was when desktop computers started appearing in households and corporate offices. HCI's journey began with video games, word processors, and numerical units.

However, with the advent of the internet and the explosion of mobile and diversified technologies such as voice-based and Internet of Things (IoT), computing became omnipresent and omnipotent. Technological competence further led to the evolution of user interactions. Consequently, the need for developing a tool that would make such man-machine interactions more human-like grew significantly. This established HCI as a technology, bringing different fields such as cognitive engineering, linguistics, neuroscience, and others under its realm.

Today, HCI focuses on designing, implementing, and evaluating interactive interfaces that enhance user experience using computing devices. This includes user interface design, user-centered design, and user experience design.



Key components of HCI

Fundamentally, HCI is made up of four key components:

1. The user

The user component refers to an individual or a group of individuals that participate in a common task. HCI studies users' needs, goals, and interaction patterns. It analyzes various parameters such as users' cognitive capabilities, emotions, and experiences to provide them with a seamless experience while interacting with computing systems.

2. The goal-oriented task

A user operates a computer system with an objective or goal in mind. The computer provides a digital representation of objects to accomplish this goal. For example, booking an airline for a destination could be a task for an aviation website. In such goal-oriented scenarios

3. The interface

The interface is a crucial HCI component that can enhance the overall user interaction experience. Various interface-related aspects must be considered, such as interaction type (touch, click, gesture, or voice), screen resolution, display size, or even color contrast. Users can adjust these depending on the user's needs and requirements.

For example, consider a user visiting a website on a smartphone. In such a case, the mobile version of the website should only display important information that allows the user to navigate through the site easily. Moreover, the text size should be appropriately adjusted so that the user is in a position to read it on the mobile

device. Such design optimization boosts user experience as it makes them feel comfortable while accessing the site on a mobile phone.

4. The context

HCI is not only about providing better communication between users and computers but also about factoring in the context and environment in which the system is accessed. For example, while designing a smartphone app, designers need to evaluate how the app will visually appear in different lighting conditions (during day or night) or how it will perform when there is a poor network connection. Such aspects can have a significant impact on the end-user experience.

Importance of HCI

HCI is crucial in designing intuitive interfaces that people with different abilities and expertise usually access. Most importantly, human-computer interaction is helpful for communities lacking knowledge and formal training on interacting with specific computing systems.

With efficient HCI designs, users need not consider the intricacies and complexities of using the computing system. User-friendly interfaces ensure that user interactions are clear, precise, and natural.

Let's understand the importance of HCI in our day-to-day lives:

1. HCI in daily lives

Today, technology has penetrated our routine lives and has impacted our daily activities. To experience HCI technology, one need not own or use a smartphone or computer. When people use an ATM, food dispensing machine, or snack vending machine, they inevitably come in contact with HCI. This is because HCI plays a vital role in designing the interfaces of such systems that make them usable and efficient.

2. Industry

Industries that use computing technology for day-to-day activities tend to consider HCI a necessary business-driving force. Efficiently designed systems ensure that employees are comfortable using the systems for their everyday work. With HCI, systems are easy to handle, even for untrained staff.

HCI is critical for designing safety systems such as those used in air traffic control (ATC) or power plants. The aim of HCI, in such cases, is to make sure that the system is accessible to any non-expert individual who can handle safety-critical situations if the need arises.

3. Accessible to disabled

The primary objective of HCI is to design systems that make them accessible, usable, efficient, and safe for anyone and everyone. This implies that people with a wide range of capabilities, expertise, and knowledge can easily use HCI-designed systems. It also encompasses people with disabilities. HCI tends to rely on user-centered techniques and methods to make systems usable for people with disabilities.

4. An integral part of software success

HCI is an integral part of software development companies that develop software for end-users. Such companies use HCI techniques to develop software products to make them usable. Since the product is finally consumed by the end-user, following HCI methods is crucial as the product's sales depend on its usability.

❖ Examples of HCI and Goals of HCI

Examples of HCI

Technological development has brought to light several tools, gadgets, and devices such as wearable systems, voice assistants, health trackers, and smart TVs that have advanced human-computer interaction technology.

Let's look at some prominent examples of HCI that have accelerated its evolution.

1. IoT technology

IoT devices and applications have significantly impacted our daily lives. According to a May 2022 report by IoT Analytics, global IoT endpoints are expected to reach 14.4 billion in 2022 and grow to 27 billion (approx.) by 2025. As users interact with such devices, they tend to collect their data, which helps understand different user interaction patterns. IoT companies can make critical business decisions that can eventually drive their future revenues and profits.

A recent development in the field of HCI introduced the concept of 'pre-touch sensing' through pre-touch phones. This means the phone can detect how the user holds the phone or which finger approaches the screen first for operation. Upon detecting the user's hand movements, the device immediately predicts the user's intentions and performs the task before the user gives any instructions.

2. Eye-tracking technology

Eye-tracking is about detecting where a person is looking based on the gaze point. Eye-tracking devices use cameras to capture the user's gaze along with some embedded light sources for clarity. Moreover, these devices use machine learning algorithms and image processing capabilities for accurate gaze detection.

Businesses can use such eye-tracking systems to monitor their personnel's visual attention. It can help companies manage distractions that tend to trouble their employees, enhancing their focus on the task. In this manner, eye-tracking technology, along with HCI-enabled interactions, can help industries monitor the daily operations of their employees or workers.

3. Speech recognition technology

Speech recognition technology interprets human language, derives meaning from it, and performs the task for the user. Recently, this technology has gained significant popularity with the emergence of chatbots and virtual assistants.

For example, products such as Amazon's Alexa, Microsoft's Cortana, Google's Google Assistant, and Apple's Siri employ speech recognition to enable user interaction with their devices, cars, etc. The combination of HCI and speech recognition further fine-tune man-machine interactions that allow the devices to interpret and respond to users' commands and questions with maximum accuracy. It has various applications, such as transcribing conference calls, training sessions, and interviews.

4. AR/VR technology

AR and VR are immersive technologies that allow humans to interact with the digital world and increase the productivity of their daily tasks. For example, smart glasses enable hands-free and seamless user interaction with computing systems. Consider an example of a chef who intends to learn a new recipe. With smart glass technology, the chef can learn and prepare the target dish simultaneously.

5. Cloud computing

Today, companies across different fields are embracing remote task forces. According to a 'Breaking Barriers 2020' survey by Fuze (An 8×8 Company), around 83% of employees feel more productive working remotely. Considering the current trend, conventional workplaces will witness a massive rejig and transform entirely in a couple of decades. Thanks to cloud computing and human-computer interaction, such flexible offices have become a reality.

Goals of HCI

The principal objective of HCI is to develop functional systems that are usable, safe, and efficient for end-users. The developer community can achieve this goal by fulfilling the following criteria:

- Have sound knowledge of how users use computing systems
- Design methods, techniques, and tools that allow users to access systems based on their needs

- Adjust, test, refine, validate, and ensure that users achieve effective communication or interaction with the systems
- Always give priority to end-users and lay the robust foundation of HCI

To realize the above points, developers must focus on two relevant areas: usability and user experience. Let's look at each category in detail:

1. **Usability**

Usability is key to HCI as it ensures that users of all types can quickly learn and use computing systems. A practical and usable HCI system has the following characteristics:

- **How to use it:** This should be easy to learn and remember for new and infrequent users to learn and remember. For example, operating systems with a user-friendly interface are easier to understand than DOS operating systems that use a command-line interface.
- **Safe:** A safe system safeguards users from undesirable and dangerous situations. This may refer to users making mistakes and errors while using the system that may lead to severe consequences. Users can resolve this through HCI practices. For example, systems can be designed to prevent users from activating specific keys or buttons accidentally.
- **Efficient**: An efficient system defines how good the system is and whether it accomplishes the tasks that it is supposed to. Moreover, it illustrates how the system provides the necessary support to users to complete their tasks.
- **Effective**: A practical system provides high-quality performance. It describes whether the system can achieve the desired goals.
- **Enjoyable**: Users find the computing system enjoyable to use when the interface is less complex to interpret and understand.

2. User experience

User experience is a subjective trait that focuses on how users feel about the computing system when interacting with it. Here, user feelings are studied individually so that developers and support teams can target particular users to evoke positive feelings while using the system.

HCI systems classify user interaction patterns into the following categories and further refine the system based on the detected pattern:

- **Desirable traits** – satisfying, enjoyable, motivating, or surprising
- **Undesirable traits** – Frustrating, unpleasant, or annoying

3. Takeaway

Cleverly designed computer interfaces motivate users to use digital devices in this modern technological age. HCI enables a two-way dialog between man and machine. Such effective communication makes users believe they are interacting with human personas and not any complex computing system. Hence, it is crucial to build a strong foundation of HCI that can impact future applications such as personalized marketing, eldercare, and even psychological trauma recovery.

❖ IMPORTANCE OF GOOD DESIGN

The following experimental design principles are considered, when evaluating a current user interface, or designing a new user interface:

- **Early Focus is Placed on the User and Task:** How many users are needed to perform the task is established and who the appropriate users should be is determined (someone who has never used the interface, and will not use the interface in the future, is most likely not a valid user). In addition, the task the users will be performing and how often the task needs to be performed is defined.
- **Empirical Measurement:** The interface is tested with real users who meet the interface daily. The results can vary with the performance level of the user and the typical human computer interaction may not always be represented. Quantitative usability specifics, such as the number of users performing the task, the time to complete the task, and the number of errors made during the task are determined.
- **Iterative Design:** After determining what users, tasks, and empirical measurements to include, the following iterative design steps are performed:

    1. Design the User Interface
    2. Test
    3. Analyze Results
    4. Repeat

The iterative design process is repeated until a sensible, user-friendly interface is created.

Methodologies

Various strategies delineating methods for human–PC interaction design have developed since the ascent of the field during the 1980s. Most plan philosophies come from a model for how clients, originators, and specialized frameworks interface. Early techniques treated clients' psychological procedures as unsurprising and quantifiable and urged plan specialists to look at subjective

science to establish zones, (for example, memory and consideration) when structuring UIs. Presentday models, in general, centre around a steady input and discussion between clients, creators, and specialists and push for specialized frameworks to be folded with the sorts of encounters clients need to have, as opposed to wrapping user experience around a finished framework.

- **Activity Theory:** Utilized in HCI to characterize and consider the setting where human cooperation with PCs occurs. Action hypothesis gives a structure for reasoning about activities in these specific circumstances and illuminates design of interactions from an action-driven perspective.
- **User-Focused Design (UFD):** Client Focused Structure (CFS) is a cutting edge, broadly rehearsed plan theory established on the possibility that clients must become the overwhelming focus in the plan of any PC framework. Clients, architects, and specialized experts cooperate to determine the requirements and restrictions of the client and make a framework to support these components. Frequently, client-focused plans are informed by ethnographic investigations of situations in which clients will associate with the framework. This training is like participatory design, which underscores the likelihood for end-clients to contribute effectively through shared plan sessions and workshops.
- **Principles of UI Design:** These standards may be considered during the design of a client interface: resistance, effortlessness, permeability, affordance, consistency, structure, and feedback.
- **Value Delicate Design:** A technique for building innovation that accounts for the individuals who utilize the design straightforwardly, and just as well for those who the design influences, either directly or indirectly. VSD utilizes an iterative plan process that includes three kinds of examinations: theoretical, exact, and specialized.

Benefits of Good Design

- Screens looks very friendly and less messy.
- Help in understanding the overall design and so easy to start communication without wasting much time.
- Help in reducing the training time so the cost as well.
- The organization customers benefit because of improved services.
- Less user support costs.
- Increases employee satisfaction is increased because aggravation and frustration are reduced.
- Increased productivity

UNIT-5

❖ Quality Metrics

This post gives a high-level overview of 14 metrics every quality executive should consider monitoring, depending on your specific goals and improvement needs.

1. Cost of Quality

Cost of quality is one of the most important, yet often overlooked, metrics to monitor. The true cost of quality includes both the cost of poor quality and investments in good quality. ASQ, or the American Society of Quality, developed the following formula for Cost of Quality:

COQ = Cost of Good Quality + Cost of Poor Quality

2. Defects

There are a couple ways to look at defects that tend to confuse people:

- **Defective parts per million (DPPM):** Interchangeably called parts per million (PPM) or defects per million (DPM), you can calculate DPPM with the following formula:

$$\left(\frac{Defective\ parts}{Total\ parts}\right) \times 1{,}000{,}000$$

- **Defects per million opportunities (DPMO**): This metric is more useful when looking at defects in subassemblies, which may have multiple opportunities for failure. Calculate DPMO with the following formula:

$$\text{DPMO} = \frac{1{,}000{,}000 \times \text{number of defects}}{\text{number of units} \times \text{number of defects opportunities per unit}}$$

3. Customer Complaints and Returns

Closely monitoring customer issues is the only way to systematically prevent them. Figures to help you track customer-related issues include:

- Complaints, rejects or returns over a specific period
- Number resolved during a specific period
- Average taken to resolve customer complaints
- Warranty costs

4. Scrap

Scrap rate is the percentage of materials sent to production that never become part of finished products. In addition, you'll want to keep a close eye on total scrap costs.

Scrap to include in your calculations would be: vendor scrap, internal scrap, and internal

setup scrap. Manufacturers usually have their own internal ways of calculating scrap, for example some companies would not include setup scrap, so its important to check with your company on what to include.

An easy way to calculate scrap is:

$$Scrap\ rate = \frac{Total\ scrap}{Total\ product\ run}$$

5. Yield

Yield is a classic measure of process or plant effectiveness. Beyond total yield, consider monitoring first-pass yield (FPY), the percentage of products manufactured correctly the first time through without rework.

For example:

- 200 units enter A and 150 leave. The FPY for process A is 150/200 = .75
- 150 units go into B and 145 units leave. The FPY for process B is 145/150 = .97
- 145 units go into C and 130 leave. The FPY for C is 130/145 = .89
- 130 units got into D and 129 leave. The FPY for D is 129/130 = .99

Total process yield = FPY(A)x FPY(B)x FPY(C)x FPY(D)
=.75*.97*.89*.99=.64

6. Overall Equipment Effectiveness (OEE)

Overall Equipment Effectiveness (OEE) is an important measure of productivity and efficiency, calculated in simple terms as availability multiplied by performance and quality. Here's a more detailed look at each of those component metrics:

OEE = Availability × Performance × Quality

7. Throughput

Throughput is the quantity of goods produced over a given time period. You can measure throughput:

- Per machine
- Per product line
- For the entire plant

$$Throughput = \frac{Total\ units}{processing\ time} \ x \ \frac{processing\ time}{Total\ time} \ x \ \frac{Good\ units}{Total\ units}$$

8. Supplier Quality Metrics

Suppliers have a huge impact on quality costs. Metrics to track here include:

- **Supplier defect rate**: Percentage of materials from suppliers not meeting quality specifications
- **Supplier chargebacks**: Total charged to suppliers for cost of non-conforming materials (possibly including late delivery and payroll costs)
- **Incoming supplier quality**: Percentage of materials received meeting quality requirements

9. Delivery Metrics

There are two crucial metrics you should be measuring with regards to delivery from a customer satisfaction and efficiency perspective:

- **On-time delivery (OTD)** is calculated as the percentage of units delivered within the OTD window.
- **Perfect order metric (POM)** or fill rate is the percentage of orders that arrive complete, on time, damage-free and with a correct invoice.

It's harder to achieve a good POM considering that each component of this metric gets multiplied together:

$$POM = (\%complete) \times (\%on\ time) \times (\%damage\ free) \times (\%correctly\ invoiced)$$

10. Internal Timing Efficiency Metrics

A number of metrics provide insight into how efficiently your facility runs in terms of timing. A few basics include:

- **Manufacturing Cycle Time:** How much time it takes from order to production to finished goods Throughput time = Process time + Inspection time + move time + Queue time
- **Changeover Time:** How much time it takes to switch a line to another product, which can last anywhere from a few minutes to several weeks
- **Change order cycle time:** Average time to execute change orders from documentation through production
- **New product introduction (NPI) rate:** Average time to introduce a new product to market

11. Capacity Utilization Rate

Capacity utilization is the percentage of total output capacity used at any given point. This KPI can help with strategic planning and is also an indicator of market demand.

$$Capacity\ utilization\ rate = \frac{Actual\ output}{Potential\ output} \times 100$$

12. Schedule Realization

This metric tells you how often your plant reaches production targets over a given period of time. A simple calculation is orders completed by scheduled date divided by total number of orders.

$$Schedule\ realization = \frac{Orders\ completed\ by\ scheduled\ date}{Total\ number\ of\ orders}$$

❖ Software Reliability

Software Reliability means **Operational reliability**. It is described as the ability of a system or component to perform its required functions under static conditions for a specific period.

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the input are free of error.

Software Reliability is an essential connect of software quality, composed with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve because the complexity of software turn to be high. While any system with a high degree of complexity, containing software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the speedy growth of system size and ease of doing so by upgrading the software.

For example, large next-generation aircraft will have over 1 million source lines of software on- board; next-generation air traffic control systems will contain between one and two million lines; the upcoming International Space Station will have over two million lines on-board and over 10 million lines of ground support software; several significant life-critical defense systems will have over 5 million source lines of software. While the complexity of software is inversely associated with software reliability, it is directly related to other vital factors in software quality, especially functionality, capability, etc.

❖ Software Testing

Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free. It involves

execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Some prefer saying Software testing as a White Box and Black Box Testing. In simple terms, Software Testing means the Verification of Application Under Test (AUT). This tutorial introduces testing software to the audience and justifies its importance

Benefits of Software Testing

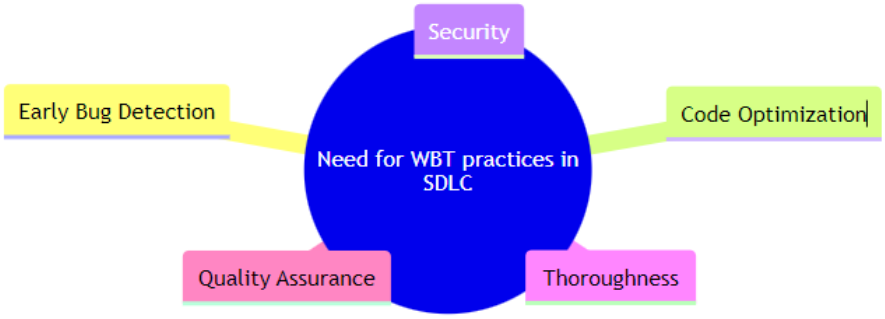
- **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience.

❖ White Box Testing (or) structural testing

White Box Testing,or glass box testing, is a software testing technique that focuses on the software's internal logic, structure, and coding. It provides testers with complete application knowledge, including access to source code and design documents, enabling them to inspect and verify the software's inner workings, infrastructure, and integrations.

Test cases are designed using an internal system perspective, and the methodology assumes explicit knowledge of the software's internal structure and implementation details. This in-depth visibility allows White Box Testing to identify issues that may be invisible to other testing methods.
Unlike black box testing, which focuses on testing the software's functionality without knowledge of its internal workings, white box testing involves looking inside the application and understanding its code, logic, and structure.

perform White Box Testing



White Box Testing practices are integral to the Software Development Life Cycle (SDLC) for several reasons:

- **Early Bug Detection:** White Box Testing allows for detecting bugs and errors early in development. This early detection can save time, effort, and resources, as fixing bugs later in the development process can be more complex and costly.
- **Code Optimization:** Identify redundant code and software areas that can be optimized. This leads to more efficient and streamlined software.
- **Security:** Uncover security vulnerabilities in the code. By examining the internal structure of the software, testers can identify potential security risks and ensure that security best practices have been followed.
- **Thoroughness:** It examines all the internal workings of the software. This thoroughness ensures that every part of the code is tested and validated, leading to robust and reliable software.
- **Quality Assurance:** White Box Testing is a critical part of ensuring software quality. By testing the software's internal structure, White Box Testing ensures that the software functions as expected and meets the required standards.
  White Box Testing practices are crucial to the SDLC, contributing to developing high-quality, secure, and efficient software.

Types of White Box Testing

Different types of White Box Testing are:

- **Unit Testing:** Imagine you're building a bicycle. Unit testing would be like checking each part separately – testing the brakes, the gears, the pedals, etc., to ensure they all work correctly before assembling the whole bicycle.

- **Static Analysis:** This is like proofreading a book before it's published. You're looking for errors in grammar, punctuation, and sentence structure. Still, you need to read the book as a whole to understand the story (which would be more like dynamic analysis).
- **Dynamic Analysis:** This would be like test-driving a car. You're not just looking at the components (like in static analysis), but you're driving the car to see how it performs on the road.
- **Statement Coverage:** Imagine you're a teacher checking a student's homework. Statement coverage would be like ensuring the student has answered every question on the assignment.
- **Branch Testing:** This is like exploring all possible routes on a GPS. If you're at an intersection, branch testing involves going straight, turning left, and turning right to ensure all paths lead to valid destinations.
- **Path Testing:** This would be like a postman ensuring they can deliver mail to every house on their route. They need to make sure every possible path is covered.
- **Loop Testing:** This is like checking a playlist on repeat. You want to ensure it loops back to the first song correctly after the last song finishes.

❖ Path Testing & Basis Path Testing with EXAMPLES

Path Testing

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program.

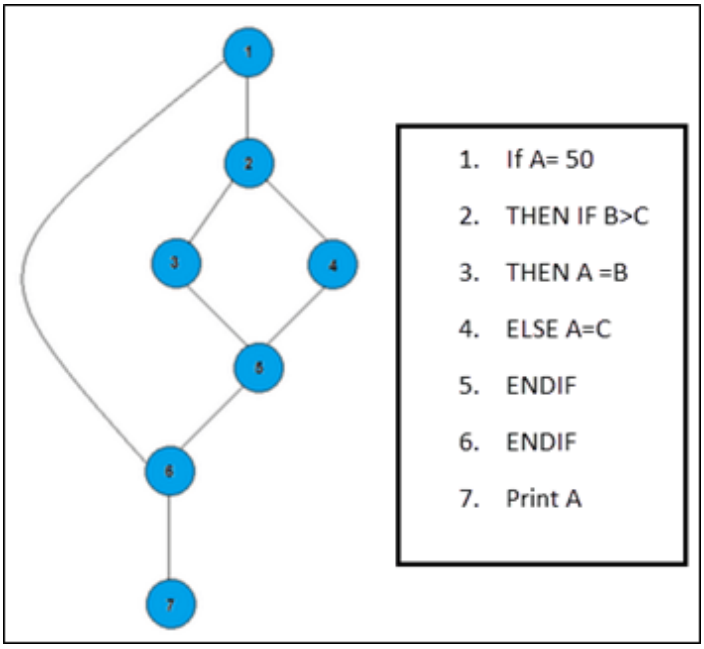
Any software program includes, multiple entry and exit points. Testing each of these points is a challenging as well as time-consuming. In order to reduce the redundant tests and to achieve maximum test coverage, basis path testing is used.

Basis Path Testing

Basis Path Testing in software engineering is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program. The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.

In software engineering, Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases. It is a hybrid method of branch testing and path testing methods.

Here we will take a simple example, to get a better idea what is basis path testing include



In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

- **Path 1**: 1,2,3,5,6, 7
- **Path 2**: 1,2,4,5,6, 7
- **Path 3**: 1, 6, 7

Steps for Basis Path testing

The basic steps involved in basis path testing include

- Draw a control graph (to determine different program paths)
- Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- Find a basis set of paths
- Generate test cases to exercise each path

Advantages of Basic Path Testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in a program at least once

❖ Control Structure Testing

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. *Condition Testing :*

Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on.

The common types of logical conditions that are tested using condition testing are-

1. A relation expression, like E1 op E2 where 'E1' and 'E2' are arithmetic expressions and 'OP' is an operator.
2. A simple condition like any relational expression preceded by a NOT (~) operator. For example, (~E1) where 'E1' is an arithmetic expression and '~' denotes NOT operator.
3. A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis.

   For example, (E1 & E2)|(E2 & E3) where E1, E2, E3 denote arithmetic expression and '&' and '|' denote AND or OR operators.
4. A Boolean expression consists of operands and a Boolean operator like 'AND', OR, NOT.

   For example, 'A|B' is a Boolean expression where 'A' and 'B' denote operands and | denotes OR operator.

2. *Data Flow Testing :*

It is a group of testing approaches used to observe the control flow of programs to discover the sequence of variables as per the series of events.

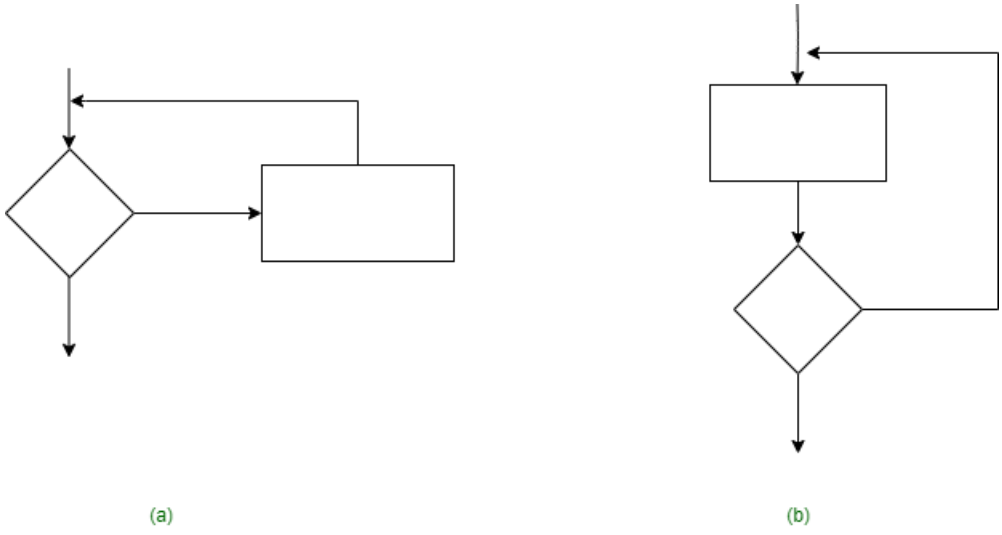
It implements a control flow graph and analysis the points where the codes can change the data.

If we execute the data flow testing technique, the information is kept safe and unchanged during the code's implementation.

3. *Loop Testing :*

Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction.
Following are the types of loops.

1. **Simple Loop –** The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n.
    1. Skip the entire loop.
    2. Traverse the loop only once.
    3. Traverse the loop two times.
    4. Make p passes through the loop where p<n.
    5. Traverse the loop n-1, n, n+1 times.



SIMPLE LOOPS

Concatenated Loops – If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested loops.

1. **Nested Loops –** Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases.

The following steps for testing nested loops are as follows-

6. Start with inner loop. set all other loops to minimum values.
7. Conduct simple loop testing on inner loop.
8. Work outwards.
9. Continue until all loops tested.

2. **Unstructured loops –** This type of loops should be redesigned, whenever possible, to reflect the use of unstructured the structured programming constructs.

❖ Black box testing

Black box testing is a type of software testing in which the functionality of the software is not known. The testing is done without the internal knowledge of the products.

Black box testing can be done in following ways:

1. Syntax Driven Testing – This type of testing is applied to systems that can be syntactically represented by some language. For example- compilers,language that can be represented by context free grammar. In this, the test cases are generated so that each grammar rule is used at least once.

2. Equivalence partitioning – It is often seen that many type of inputs work similarly so instead of giving all of them separately we can group them together and test only one input of each group. The idea is to partition the input domain of the system into a number of equivalence classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error.

The technique involves two steps:

1. **Identification of equivalence class –** Partition any input domain into minimum two sets: **valid values** and **invalid values**. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.
2. **Generating test cases –**

(i) To each valid and invalid class of input assign unique identification number.
(ii) Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.

To calculate the square root of a number, the equivalence classes will be:

(a) Valid inputs:

- Whole number which is a perfect square- output will be an integer.
- Whole number which is not a perfect square- output will be decimal number.
- Positive decimals

(b) Invalid inputs:

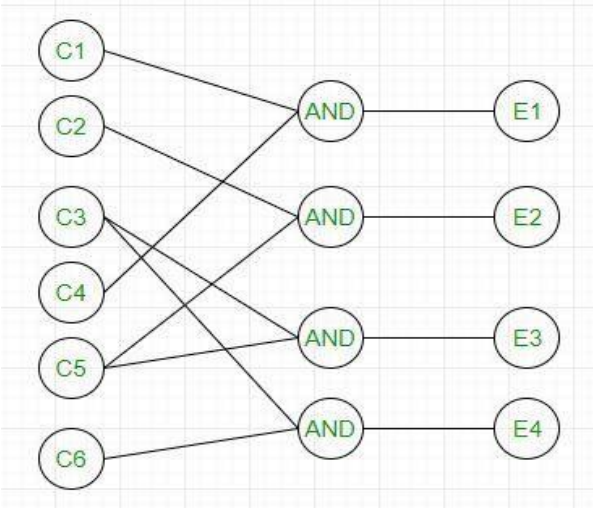
- Negative numbers(integer or decimal).
- Characters other that numbers like "a","!",";",etc.

3. Boundary value analysis – Boundaries are very good places for errors to occur. Hence if test cases are designed for boundary values of input domain then the efficiency of testing improves and probability of finding errors also increase. For example – If valid range is 10 to 100 then test for 10,100 also apart from valid and invalid inputs.

4. Cause effect Graphing – This technique establishes relationship between logical input called causes with corresponding actions called effect. The causes and effects are represented using Boolean graphs. The following steps are followed:

1. Identify inputs (causes) and outputs (effect).
2. Develop cause effect graph.
3. Transform the graph into decision table.
4. Convert decision table rules to test cases.

For example, in the following cause effect graph:



It can be converted into decision table like:

		1	2	3	4
CAUSES	C1	1	0	0	0
	C2	0	1	0	0
	C3	0	0	1	1
	C4	1	0	0	0
	C5	0	1	1	0
	C6	0	0	0	1
EFFECTS	E1	x	-	-	-
	E2	-	x	-	-
	E3	-	-	x	-
	E4	-	-	-	x

Each column corresponds to a rule which will become a test case for testing. So there will be 4 test cases.

5. Requirement based testing – It includes validating the requirements given in SRS of software system.

6. Compatibility testing – The test case result not only depend on product but also infrastructure for delivering functionality. When the infrastructure parameters are changed it is still expected to work properly. Some parameters that generally affect compatibility of software are:

1. Processor (Pentium 3,Pentium 4) and number of processors.
2. Architecture and characteristic of machine (32 bit or 64 bit).
3. Back-end components such as database servers.
4. Operating System (Windows, Linux, etc).

Levels of Black Box Testing: Integration, Validation, and System Testing

1. Integration Testing

- **Purpose**: Integration Testing focuses on verifying that different modules or components work together as expected.
- **Process**:
    - **Big Bang Integration**: All components are integrated simultaneously and tested as a whole. Simple but may delay bug detection.
    - **Incremental Integration**: Integrates components gradually, testing each interaction step-by-step.
        - **Top-Down Integration**: Integrates modules from top to bottom, verifying each layer before moving downward.
        - **Bottom-Up Integration**: Integrates modules from bottom to top, testing lower-level modules first.

2. Validation Testing

- **Purpose**: Validation Testing ensures the software meets user needs and expectations by confirming it operates correctly in real-world scenarios.
- **Process**:
  - **Requirement Validation**: Checks if the software meets functional requirements (e.g., feature behavior, outputs).
  - **Non-Functional Validation**: Confirms that the software meets performance, usability, and security requirements.

3. System Testing

- **Purpose**: System Testing tests the complete and integrated system as a whole, validating both functional and non-functional requirements.
- **Process**:
  - **End-to-End Testing**: Verifies system functionality from start to finish, simulating real user workflows.
  - **Performance Testing**: Assesses system responsiveness and stability under load.
  - **Security Testing**: Evaluates system security, checking for vulnerabilities and ensuring data protection.
  - **Usability Testing**: Confirms that the software is user-friendly and meets accessibility standards.

Advantages of Black Box Testing

1. **User-Centric**: Evaluates software from the user's perspective, helping ensure it meets real-world expectations.
2. **No Need for Code Knowledge**: Allows testers to focus on functionality without requiring detailed knowledge of the code.
3. **Broad Coverage**: Can test the entire system, including inputs, outputs, and interactions, ensuring end-to-end functionality.

Disadvantages of Black Box Testing

1. **Limited Debugging Capability**: Does not examine internal code, making it harder to identify root causes of defects.
2. **Risk of Missing Paths**: May not cover all possible paths, especially if test cases are not comprehensive.
3. **Potential Overlap with White-Box Testing**: Black-box testing does not always detect specific logic errors or hidden bugs within the code.

❖ Software Re-engineering

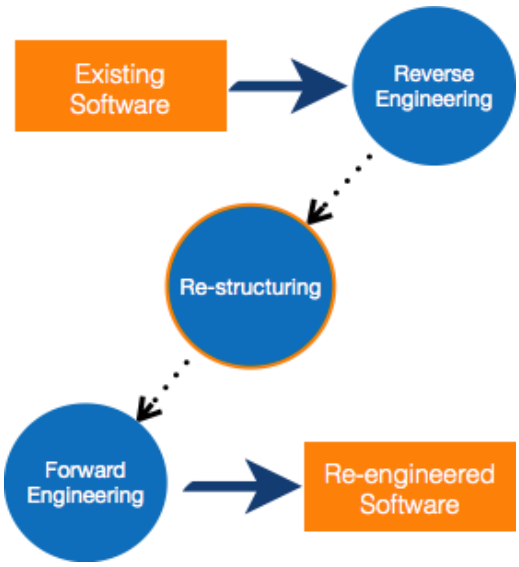
When we need to update the software to keep it to the current market, without

impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.



Re-Engineering Process

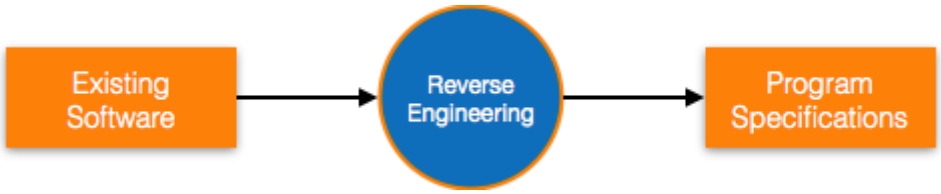
- **Decide** what to re-engineer. Is it whole software or a part of it?
- **Perform** Reverse Engineering, in order to obtain specifications of existing software.
- **Restructure Program** if required. For example, changing function-oriented programs into object- oriented programs.
- **Re-structure data** as required.
- **Apply Forward engineering** concepts in order to get re-engineered software.

There are few important terms used in Software re-engineering

❖ Reverse Engineering

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.



Program Restructuring

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

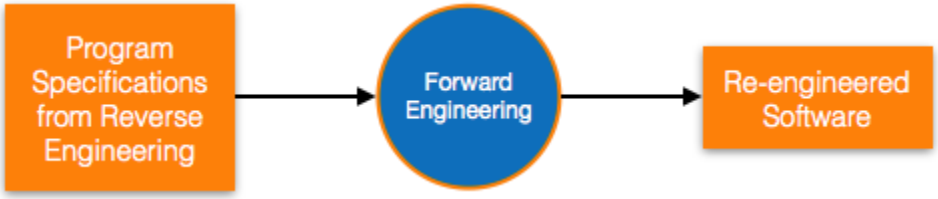
Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

Forward Engineering

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.



❖ Software Case Tools

CASE stands for **C**omputer **A**ided **S**oftware **E**ngineering. It means, development and maintenance of software projects with help of various automated software tools.

CASE Tools

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

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

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

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.

CASE tools can be grouped together if they have similar functionality, process activities and capability of getting integrated with other tools.

Scope of Case Tools

The scope of CASE tools goes throughout the SDLC.
Case Tools Types

Now we briefly go through various CASE tools

Diagram tools

These tools are used to represent system components, data and control flow among various software components and system structure in a graphical form. For example, Flow Chart Maker tool for creating state-of-the-art flowcharts.

Process Modeling Tools

Process modeling is method to create software process model, which is used to develop the software. Process modeling tools help the managers to choose a process model or modify it as per the requirement of software product. For example, EPF Composer

Project Management Tools

These tools are used for project planning, cost and effort estimation, project scheduling and resource planning. Managers have to strictly comply project execution with every mentioned step in software project management. Project management tools help in storing and sharing project information in real-time throughout the organization. For example, Creative Pro Office, Trac Project, Basecamp.

Analysis Tools

These tools help to gather requirements, automatically check for any inconsistency, inaccuracy in the diagrams, data redundancies or erroneous omissions. For example, Accept 360, Accompa, CaseComplete for requirement analysis, Visible Analyst for total analysis.

Design Tools

These tools help software designers to design the block structure of the software, which may further be broken down in smaller modules using refinement techniques. These

tools provides detailing of each module and interconnections among modules. For example, Animated Software Design

Configuration Management Tools

An instance of software is released under one version. Configuration Management tools deal with –

- Version and revision management
- Baseline configuration management
- Change control management

CASE tools help in this by automatic tracking, version management and release management. For example, Fossil, Git, Accu REV.